

**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 | 郑旭然 | | 院系 | 软件工程 | | |
| 班级 | 2037102 | | 学号 | 120L020719 | | |
| 任课教师 | 李全龙 | | 指导教师 | 李全龙 | | |
| 实验地点 | 格物楼207 | | 实验时间 | 2022.9.28 | | |
| 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。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： |
| 1. **Socket 编程的客户端和服务器端主要步骤**   **客户端软件流程**  1. 根据目标服务器IP地址与端口号创建套接字（socket）  2. 连接服务器（connect）：三次握手  3. 发送请求报文（send）  4. 接收返回报文（recv）  5. 关闭连接（closesocket）  **服务器端软件流程**  1. 创建套接字（socket），绑定套接字的本地IP地址和端口号（bind），然后转到监听模式并设置连接请求队列大小（listen）。  2. 从连接请求队列中取出一个连接请求，并同意连接（accept）。在TCP连接过程中进行了三次握手。  3. 收到请求报文（recv）  4. 发送数据（send）  5. 关闭连接（closesocket）  对于UDP协议上的通信，无需提前建立连接，只需在开始时建立相应的socket，进入无限循环，接收消息后直接与源地址进行通信即可。对于TCP协议上的通信，服务器需要有一个socket负责控制，在进入无限循环前建立绑定指定的端口号，并在无限循环内，对于每一个连接新建TCP连接与源主机进行通信即可。    图 1 网络应用的Socket API(TCP)调用基本流程   1. **HTTP 代理服务器的基本原理**   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 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。  我们实验中设计的是多用户代理服务器。多用户的简单代理服务器可以实现为一个多线程并发服务器。首先，代理服务器创建 HTTP 代理服务的 TCP 主套接字，通过该主套接字监听等待客户端的连接请求。当客户端连接之后，创建一个子线程，由子线程执行上述一对一的代理过程，服务结束之后子线程终止。与此同时，主线程继续接受下一个客户的代理服务。   1. **HTTP 代理服务器的程序流程图**     图 3 HTTP 代理服务器的程序流程图   1. **实现 HTTP 代理服务器的关键技术及解决方案**   首先我们给出一些前提。定义宏常量，设置我们屏蔽了今日哈工大，钓鱼时从教务处钓鱼到选课系统。我们都是使用HTTP协议而不是HTTPS协议。    **实现基本代理：**我们考察示例代码中一开始就给出的四个函数，由此展开：  **BOOL InitSocket();** 用于加载套接字库，使用了以下几个函数：  socket(AF\_INET, SOCK\_STREAM, 0);  bind(ProxyServer, (SOCKADDR\*)&ProxyServerAddr, sizeof(SOCKADDR))；  listen(ProxyServer, SOMAXCONN);  实现了服务器的socket()、bind()、listen()。初始化一个套接字，bind()将套接字与服务器的host地址绑定，并且绑定端口号。最后用listen()监听。  **void ParseHttpHead(char \*buffer,HttpHeader \* httpHeader);** 解析请求报文头，get到报文的method、url、host等。ConnectToServer()将于目标服务器将于目标服务器建立连接。  **BOOL ConnectToServer(SOCKET \*serverSocket,char \*host);** 使用socket创建套接字，连接到目标服务器。  **unsigned int \_\_stdcall ProxyThread(LPVOID lpParameter);** 这里从客户端接受请求报文并发送给服务器，再发送给客户端响应报文。recv()和send()接受客户端的HTTP请求，并通过代理服务器将该请求转发给服务器；服务器将响应发给代理服务器，代理服务器再将响应发送给客户端。  