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	孙月晴		院系	计算机科学与技术	
班级	1603104		学号	1160300901	
任课教师	刘亚维		指导教师	刘亚维	
实验地点	格物 213		实验时间	2018 年 10 月 24	
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：

(1) 熟悉并掌握 Socket 网络编程的过程与技术；

(2) 深入理解 HTTP 协议，掌握 HTTP 代理服务器的基本工作原理；

(3) 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。（选作内容）

(3) 扩展 HTTP 代理服务器，支持如下功能：（选作内容）

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）。

实验原理

(1). Socket编程的客户端和服务器端主要步骤

客户端:

① 确定服务器 IP 地址和端口号

② 创建套接字,通过 connect 连接到服务器所在主机的特定端口

③ 向服务器发出连接请求

④ 和服务器端进行通信

发送数据：

socket 类中的 send（string[,flag]）方法将 string 中的数据发送到连接的套接字。返回值是要发送的字节数量；成功返回 None，失败则抛出异常。

接收数据：

socket 类中的 recv（bufsize[,flag]）方法，数据以字符串形式返回。

⑤ 关闭套接字

socket 类中的 close()方法。

服务器端:

① 创建套接字

② 绑定套接字到相应的 IP 地址和一个端口上

客户端不需要这一步因为操作系统会完成，使用 socket 类中的 bind(address)

③ 将套接字设置为监听模式等待连接请求
④ 请求到来后，接受连接请求，得到新的对应于此次连接的套接字
⑤ 用返回的套接字和客户端进行通信
⑥ 关闭套接字

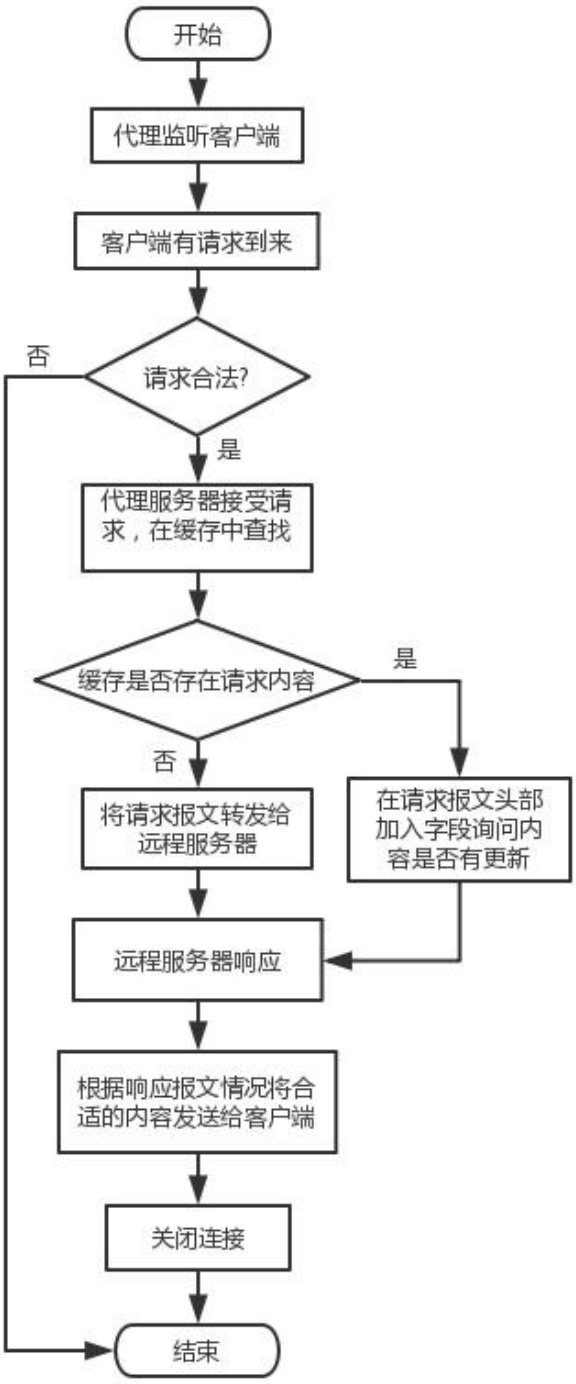
(2). HTTP 代理服务器的基本原理

如果不使用代理服务器去访问 HTTP 协议的网站，浏览器会向服务器发送一个请求，服务器会产生响应，客户端从响应中得到相应的数据。而如果使用代理服务器，则当访问 HTTP 协议的网站时，浏览器将向代理服务器发送请求，代理服务器处理请求，向服务器发送请求，当代理服务器收到服务器 的响应后，将响应报文发送给客户端。

加了代理服务器之后，相对于客户端来说，代理服务器就是服务器；相对于真正的服务器来说，代理服务器就是客户端。

除此之外，还可以利用代理服务器设置缓存，即再报文解析后代理服务器在本地查找缓存，如果有缓存并且没有过期的情况下，直接使用缓存的内容，这样能真正的减少服务器端的负荷，提高客户端的访问速度。

