

**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 | 梅智敏 | | 院系 | 计算学部软件工程 | | |
| 班级 | 1837101 | | 学号 | 1183710118 | | |
| 任课教师 | 李全龙 | | 指导教师 | 李全龙 | | |
| 实验地点 | 格物207 | | 实验时间 | 2020.10.31 | | |
| 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 | | | | | | |
|  | | | | | | |

**计算学部**

|  |
| --- |
| 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。（选作内容，加分项目，可以当堂完成或课下完成） 3. 扩展 HTTP 代理服务器，支持如下功能：（选作内容，加分项目，可以当堂完成或课下完成） 4. 网站过滤：允许/不允许访问某些网站； 5. 用户过滤：支持/不支持某些用户访问外部网站； 6. 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**  服务器端：   * 新建一个主socket * 调用bind函数，将主socket绑定IP地址和端口号 * 调用listen函数，持续监听主socket，以将接收到的“客户端连接请求”放入队列 * 调用accept函数，从队列获取请求；若无请求便会阻塞，直到接收到请求，然后返回一个Socket用于和客户端通信。（为了能够同时与多个客户端通信，往往使用多线程技术创建一个子线程，在子线程中创建另一个新的Socket负责与客户端的连接。） * 通过“三次握手”建立TCP连接 * 调用IO函数和客户端双向通信 * 通信结束后，关闭accept返回的socket   客户端：   * 创建Socket * 调用connect函数，向服务器发起连接请求 * 当服务器通过accept函数成功接收到请求之后，双方进行“三次握手”完成TCP连接 * 调用IO函数和服务器双向通信 * 通信结束后，关闭Socket   示意图如下：    **（2）HTTP 代理服务器的基本原理及程序流程图**  展示实验指导书上的示意图：  ds  通过上图可以发现，代理服务器其实相当于一个“中介”，以帮助客户端和服务器实现间接的连接，客户端和服务器的所有收发请求都要经过Proxy的中转。  本实验需实现的 HTTP 代理服务器，可以分为两个步骤： （首先设置浏览器开启本地代理，注意设置代理端口与代理服务器监听端口保持一致）   1. 单用户代理服务器   单用户的简单代理服务器可以设计为一个非并发的循环服务器。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，读取客户端的HTTP请求报文，通过请求行中的URL，解析客户期望访问的原服务器IP地址；创建访问原（目标）服务器的TCP套接字，将HTTP请求报文转发给目标服务器，接收目标服务器的响应报文，当收到响应报文之后，将响应报文转发给客户端，最后关闭套接字，等待下一次连接。   1. 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。  流程图如下：  **（3）HTTP 代理服务器的关键技术及解决方案**   * **基本功能**     首先，依次调用socket、bind和listen函数以初始化主套接字。当接收到来自客户端的连接请求时，新建子线程。对请求报文的头部文件进行解析，得到请求报文中的method, url, host和cookie等，用于ConnectToServer函数与目标服务器建立连接。此后，Proxy便可充当客户端和服务器通信的中介。   * **缓存功能**     关键代码如下：     * **屏蔽IP**     关键代码如下：     * **屏蔽网址**     关键代码如下：     * **钓鱼功能**     关键代码如下： |
| 实验结果： |
| * **基本功能展示**   首先设置浏览器的代理服务器的IP地址和端口：    接下来访问哈工大官网（不打开Proxy时）：    当我们打开Proxy时：    可见我们实现了Proxy的基本功能   * **缓存功能展示**   当我们首次访问4333小游戏页面时，Proxy会将页面的内容添加到本地的缓存文件中。    这便是产生的本地缓存文件：    当我们再次访问4399小游戏页面时：    提示本地有缓存文件，且Proxy会从缓存文件中直接获取对象。这便证明了我们的Proxy实现了缓存功能。   * **屏蔽IP展示**     我们将Proxy绑定的IP地址更换，使得本地地址无法访问，现在再尝试去访问4399小游戏官网：    可见此时本机IP地址无法与Proxy取得连接，即实现了屏蔽IP的功能。   * **屏蔽网址展示**     我们尝试访问被屏蔽的qq网址    可见成功实现了屏蔽网址的功能。   * **钓鱼功能展示**     我们尝试访问钓鱼源网址：上海交通大学官网      可见我们的Proxy实现了钓鱼功能。 |
| 问题讨论： |
| 对实验过程中的思考问题进行讨论或回答。   * goto 语句后面，不要再新定义变量。否则，Clion会报错。 * 在CLion中要想使用Socket 相关的API，必须执行下面三步，以确保导入了所需的库。  1. 在CPP源文件中添加 #include <winsock2.h> 2. 在CPP源文件中添加 #pragma comment(lib,"Ws2\_32.lib") 3. 在CMakeLists.txt文件中添加 link\_libraries(ws2\_32) |
| 心得体会： |