另外在主函数中accept()函数对请求进行接收和响应，对每一个请求代理服务器都是新建一个子线程来接受的；最后处理完成后，关闭线程并清理缓存，然后接收下一个请求。  **实现cache功能：**将所有的请求的文件保存在本地上。代理服务器第一次与客户端通信后，会留下cache；当客户端再次访问相同的文件时，代理服务器向服务器发送一个请求，该请求需要增加“If-Modified-Since”的头，将服务器发过来的Last-Modified发送回去，给出服务器缓存资源最后修改的时间，服务器通过对比最后修改时间来判断缓存是否过期，如果服务器返回状态码304，表示内容是最新的，代理服务器直接将缓存发送给客户端；如果缓存过期，服务器返回状态码200，目标服务器返回一个新的响应，代理服务器接收后将该响应发回给客户端，并更新本地缓存。  在代码中，我们修改ParseHttpHead()函数解析HTTP头部，判断其中URL是否已经在缓存中。ParseHttpHead()返回Have\_cache，可以告知请求页面在代理上是否有缓存，如果有缓存则在客户端请求报文首部插入“If-Modified-Since”的头发送给目标服务器，其返回数据后读取返回的状态及页面最后修改时间。状态码为304或200，按照上面所说的进行操作。如果有错误则跳转到error关闭套接字，结束线程处理。这部分代码范围大，暂时不在此处贴出，后附全部代码。  **网站屏蔽：**在ProxyThread()中ParseHttpHead()解析TCP报文中的HTTP头部，将报文头部的host与被屏蔽网站对比，如果一样则直接跳转到error，实现网站屏蔽。    **用户过滤：**例如只允许本机用户访问代理。代码如下：    **网站引导：**即钓鱼。这部分在ProxyThread()中实现。此部分和网站屏蔽有些相似，也是将请求报文中URL与被引导网站比较，如果一样的话就钓鱼（重定向）到那个引导网站上去。在重定向时我们会将构造的302报文中的地址改为钓鱼的IP，然后将这个改好的302报文send回客户端。网站引导的代码如下： |
| 实验结果： |
| 1. **HTTP 代理服务器实验验证过程以及实验结果** 2. 首先修改系统代理，之后运行程序。      1. **基本功能**：运行程序后打开一个网站，成功，控制台输出如下。        1. **Cache功能**：再次打开jwes网站后可以发现文件夹相应位置有缓存文件。      1. **网站过滤**：我们选择屏蔽 <http://today.hit.edu.cn> ，尝试访问失败，证明网站过滤成功。        1. **用户过滤**：修改代码，令本机无访问权限。打开网页无任何响应。        1. **钓鱼**：我们计划当用户访问 <http://jwc.hit.edu.cn> 时跳转至 <http://jwts.hit.edu.cn> 。如下验证。     **2.HTTP 代理服务器源代码（带有详细注释）。**  见报告后。 |
| 问题讨论： |
| 1. 测试缓存时 有时会出现只能接收到200而没有304的情况，这时应该及时主动更换网站；经过测试，换成jwes网站或7k7k小游戏网站、http://info.cern.ch/都是可行的。 2. 一开始我无法使用#pragma comment(lib,"Ws2\_32.lib")，发现是使用了MinGW编译，而MinGW是不支持上书写法的，这一命令是静态链接Ws2\_32.lib库，如果坚持使用MinGW的话就要加上-lws\_32。 3. 注意goto语句后不能再定义新变量。 4. 在做网站屏蔽的功能时尽量将请求转到一个html上（可以自己在github上搞一个github page），不然对于某些浏览器可能会反复请求，跳不出循环。 5. 最后在验收时发现很多功能实现都失败了，例如最后的网站过滤、用户过滤等功能都无法正常实现，重启IDE并且最后在助教的帮助下发现是需要另外打开 chrome 的 Disable cache 才能正常运行。感谢陈学长、付学长和杨学姐 > <！ |
| 心得体会： |
| 本次实验让我对TCP协议传输数据的流程和方式有了更深的体会，对 socket 编程有了初步的了解，通过动手实践有了很大收获；掌握了 HTTP 代理服务器的基本原理，掌握了 HTTP 代理服务器设计与编程实现的基本技能，对 HTTP 请求和响应原理有了更深的认识；同时，也对网站钓鱼、网站屏蔽等有了深刻的理解；感受到了 HTTP cache 的重要作用。 |