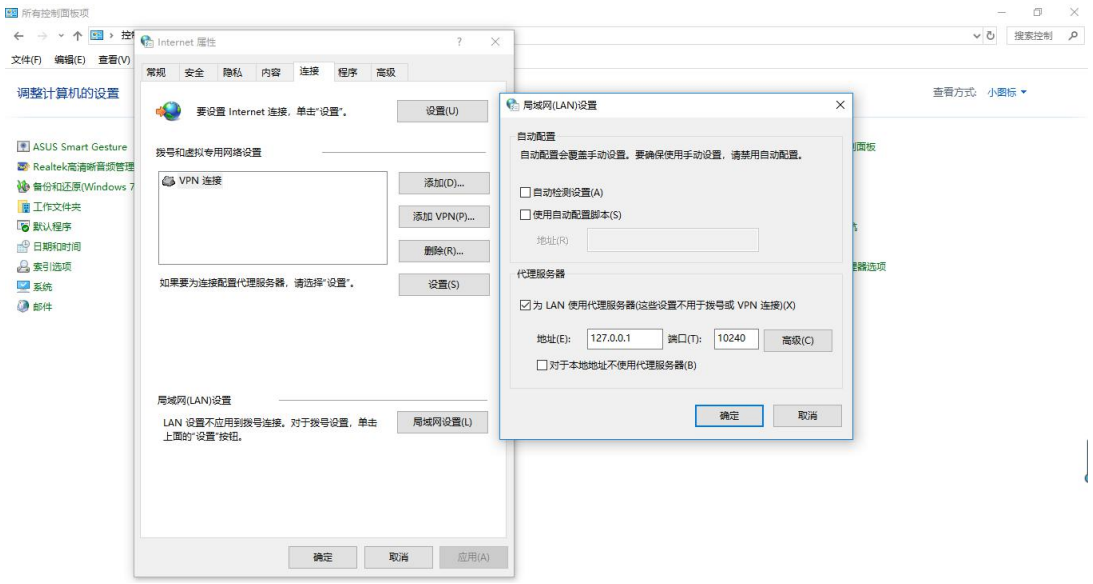
(3). HTTP 代理服务器的程序流程图


开始
代理监听客户端
客户端有请求到来
请求合法?
否
是
代理服务器接受请求，在缓存中查找
缓存是否存在请求内容
是
否
将请求报文转发给远程服务器
在请求报文头部加入字段询问内容是否有更新
远程服务器响应
根据响应报文情况将合适的内容发送给客户端
关闭连接
结束

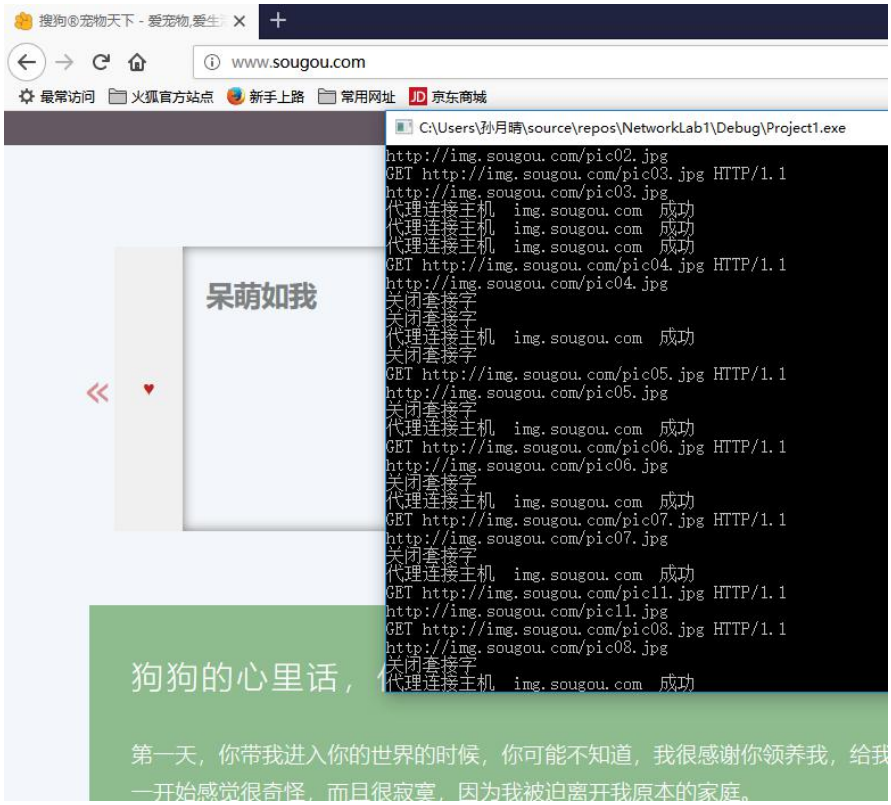
实验过程：

(1). 浏览器使用代理

代理服务器地址:127.0.0.1，端口10240



(2). 验证http代理服务器基本功能：在IE浏览器输入www.sougou.com，在程序运行窗口发现了请求报文，并在浏览器端接收到了网页的数据，说明代理服务器基本功能实现。