| 结合实验过程和结果给出实验的体会和收获。   * 本次实验，让我对socket编程有了初步的了解，进一步理解了基于TCP连接的通信过程，掌握了HTTP代理服务器的基本原理，对HTTP请求和响应原理有了更深的认识；同时，也对钓鱼功能、网站屏蔽等有了更多的兴趣。 * 在实现HTTP缓存的过程中，错综复杂的指针使用让我头晕目眩。但是经过这一番锻炼，我对C++中的指针使用更加地熟练，强大的指针工具确实需要我们通过实践去逐渐掌握。 * 我使用C++编写此次试验（其实是为了借用指导书上的示例代码），虽然语法比python要繁杂不少，但是它在处理HTTP段结构的时候精确到bit，这更有助于我们理解一些细节。   源代码：  myproxy.cpp  #include <winsock2.h> #include <process.h> #include <string.h> #include <iostream> #include "cache.cpp"  using namespace std; #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 **INVALID\_WEBSITE** "http://www.qq.com/" *//被屏蔽的网站* #define **FISHING\_WEB\_SRC** "http://sjtu.edu.cn/" *//钓鱼的源网址* #define **FISHING\_WEB\_DEST** "http://jwts.hit.edu.cn/" *//钓鱼的目的网址* #define **FISHING\_WEB\_HOST** "jwts.hit.edu.cn" *//钓鱼目的地址的主机名  //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));  } };  **BOOL** InitSocket(); void ParseHttpHead(char \*buffer,HttpHeader \* httpHeader); **BOOL** ConnectToServer(SOCKET \*serverSocket,char \*host); unsigned int **\_\_stdcall** ProxyThread(LPVOID lpParameter);  *//proxy上的Socket* SOCKET ProxyServer; *//Socket需要绑定的地址变量* sockaddr\_in ProxyServerAddr; const int ProxyPort = 8080;  *//缓存相关参数* boolean haveCache = **FALSE**; boolean needCache = **TRUE**;  struct ProxyParam{  SOCKET clientSocket;  SOCKET serverSocket; };  int main(int argc, char\* argv[]) {   cout << "Proxy is running "<<endl;  cout << "正在初始化套接字......" << endl;  if(!InitSocket()){  cout << "socket 初始化失败"<< endl;  return -1;  }  cout << "初始化成功，正在监听PORT: " <<ProxyPort <<endl;  SOCKET acceptSocket = **INVALID\_SOCKET**;  ProxyParam \*lpProxyParam;  HANDLE hThread;  *//代理服务器不断监听* while(true){  acceptSocket = accept(ProxyServer,**NULL**,**NULL**);  lpProxyParam = new ProxyParam;  if(lpProxyParam == **NULL**){  continue;  }  lpProxyParam->clientSocket = acceptSocket;  hThread = (HANDLE)\_beginthreadex(**NULL**, 0, &ProxyThread,(LPVOID)lpProxyParam, 0, 0);  CloseHandle(hThread);  Sleep(500);  } }  *//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // 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结构体中主要包含了系统所支持的Winsock版本信息* 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* cout << "load winsock failed, the error ID is : "<<WSAGetLastError() <<endl;  return **FALSE**;  }  *//LOBYTE()得到一个16bit数最低（最右边）那个字节  //HIBYTE()得到一个16bit数最高（最左边）那个字节  //判断打开的是否是2.2版本* if(**LOBYTE**(wsaData.wVersion) != 2 || **HIBYTE**(wsaData.wVersion) !=2) {  cout << "can not find the right winsock version" << endl;  WSACleanup();  return **FALSE**;  }  *//AF\_INET,PF\_INET IPv4 Internet协议  //SOCK\_STREAM Tcp连接，提供序列化的、可靠的、双向连接的字节流。支持带外数据传输* ProxyServer = socket(**AF\_INET**, **SOCK\_STREAM**, 0);  if(**INVALID\_SOCKET** == ProxyServer){  cout << "creat socket is failed , the error ID is: " <<WSAGetLastError() << endl;  return **FALSE**;  }  ProxyServerAddr.sin\_family = **AF\_INET**; *//使用主机+port地址格式* ProxyServerAddr.sin\_port = htons(ProxyPort); *//将整型变量从主机字节顺序转变成网络字节顺序   //屏蔽用户  //只要不是从该地址访问代理服务器的客户端，都会被该代理服务器屏蔽  //ProxyServerAddr.sin\_addr.S\_un.S\_addr = INADDR\_ANY;//任何IP地址均可访问  //ProxyServerAddr.sin\_addr.S\_un.S\_addr = inet\_addr("127.0.0.1");//仅本机IP地址可访问* ProxyServerAddr.sin\_addr.S\_un.S\_addr = inet\_addr("127.0.0.122"); *//本机IP不可访问* if(bind(ProxyServer,(SOCKADDR\*)&ProxyServerAddr,sizeof(SOCKADDR)) == **SOCKET\_ERROR**){  cout << "bind socket is failed " << endl;  return **FALSE**;  }  if(listen(ProxyServer, **SOMAXCONN**) == **SOCKET\_ERROR**){  cout << "listen