附录：程序源代码

#include <stdio.h>

#include <Windows.h>

#include <process.h>

#include <string.h>

#include <tchar.h>

#include <fstream>

#include <map>

#include <string>

/\*

 \* author: 120L020719 zxr

 \*/

#include <iostream>

using namespace std;

#pragma comment(lib, "Ws2\_32.lib")

#define MAXSIZE 65507 \* 5 //发送数据报文的最大长度

#define HTTP\_PORT 80      // http 服务器端口

#define BANNED\_WEB "http://today.hit.edu.cn/"       //屏蔽网站

#define PHISHING\_WEB\_SRC "http://jwc.hit.edu.cn/"   // 钓鱼原网址

#define PHISHING\_WEB\_DEST "http://jwts.hit.edu.cn/" // 钓鱼目的网址

// 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));

    }

};

// 结构体cache

map<string, char \*> cache;

struct HttpCache

{

    char url[1024];

    char host[1024];

    char last\_modified[200];

    char status[4];

    char buffer[MAXSIZE];

    HttpCache()

    {

        ZeroMemory(this, sizeof(HttpCache)); // 初始化cache

    }

};

HttpCache Cache[1024];

int cached\_number = 0; //已经缓存的url数

int last\_cache = 0;    //上一次缓存的索引

BOOL InitSocket();

int ParseHttpHead(char \*buffer, HttpHeader \*httpHeader);

BOOL ConnectToServer(SOCKET \*serverSocket, char \*host);

unsigned int \_\_stdcall ProxyThread(LPVOID lpParameter);

void ParseCache(char \*buffer, char \*status, char \*last\_modified);

//代理相关参数

SOCKET ProxyServer;

sockaddr\_in ProxyServerAddr;

const int ProxyPort = 10240;

// 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

// 可以使用线程池技术提高服务器效率

// const int ProxyThreadMaxNum = 20;

// HANDLE ProxyThreadHandle[ProxyThreadMaxNum] = {0};

// DWORD ProxyThreadDW[ProxyThreadMaxNum] = {0};

struct ProxyParam

{

    SOCKET clientSocket;

    SOCKET serverSocket;

};

int main(int argc, char \*argv[])

{

    printf("代理服务器正在启动\n");

    printf("初始化...\n");

    if (!InitSocket())

    {

        printf("socket 初始化失败\n");

        return -1;

    }

    printf("代理服务器正在运行，监听端口 %d\n", ProxyPort);

    SOCKET acceptSocket = INVALID\_SOCKET;

    SOCKADDR\_IN acceptAddr;

    ProxyParam \*lpProxyParam;

    HANDLE hThread;

    DWORD dwThreadID;

    //代理服务器不断监听

    while (true)

    {

        acceptSocket = accept(ProxyServer, (SOCKADDR \*)&acceptAddr, NULL);

        lpProxyParam = new ProxyParam;

        if (lpProxyParam == NULL)

        {

            continue;

        }

        lpProxyParam->clientSocket = acceptSocket;

        hThread = (HANDLE)\_beginthreadex(NULL, 0,

                                         &ProxyThread, (LPVOID)lpProxyParam, 0, 0);

        CloseHandle(hThread);

        Sleep(200);

    }

    closesocket(ProxyServer);

    WSACleanup();

    return 0;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Method: InitSocket

// FullName: InitSocket

// Access: public

// Returns: BOOL

// Qualifier: 初始化套接字

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

BOOL InitSocket()

{

    //加载套接字库（必须）

    WORD wVersionRequested;

    WSADATA wsaData;

    //套接字加载时错误提示

    int err;

    //版本 2.2

    wVersionRequested = MAKEWORD(2, 2);

    //加载 dll 文件 Socket 库

    err = WSAStartup(wVersionRequested, &wsaData);

    if (err != 0)

    {

        //找不到 winsock.dll

        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); // 创建一个TCP/IP协议族的流套接字

    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;

    ProxyServerAddr.sin\_addr.S\_un.S\_addr = inet\_addr("127.0.0.1"); //只允许本机用户访问服务器

    if (bind(ProxyServer, (SOCKADDR \*)&ProxyServerAddr, sizeof(SOCKADDR)) == SOCKET\_ERROR)

    {

        printf("绑定套接字失败\n");

        return FALSE;