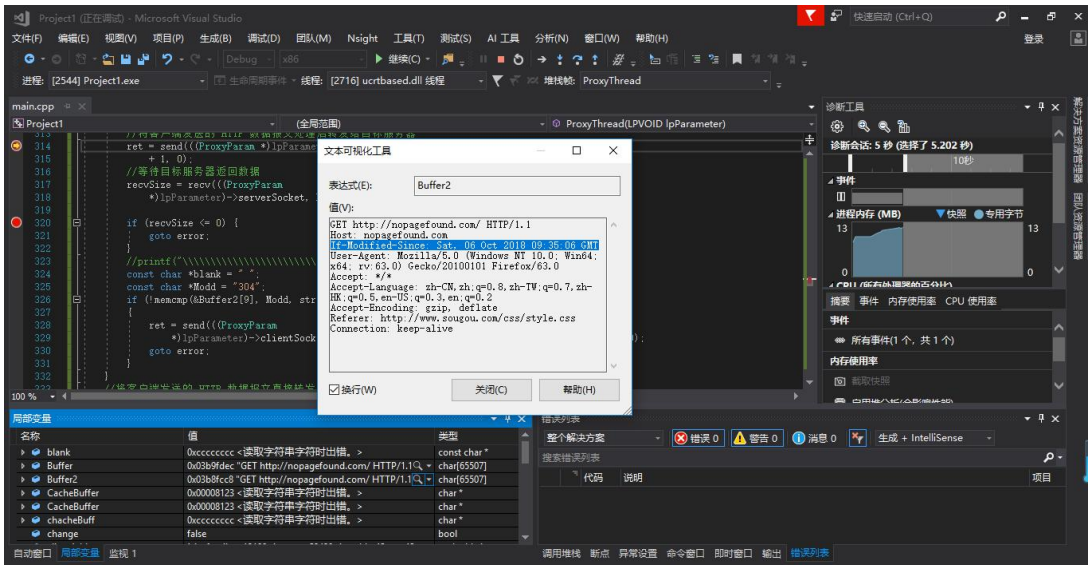


(3). 验证HTTP代理服务器的Cache 功能:在加入缓冲区的代码中设置一个断

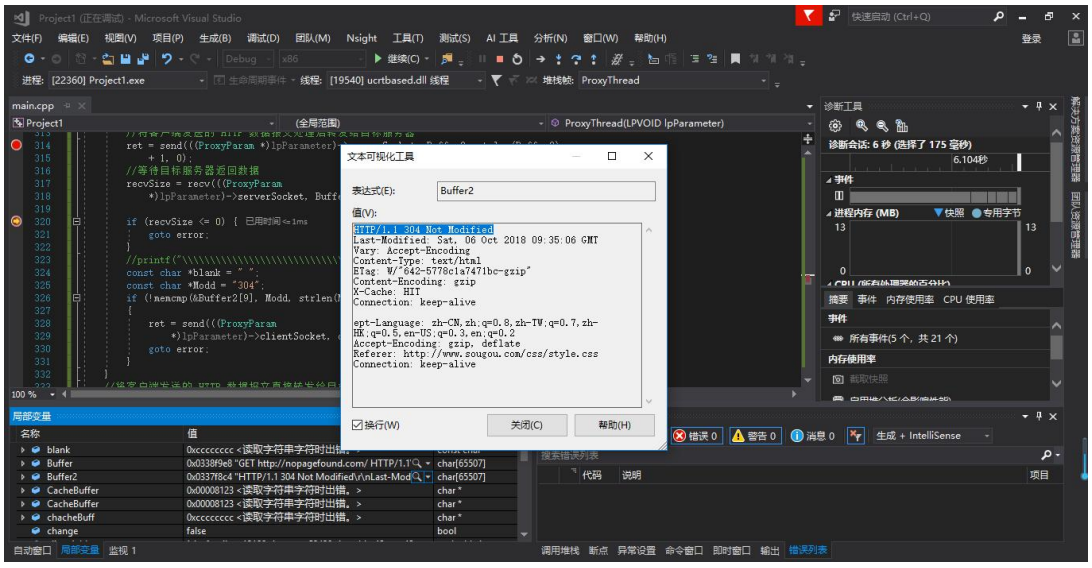
点，如图：

```
313         //将客户端发送的 HTTP 数据报文处理后转发给目标服务器
314         ret = send(((ProxyParam *)lpParameter)->serverSocket, Buffer2, strlen(Buffer2)
315             + 1, 0);
316         //等待目标服务器返回数据
317         recvSize = recv(((ProxyParam
318             *)lpParameter)->serverSocket, Buffer2, MAXSIZE, 0);
319
```

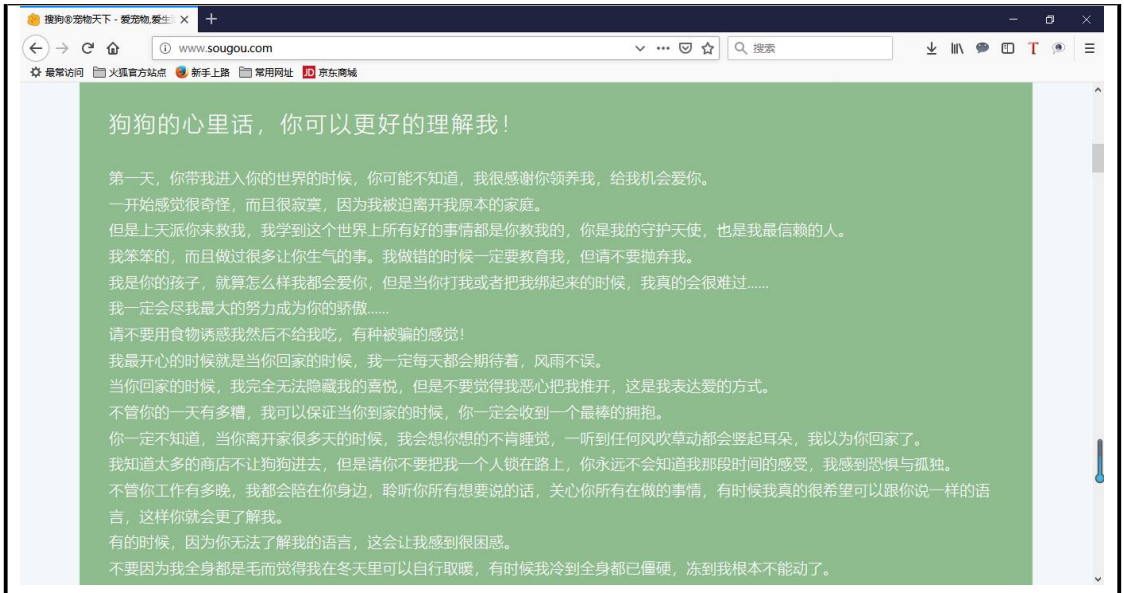
Buffer2 是在缓冲区找到原网页，并且在原请求报文上加上了 If-Modified-Since: 段的请求报文，刷新网页，查看 Buffer2:



F10 键单步运行，查看原服务器返回的响应报文：



发现响应报文中有 HTTP/1.1 304 Not Modified，表示没有更新，直接将缓存中的数据发送给浏览器，此时查看浏览器：

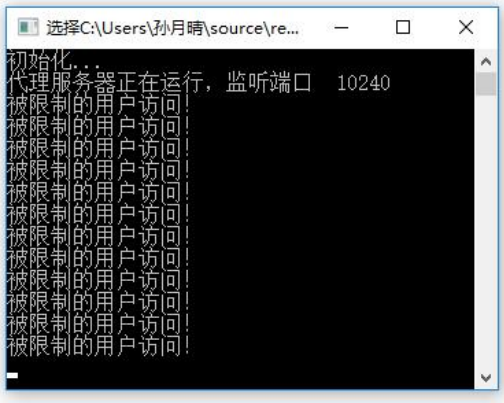


(4). 验证用户过滤功能:首先我限制只能用户来访问-即本机

```
if (strcmp("127.0.0.1", inet_ntoa(verAddr.sin_addr))) {
    printf("被限制的用户访问！\n");
    continue;
}
```

根据前面的实验过程可知能成功访问,现在进行限制本机访问：

```
if (!strcmp("127.0.0.1", inet_ntoa(verAddr.sin_addr))) {
    printf("被限制的用户访问！\n");
    continue;
}
```