    }

    if (listen(ProxyServer, SOMAXCONN) == SOCKET\_ERROR)

    {

        printf("监听端口%d 失败", ProxyPort);

        return FALSE;

    }

    return TRUE;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Method: ProxyThread

// FullName: ProxyThread

// Access: public

// Returns: unsigned int \_\_stdcall

// Qualifier: 线程执行函数

// Parameter: LPVOID lpParameter

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

unsigned int \_\_stdcall ProxyThread(LPVOID lpParameter)

{

    char Buffer[MAXSIZE];

    char sendBuffer[MAXSIZE];

    char phishBuffer[MAXSIZE];

    char \*CacheBuffer;

    ZeroMemory(Buffer, MAXSIZE);

    ZeroMemory(sendBuffer, MAXSIZE);

    ZeroMemory(phishBuffer, MAXSIZE);

    SOCKADDR\_IN clientAddr;

    int length = sizeof(SOCKADDR\_IN);

    int recvSize;

    int ret;

    int Have\_cache;

    //接收客户端的请求

    recvSize = recv(((ProxyParam \*)lpParameter)->clientSocket, Buffer, MAXSIZE, 0);

    // 为避免报jump to label 'error'的错误，将其注释

    // if (recvSize <= 0)

    // {

    //  goto error;

    // }

    HttpHeader \*httpHeader = new HttpHeader();

    memcpy(sendBuffer, Buffer, recvSize);

    CacheBuffer = new char[recvSize + 1];

    ZeroMemory(CacheBuffer, recvSize + 1);

    memcpy(CacheBuffer, Buffer, recvSize);

    // ParseHttpHead(CacheBuffer, httpHeader);

    Have\_cache = ParseHttpHead(CacheBuffer, httpHeader);

    delete CacheBuffer;

    if (!ConnectToServer(&((ProxyParam \*)lpParameter)->serverSocket, httpHeader->host))

    {

        printf("代理连接主机 %s 失败\n", httpHeader->host);

        goto error;

    }

    printf("代理连接主机 %s 成功\n", httpHeader->host);

    // 网站屏蔽

    if (strcmp(httpHeader->url, BANNED\_WEB) == 0)

    {

        printf("网站 %s 已被屏蔽\n", BANNED\_WEB);

        goto error;

    }

    //网站钓鱼  访问jwc.hit.edu.cn  重定向到jwts.hit.edu.cn

    if (strstr(httpHeader->url, PHISHING\_WEB\_SRC) != NULL)

    {

        char \*pr;

        int phishing\_len;

        // 打印信息

        printf("网站 %s 已被成功重定向至 %s\n", PHISHING\_WEB\_SRC, PHISHING\_WEB\_DEST);

        // 构造报文

        char head1[] = "HTTP/1.1 302 Moved Temporarily\r\n";

        phishing\_len = strlen(head1);

        memcpy(phishBuffer, head1, phishing\_len);

        pr = phishBuffer + phishing\_len;

        char head2[] = "Connection:keep-alive\r\n";

        phishing\_len = strlen(head2);

        memcpy(pr, head2, phishing\_len);

        pr += phishing\_len;

        char head3[] = "Cache-Control:max-age=0\r\n";

        phishing\_len = strlen(head3);

        memcpy(pr, head3, phishing\_len);

        pr += phishing\_len;

        //重定向到jwts.hit.edu.cn

        char phishing\_dest[] = "Location: ";

        strcat(phishing\_dest, PHISHING\_WEB\_DEST);

        strcat(phishing\_dest, "\r\n\r\n");

        phishing\_len = strlen(phishing\_dest);

        memcpy(pr, phishing\_dest, phishing\_len);

        //将302报文返回给客户端

        ret = send(((ProxyParam \*)lpParameter)->clientSocket, phishBuffer, sizeof(phishBuffer), 0);

        goto error;

    }

    //实现cache功能

    if (Have\_cache) //请求的页面在服务器有缓存

    {

        char cached\_buffer[MAXSIZE];

        ZeroMemory(cached\_buffer, MAXSIZE);

        memcpy(cached\_buffer, Buffer, recvSize);