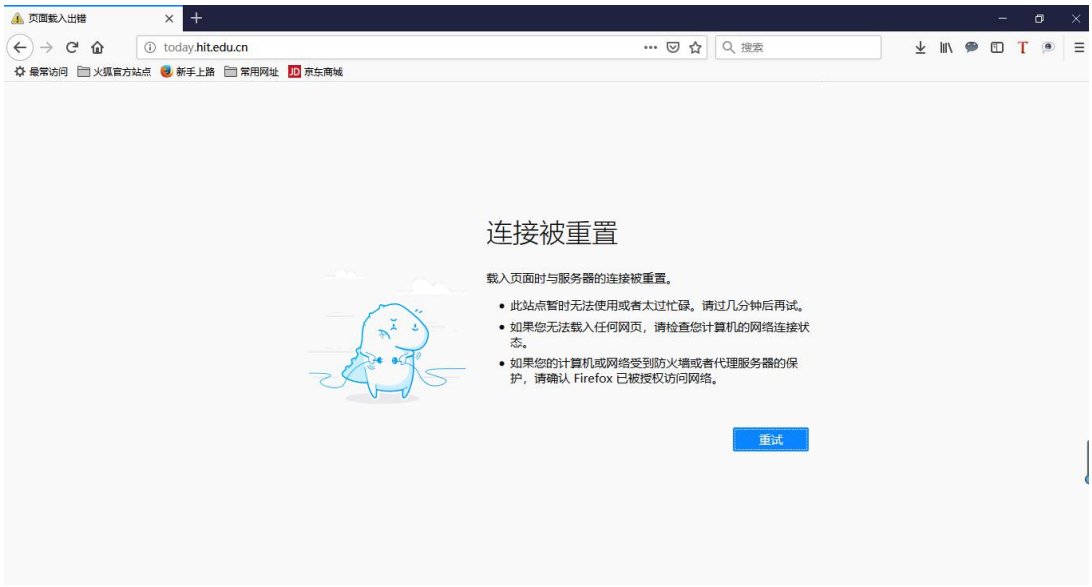


成功限制本机访问

(5). 验证网站过滤扩展功能

以下三个网站主机被禁，接下来我们在浏览器输入被禁网站之一 http://today.hit.edu.cn/，看其反馈：

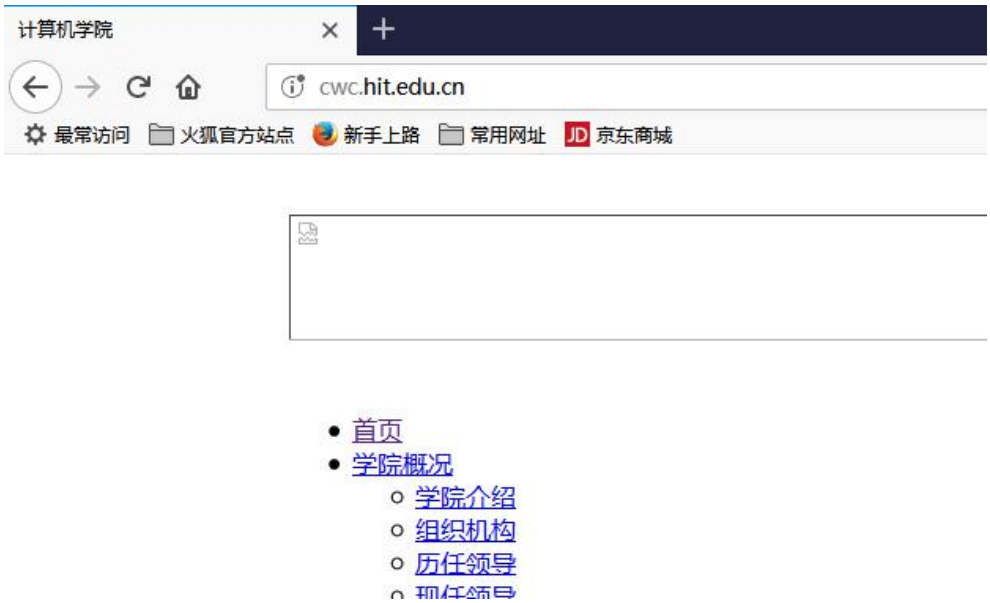
```
//禁止访问的主机，禁用网站
const char* host[10] = { "today.hit.edu.cn", "jwts.hit.edu.cn", "www.hit.edu.cn"
const int host_number = 3;
```



成功禁用

(6). 验证网站钓鱼功能

将网站 http://cwc.hit.edu.cn/诱导到 http://cs.hit.edu.cn/，在浏览器中输入网站 http://cwc.hit.edu.cn/：



可见,想要访问财务处,却被诱导到了计算机学院

实现 HTTP 代理服务器的关键技术及解决方案:

(1). 代理服务器设置 cahce 实现方式

首先设置一个结构体，为存储在 cache 中的数据格式：

```
struct __CACHE{
    cache_HttpHeader htphed;
    char buffer[MAXSIZE];
    char date[DATELENGTH];//存储的更新时间  }};
```

htphed用于在缓冲区中找存储的请求报文的头部，buffer是该请求报文在服务器端返回的响应报文，date指该响应报文最后一次更新的时间，即该响应报文中的last-modifited。

当收到请求报文时，在对报文头部处理之后，首先在cache中寻找该请求，如果找到了，在请求报文之中-第三行加入if-modified-since: date，发送给服务器，接收到服务器返回的响应报文，对响应报文进行处理，看其头部是否为304 not modified，如果是，直接将cache中的响应报文返回给浏览器，如果不是，首先将该响应报文存入cache中，即对cache进行更新—仍存储在之前的那个位置上，然后将响应报文返回给浏览器。如果在cache中没有找到该请求，将处理后的请求报文头部存入Cache，得到响应报文之后，对响应报文进行解析，得到date，然后将响应报文和date存入cache。

如果cache的100个区域满了，会再次从0开始覆盖原来的cache中的数据。

(2). 网站过滤

设置了一个全局数组，里面存放的是被禁止访问的网站的主机。

在ProxyThread函数中解析出请求报文头部之后，将请求报文头部中的host与全局数组中的数据进行比较，如果出现相同的表示访问的网站被禁止访问，直接跳转到结束位置。实现代码如下：

```
for (j = 0; j < host_number; j++)//3表示禁用网站的个数，虽然那里是10，但是实际上只有3个，所以是3
{
    int i = 0;
    bool find = true;
    for (i = 0; i < strlen(host[j]);i++) {
        if (host[j][i] != httpHeader->host[i]) {
            find = false;
            break;
        }
    }
    if (find)
        goto error;
}
```

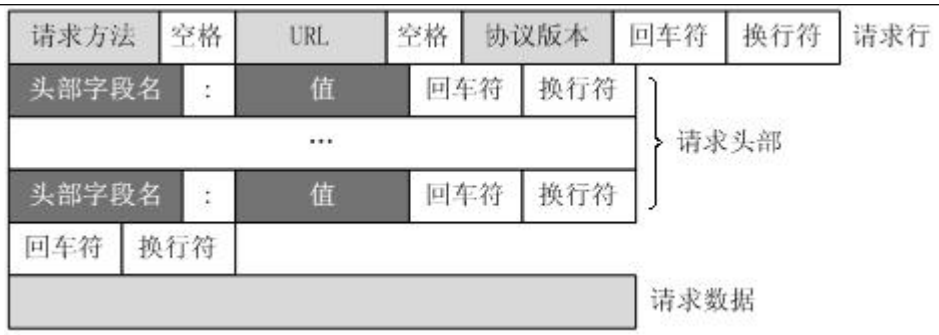
(3). 网站引导

类似于网站过滤，在 ProxyThread 函数中解析出请求报文头部之后，将请求报文头部中的 url 与被引导的网站比较，如果相同，将请求报文中的 url 改为引导向的网站的 url，host 也变为引导向的网站的 host 即可。

问题讨论：

1. 解析 http 报文

一个 HTTP 请求报文由请求行（request line）、请求头部（header）、空行和请求数据 4 个部分组成，如图:



HTTP 响应报文也由三个部分组成，分别是：状态行、消息报头、响应正文。在响应中唯一真正的区别在于第一行中用状态信息代替了请求信息。状态行通过提供一个状态码来说明所请求的资源情况。状态行格式如下：

HTTP-Version Status-Code Reason-Phrase CRLF

2. 代理服务器的正确实现

代理服务器每一步都有正确的顺序，加了代理服务器之后，相对于客户端来说，代理服务器就是服务器；相对于真正的服务器来说，代理服务器就是客户端。因而代理服务器要同时正确的实现客户端套接字编程和服务器端套接字编程，其中的操作顺序不能颠倒。

心得体会：

(1). 此次实验，我在实践过程中很清晰地学到了 TCP 协议在传输数据的流程和方式；熟悉了 Socket 网络编程的过程与技术；同时也更清晰地掌握了 HTTP 代理服务器的基本工作原理；掌握了 HTTP 代理服务器设计与编程实现的基本技能。

(2). 熟悉了通过 C++实现网络编程的简单流程和技巧，对比在 python 中实现网络编程的步骤，可以很明显的发现，C 代码量更大，而 python 实现 socket 编程简单几行代码就能实现 C 中的很长一段代码。

(3). 在设计 Cache 和网站过滤、网站引导和用户过滤的过程中，也了解到了很多 C 语言 scoket 编程中函数的一些用法，一些技巧，受益匪浅。

详细注释源程序:

```
#include <stdio.h>
#include <Windows.h>
#include <process.h>
#include <string.h>
#include <tchar.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 CACHE_MAXSIZE 100 //一个大小为100的数组用以作为CACHE
#define DATELENGTH 40
//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));
    }
};
//这个是高仿的htpp的头部数据，用于在cache中找到对象，但是节约了cookie的空间
struct cache_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
    cache_HttpHeader() {
        ZeroMemory(this, sizeof(cache_HttpHeader));
    }
};
//实现代理服务器的缓存技术
struct __CACHE {
    cache_HttpHeader htphed;//用于在缓冲区中找存储的请求报文的头部
    char buffer[MAXSIZE];//该请求报文在服务器端返回的响应报文
    char date[DATELENGTH];//存储的更新时间
    __CACHE() {
        ZeroMemory(this->buffer, MAXSIZE);
        ZeroMemory(this->buffer, sizeof(date));
    }
```

```
};
int __CACHE_number = 0;//标记下一个应该放缓存的位置
__CACHE cache[CACHE_MAXSIZE];//真`缓存

BOOL InitSocket();
BOOL ParseHttpHead(char *buffer, HttpHeader * httpHeader);
BOOL ConnectToServer(SOCKET *serverSocket, char *host);
unsigned int __stdcall ProxyThread(LPVOID lpParameter);
int Cache_find(__CACHE *cache, HttpHeader htp);

//代理相关参数
SOCKET ProxyServer;
sockaddr_in ProxyServerAddr;
const int ProxyPort = 10240;
//禁止访问的主机，禁用网站
const char* host[10] = { "today.hit.edu.cn", "jwts.hit.edu.cn", "www.hit.edu.cn" };
const int host_number = 3;

//网站诱导
const char* host_to_another = "cwc.hit.edu.cn";//诱导到cs.hit.edu.cn
const char* another[2] = { "cs.hit.edu.cn","http://cs.hit.edu.cn/" };//跳转到的地址

struct ProxyParam {
    SOCKET clientSocket;
    SOCKET serverSocket;
};

int _tmain(int argc, _TCHAR* 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;
    ProxyParam *lpProxyParam;
    HANDLE hThread;
    DWORD dwThreadID;
    //代理服务器不断监听
    sockaddr_in verAddr;
    int hahaha = sizeof(SOCKADDR);
    while (true) {
        acceptSocket = accept(ProxyServer, (SOCKADDR*)&verAddr, &(hahaha));
        lpProxyParam = new ProxyParam;
        if (lpProxyParam == NULL) {
            continue;
        }
        if (strcmp("127.0.0.1", inet_ntoa(verAddr.sin_addr))) {
            printf("被限制的用户访问！\n");
            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 Scoket 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);
    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");
        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 *CacheBuffer;
    ZeroMemory(Buffer, MAXSIZE);
    SOCKADDR_IN clientAddr;
    int length = sizeof(SOCKADDR_IN);
    int recvSize;
    int ret;
    HttpHeader* httpHeader;
    bool change;
    int j;
    int find;
    char *chacheBuff;
    const char *delim;
    char *p;
    bool cun;
```

```
    recvSize = recv(((ProxyParam
        *)lpParameter)->clientSocket, Buffer, MAXSIZE, 0);
    if (recvSize <= 0) {
        goto error;
    }

    httpHeader = new HttpHeader();
    CacheBuffer = new char[recvSize + 1];
    ZeroMemory(CacheBuffer, recvSize + 1);
    memcpy(CacheBuffer, Buffer, recvSize);
    change = ParseHttpHead(CacheBuffer, httpHeader);
    delete CacheBuffer;

    j = 0;
    //对对应的网站不能访问--禁用网站
    for (j = 0; j < host_number; j++)//3表示禁用网站的个数，虽然那里是10，但是实际上只有3个，所以是3
    {
        int i = 0;
        bool find = true;
        for (i = 0; i < strlen(host[j]);i++) {
            if (host[j][i] != httpHeader->host[i]) {
                find = false;
                break;
            }
        }
        if (find)
            goto error;
    }

    if (!ConnectToServer(&((ProxyParam
        *)lpParameter)->serverSocket, httpHeader->host)) {
        goto error;
```

```
}
printf("代理连接主机   %s   成功\n", httpHeader->host);
//跳转到另外一个网站，如果change==tre
if (change) {
    char *CacheBuffer;
    CacheBuffer = new char[MAXSIZE];
    ZeroMemory(CacheBuffer, MAXSIZE);
    //memcpy(CacheBuffer, Buffer, recvSize);
    int ii = 0, lengthth = 0;
    for (ii = 0; ii < strlen(httpHeader->method); ii++) {
        CacheBuffer[lengthth++] = httpHeader->method[ii];
    }
    CacheBuffer[lengthth++] = ' ';
    for (ii = 0; ii < strlen(another[1]); ii++) {
        CacheBuffer[lengthth++] = another[1][ii];
    }
    CacheBuffer[lengthth++] = ' ';
    const char *hh = "HTTP/1.1";
    for (ii = 0; ii < strlen(hh); ii++) {
        CacheBuffer[lengthth++] = hh[ii];
    }
    CacheBuffer[lengthth++] = '\r';
    CacheBuffer[lengthth++] = '\n';
    const char *hhh = "HOST: ";
    for (ii = 0; ii < strlen(hhh); ii++) {
        CacheBuffer[lengthth++] = hhh[ii];
    }
    for (ii = 0; ii < strlen(httpHeader->host); ii++) {
        CacheBuffer[lengthth++] = httpHeader->host[ii];
    }
    CacheBuffer[lengthth++] = '\r';
    CacheBuffer[lengthth++] = '\n';
    char *ptr;
    const char *delim = "\r\n";
    char *p = strtok_s(Buffer, delim, &ptr);
```

```
        int length1 = strlen(p);
        length1 += 2;
        p = strtok_s(NULL, delim, &ptr);
        length1 += strlen(p);
        length1 += 2;
        for (ii = 1; ii < recvSize - length1 + 1; ii++) {
            CacheBuffer[lengthth++] = ptr[ii];
        }
        memcpy(Buffer, CacheBuffer, max(strlen(CacheBuffer), recvSize));
        delete CacheBuffer;
    }

    //在缓冲区找到了该对象
    find = Cache_find(cache, *httpHeader);
    if (find >= 0) {
        char *CacheBuffer;
        char Buffer2[MAXSIZE];
        int i = 0, length = 0, length2 = 0;
        ZeroMemory(Buffer2, MAXSIZE);
        CacheBuffer = new char[recvSize + 1];