        //构造缓存的报文头

        char \*pr = cached\_buffer + recvSize;

        printf(",,");

        memcpy(pr, "If-modified-since: ", 19);

        pr += 19;

        int length = strlen(Cache[last\_cache].last\_modified);

        memcpy(pr, Cache[last\_cache].last\_modified, length);

        pr += length;

        //将客户端发送的 HTTP 数据报文直接转发给目标服务器

        ret = send(((ProxyParam \*)lpParameter)->serverSocket, cached\_buffer, strlen(cached\_buffer) + 1, 0);

        //等待目标服务器返回数据

        recvSize = recv(((ProxyParam \*)lpParameter)->serverSocket, cached\_buffer, MAXSIZE, 0);

        if (recvSize <= 0)

        {

            goto error;

        }

        //解析包含缓存信息的HTTP报文头

        CacheBuffer = new char[recvSize + 1];

        ZeroMemory(CacheBuffer, recvSize + 1);

        memcpy(CacheBuffer, cached\_buffer, recvSize);

        char last\_status[4];    //记录主机返回的状态字

        char last\_modified[30]; //记录返回页面的修改时间

        ParseCache(CacheBuffer, last\_status, last\_modified);

        delete CacheBuffer;

        //分析cache的状态字

        if (strcmp(last\_status, "304") == 0) // 304状态码，文件没有被修改

        {

            printf("页面未被修改,缓存URL:%s\n", Cache[last\_cache].url);

            //直接将缓存数据转发给客户端

            ret = send(((ProxyParam \*)lpParameter)->clientSocket, Cache[last\_cache].buffer, sizeof(Cache[last\_cache].buffer), 0);

            if (ret != SOCKET\_ERROR)

                printf("由缓存发送\n");

        }

        else if (strcmp(last\_status, "200") == 0) // 200状态码，表示文件已被修改

        {

            //首先修改缓存内容

            printf("页面被修改,缓存URL:%s\n", Cache[last\_cache].url);

            memcpy(Cache[last\_cache].buffer, cached\_buffer, strlen(cached\_buffer));

            memcpy(Cache[last\_cache].last\_modified, last\_modified, strlen(last\_modified));

            //将目标服务器返回的数据直接转发给客户端

            ret = send(((ProxyParam \*)lpParameter)->clientSocket, cached\_buffer, sizeof(cached\_buffer), 0);

            if (ret != SOCKET\_ERROR)

                printf("由缓存发送，已修改\n");

        }

    }

    else //没有缓存过该页面

    {

        //将客户端发送的 HTTP 数据报文直接转发给目标服务器

        ret = send(((ProxyParam \*)lpParameter)->serverSocket, Buffer, strlen(Buffer) + 1, 0);

        //等待目标服务器返回数据

        recvSize = recv(((ProxyParam \*)lpParameter)->serverSocket, Buffer, MAXSIZE, 0);

        if (recvSize <= 0)

        {

            goto error;

        }

        //将该页面缓存到cache中

        //将目标服务器返回的数据直接转发给客户端

        ret = send(((ProxyParam \*)lpParameter)->clientSocket, Buffer, sizeof(Buffer), 0);

        if (ret != SOCKET\_ERROR)

        {

            printf("来自服务器\n成功发送给客户端的报文(目标服务器返回的)buffer ret = %d \n", ret);

        }

    }

//错误处理

error:

    printf("关闭套接字\n\n");

    Sleep(200);

    closesocket(((ProxyParam \*)lpParameter)->clientSocket);

    closesocket(((ProxyParam \*)lpParameter)->serverSocket);

    delete lpParameter;

    \_endthreadex(0);

    return 0;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Method: ParseCache

// FullName: ParseCache

// Access: public

// Returns: void

// Qualifier: 解析 TCP 报文中的 HTTP 头部,在已经cache命中的时候使用

// Parameter: char \*buffer

// Parameter: char \* status

// Parameter: HttpHeader \*httpHeader

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void ParseCache(char \*buffer, char \*status, char \*last\_modified)