        ZeroMemory(CacheBuffer, recvSize + 1);
        memcpy(CacheBuffer, Buffer, recvSize);

        const char *delim = "\r\n";
        char *ptr;
        char *p = strtok_s(CacheBuffer, delim, &ptr);
        length += strlen(p);
        length += 2;
        p = strtok_s(NULL, delim, &ptr);
        length += strlen(p);
        length += 2;
        length2 = length;

        const char *ife = "If-Modified-Since: ";
        for (i = 0; i < length; i++) {
```

```
        Buffer2[i] = Buffer[i];
    }
    for (i = 0; i < strlen(ife); i++)
    {
        Buffer2[length + i] = ife[i];
    }
    length = length + strlen(ife);
    for (i = 0; i < strlen(cache[find].date); i++)
    {
        Buffer2[length + i] = cache[find].date[i];
    }
    length += strlen(cache[find].date);
    Buffer2[length++] = '\r';
    Buffer2[length++] = '\n';
    for (i = length2; i < recvSize;i++) {
        Buffer2[length++] = Buffer[i];
    }
    delete CacheBuffer;


    //将客户端发送的 HTTP 数据报文处理后转发给目标服务器
    ret = send(((ProxyParam *)lpParameter)->serverSocket, Buffer2, strlen(Buffer2)
        + 1, 0);
    //等待目标服务器返回数据
    recvSize = recv(((ProxyParam
        *)lpParameter)->serverSocket, Buffer2, MAXSIZE, 0);

    if (recvSize <= 0) {
        goto error;
    }

    const char *blank = " ";
    const char *Modd = "304";
    if (!memcmp(&Buffer2[9], Modd, strlen(Modd)))
    {
```

```
                ret = send(((ProxyParam
                    *)lpParameter)->clientSocket, cache[find].buffer, strlen(cache[find].buffer)
+ 1, 0);
                goto error;
            }
        }
        //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;
        }

        //得到时间
        chacheBuff = new char[MAXSIZE];
        ZeroMemory(chacheBuff, MAXSIZE);
        memcpy(chacheBuff, Buffer, MAXSIZE);
        delim = "\r\n";
        char *ptr;
        char dada[DATELENGTH];
        ZeroMemory(dada, sizeof(dada));
        p = strtok_s(chacheBuff, delim, &ptr);
        cun = false;
        while (p) {
            if (p[0] == 'L') {
                if (strlen(p) > 15) {
                    char header[15];
                    ZeroMemory(header, sizeof(header));
                    memcpy(header, p, 14);
                    if (!(strcmp(header, "Last-Modified:")))
                    {
                        memcpy(dada, &p[15], strlen(p) - 15);
```

```
                    cun = true;
                    break;
                }
            }
        }
        p = strtok_s(NULL, delim, &ptr);
    }
    if (cun) {
        //把新的东东放入到缓存中去
        if (find >= 0)
        {
            memcpy(&(cache[find].buffer), Buffer, strlen(Buffer));
            memcpy(&(cache[find].date), dada, strlen(dada));

            //printf("----------------%s\n",dada);
        }
        else
        {
            memcpy(&(cache[__CACHE_number%CACHE_MAXSIZE].htphed.host),
httpHeader->host, strlen(httpHeader->host));
            memcpy(&(cache[__CACHE_number%CACHE_MAXSIZE].htphed.method),
httpHeader->method, strlen(httpHeader->method));
            memcpy(&(cache[__CACHE_number%CACHE_MAXSIZE].htphed.url),
httpHeader->url, strlen(httpHeader->url));
            memcpy(&(cache[__CACHE_number%CACHE_MAXSIZE].buffer),    Buffer,
strlen(Buffer));
            memcpy(&(cache[__CACHE_number%CACHE_MAXSIZE].date),    dada,
strlen(dada));
            __CACHE_number++;
        }
    }
    //printf("收到的响应报文: %s \n", Buffer);

    //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);
    //错误处理
error:
    printf("关闭套接字\n");
    Sleep(200);
    closesocket(((ProxyParam*)lpParameter)->clientSocket);
    closesocket(((ProxyParam*)lpParameter)->serverSocket);
    delete      lpParameter;
    _endthreadex(0);
    return 0;
}

//*************************************
// Method:              ParseHttpHead
// FullName:       ParseHttpHead
// Access:             public
// Returns:          void
// Qualifier:   解析 TCP 报文中的 HTTP 头部
// Parameter: char * buffer
// Parameter: HttpHeader * httpHeader
//*************************************
BOOL ParseHttpHead(char *buffer, HttpHeader * httpHeader) {
    char *p;
    char *ptr;
    bool change = false;
    const char * delim = "\r\n";
    /*
    strtok()用来将字符串分割成一个个片段。
    参数s指向欲分割的字符串，参数delim则为分割字符串中包含的所有字符。
    当strtok()在参数s的字符串中发现参数delim中包涵的分割字符时,则会将该字符改为\0
字符。
    在第一次调用时，strtok()必需给予参数s字符串，往后的调用则将参数s设置成NULL。
    每次调用成功则返回指向被分割出片段的指针。
    strtok函数会破坏被分解字符串的完整，调用前和调用后的s已经不一样了。
    */
```

```
p = strtok_s(buffer, delim, &ptr);//提取第一行
printf("%s\n", p);
if (p[0] == 'G') {//GET 方式
	memcpy(httpHeader->method, "GET", 3);

	memcpy(httpHeader->url, &p[4], strlen(p) - 13);
}
else if (p[0] == 'P') {//POST 方式
	memcpy(httpHeader->method, "POST", 4);
	memcpy(httpHeader->url, &p[5], strlen(p) - 14);
}
printf("%s\n", httpHeader->url);
p = strtok_s(NULL, delim, &ptr);//提取第二行

while (p) {
	//printf("-------%s\n", p);
	switch (p[0]) {
	case 'H'://Host
		if (!memcmp(&p[6], host_to_another, strlen(p) - 6))
		{
			memcpy(httpHeader->host, another[0], strlen(another[0]));
			//strcpy(httpHeader->url, "");
			memcpy(httpHeader->url, another[1], strlen(another[1]));
			change = true;
		}
		else {
			memcpy(httpHeader->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;
            default:
                break;
            }
            p = strtok_s(NULL, delim, &ptr);
        }
        return change;
}

//*************************************
// 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);//这个函数的传入值是域名或者主机名，例如"www.google.cn"等等。传出值，是一个hostent的结构。如果函数调用失败，将返回NULL。
        if (!hostent) {
                return FALSE;
        }
        /*
        返回hostent结构体类型指针
        struct hostent
        {
        char        *h_name;
```

```
    char        **h_aliases;
    int         h_addrtype;
    int         h_length;
    char        **h_addr_list;
    #define h_addr h_addr_list[0]
    };
```

hostent->h_name

表示的是主机的规范名。例如www.google.com的规范名其实是www.l.google.com。

hostent->h_aliases

表示的是主机的别名.www.google.com就是google他自己的别名。有的时候，有的主机可能有好几个别名，这些，其实都是为了易于用户记忆而为自己的网站多取的名字。

hostent->h_addrtype

表示的是主机ip地址的类型，到底是ipv4(AF_INET)，还是pv6(AF_INET6)

hostent->h_length

表示的是主机ip地址的长度

hostent->h_addr_lisst

表示的是主机的ip地址，注意，这个是以网络字节序存储的。千万不要直接用printf带%s参数来打这个东西，会有问题的哇。所以到真正需要打印出这个IP的话，需要调用inet_ntop()。

const char *inet_ntop(int af, const void *src, char *dst, socklen_t cnt) ：

这个函数，是将类型为af的网络地址结构src，转换成主机序的字符串形式，存放在长度为cnt的字符串中。返回指向dst的一个指针。如果函数调用错误，返回值是NULL。

```
    */
    in_addr Inaddr = *((in_addr*)*hostent->h_addr_list);//主机的ip地址
    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;
}
//判断两个报文是否相同--不包括cookie
BOOL Isequal(cache_HttpHeader htp1, HttpHeader htp2)
{
    if (strcmp(htp1.method, htp2.method)) return false;
    if (strcmp(htp1.url, htp2.url)) return false;
    if (strcmp(htp1.host, htp2.host)) return false;
    return true;
}

//在缓存中找到对应的对象
int Cache_find(__CACHE *cache, HttpHeader htp)
{
    int i = 0;
    for (i = 0; i < CACHE_MAXSIZE; i++)
    {
        if (Isequal(cache[i].htphed, htp)) return i;
    }
    return -1;
}
```