{

    char \*p;

    char \*ptr;

    const char \*delim = "\r\n";

    p = strtok\_s(buffer, delim, &ptr); //提取第一行

    memcpy(status, &p[9], 3);

    status[3] = '\0';

    p = strtok\_s(NULL, delim, &ptr);

    while (p)

    {

        if (strstr(p, "Last-Modified") != NULL)

        {

            memcpy(last\_modified, &p[15], strlen(p) - 15);

            break;

        }

        p = strtok\_s(NULL, delim, &ptr);

    }

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Method: ParseHttpHead

// FullName: ParseHttpHead

// Access: public

// Returns: int

// Qualifier: 解析 TCP 报文中的 HTTP 头部

// Parameter: char \*buffer

// Parameter: HttpHeader \*httpHeader

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

int ParseHttpHead(char \*buffer, HttpHeader \*httpHeader)

{

    int flag = 0; //用于表示Cache是否命中，命中为1，不命中为0

    char \*p;

    char \*ptr;

    const char \*delim = "\r\n"; //回车换行符

    p = strtok\_s(buffer, delim, &ptr);

    if (p[0] == 'G')

    { // GET方式

        memcpy(httpHeader->method, "GET", 3);

        memcpy(httpHeader->url, &p[4], strlen(p) - 13);

        printf("url：%s\n", httpHeader->url); // url

        for (int i = 0; i < 1024; i++)

        { //搜索cache，看当前访问的url是否已经存在cache中了

            if (strcmp(Cache[i].url, httpHeader->url) == 0)

            { //说明url在cache中已经存在

                flag = 1;

                break;

            }

        }

        if (!flag && cached\_number != 1023) //说明url没有在cache且cache没有满, 把这个url直接存进去

        {

            memcpy(Cache[cached\_number].url, &p[4], strlen(p) - 13);

            last\_cache = cached\_number;

        }

        else if (!flag && cached\_number == 1023) //说明url没有在cache且cache满了,把第一个cache覆盖

        {

            memcpy(Cache[0].url, &p[4], strlen(p) - 13);

            last\_cache = 0;

        }

    }

    else if (p[0] == 'P') // POST方式

    {

        memcpy(httpHeader->method, "POST", 4);

        memcpy(httpHeader->url, &p[5], strlen(p) - 14);

        for (int i = 0; i < 1024; i++)

        {

            if (strcmp(Cache[i].url, httpHeader->url) == 0)

            {

                flag = 1;

                break;

            }

        }

        if (!flag && cached\_number != 1023)

        {

            memcpy(Cache[cached\_number].url, &p[5], strlen(p) - 14);

            last\_cache = cached\_number;

        }

        else if (!flag && cached\_number == 1023)

        {

            memcpy(Cache[0].url, &p[4], strlen(p) - 13);

            last\_cache = 0;

        }

    }

    p = strtok\_s(NULL, delim, &ptr);

    while (p)

    {

        switch (p[0])

        {

        case 'H': // HOST

            memcpy(httpHeader->host, &p[6], strlen(p) - 6);

            if (!flag && cached\_number != 1023)

            {

                memcpy(Cache[last\_cache].host, &p[6], strlen(p) - 6);

                cached\_number++;

            }

            else if (!flag && cached\_number == 1023)

            {

                memcpy(Cache[last\_cache].host, &p[6], strlen(p) - 6);

            }

            break;

        case 'C': // Cookie

            if (strlen(p) > 8)

            {

                char header[8];

                ZeroMemory(header, sizeof(header));

                memcpy(header, p, 6);

                if (!strcmp(header, "Cookie"))

                {

                    memcpy(httpHeader->cookie, &p[8], strlen(p) - 8);

                }

            }

            break;

            // case '':

        default:

            break;

        }

        p = strtok\_s(NULL, delim, &ptr);

    }

    return flag;

}

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Method: ConnectToServer

// FullName: ConnectToServer

// Access: public

// Returns: BOOL

// Qualifier: 根据主机创建目标服务器套接字，并连接

// Parameter: SOCKET \* serverSocket

// Parameter: char \* host

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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));

    \*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;

}