Port"<< ProxyPort <<" is failed "<< endl;  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**], fileBuffer[**MAXSIZE**];  char \*CacheBuffer;  HttpHeader\* httpHeader = new HttpHeader();  **ZeroMemory**(Buffer,MAXSIZE);  SOCKADDR\_IN clientAddr;  int recvSize;  recvSize = recv(((ProxyParam \*)lpParameter)->clientSocket,Buffer,**MAXSIZE**,0);  CacheBuffer = new char[recvSize + 1];  **ZeroMemory**(CacheBuffer, recvSize + 1);  memcpy(CacheBuffer, Buffer, recvSize);   *//解析http首部* ParseHttpHead(CacheBuffer, httpHeader);   *//缓存* char \*DateBuffer;  DateBuffer = (char\*)malloc(**MAXSIZE**);  **ZeroMemory**(DateBuffer, strlen(Buffer) + 1);  memcpy(DateBuffer, Buffer, strlen(Buffer) + 1);  char filename[100];  **ZeroMemory**(filename, 100);  *// 根据url地址构造txt文件名* makeFilename(httpHeader->url, filename);  char \*field = "Date";  char date\_str[30]; *//保存字段Date的值* **ZeroMemory**(date\_str, 30);  **ZeroMemory**(fileBuffer, MAXSIZE);  FILE \*in;  *// 在本地查询是否存在对应的缓存文件，若有，则解析Date并给Http报文段增加“if-modified-Since”字段* if ((in = fopen(filename, "rb")) != **NULL**) {  cout << "当前访问的页面有本地缓存文件！ " << endl;  fread(fileBuffer, sizeof(char), **MAXSIZE**, in);  fclose(in);  *// 将缓存文件中的Data字段存入date\_str* ParseDate(fileBuffer, field, date\_str);  cout << "该本地缓存文件的版本为： " <<date\_str << endl;  *// 给HTTP报文段增加“if-modified-Since”字段* makeNewHTTP(Buffer, date\_str);  haveCache = **TRUE**;  goto success;  }   *//网站过滤：屏蔽一个网站  //对请求过来的 HTTP 报文头部进行检测，提取出其中的访问地址 url ，检测其是否为要被屏蔽的网址* if (strcmp (httpHeader->url, **INVALID\_WEBSITE**) == 0) {  cout << "-----------------------------------------------"<< endl;  cout << "该网站已经被屏蔽，访问失败！"<< endl;  goto error;  }   *//钓鱼：将访问网址转到其他网站* if (strstr(httpHeader->url, **FISHING\_WEB\_SRC**) != **NULL**) {  cout << "-----------------------------------------------"<< endl;  cout << "钓鱼成功！已从源网址："<<**FISHING\_WEB\_SRC** << " 转到目的网址："<<**FISHING\_WEB\_DEST** << endl;  *// 通过更改 HTTP 头部字段的 url 和 host 来实现钓鱼功能  // 第三个参数表示字符串的长度，故加1* memcpy(httpHeader->host, **FISHING\_WEB\_HOST**, strlen(**FISHING\_WEB\_HOST**) + 1);  memcpy(httpHeader->url, **FISHING\_WEB\_DEST**, strlen(**FISHING\_WEB\_DEST**));  }  delete CacheBuffer;  delete DateBuffer;   success:  if(!ConnectToServer(&((ProxyParam \*)lpParameter)->serverSocket,httpHeader->host)) {  cout << "连接目标服务器失败"<< endl;  goto error;  }  cout << "代理连接服务器成功！"<< endl;  *//将客户端发送的 HTTP 数据报文直接转发给目标服务器* 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){  cout << "返回目标服务器的数据失败! " << endl;  goto error;  }  *//有缓存时，判断返回的状态码是否是304，若是则将缓存的内容发送给客户端* if (haveCache == **TRUE**) {  getCache(Buffer, filename);  }  if (needCache == **TRUE**) {  makeCache(Buffer, httpHeader->url); *//缓存报文* }  *//将目标服务器返回的数据直接转发给客户端* send(((ProxyParam \*) lpParameter)->clientSocket, Buffer, sizeof(Buffer), 0);   *//错误处理* error:  cout << "close socket"<< endl;  Sleep(500);  closesocket(((ProxyParam\*)lpParameter)->clientSocket);  closesocket(((ProxyParam\*)lpParameter)->serverSocket);  delete lpParameter;  \_endthreadex(0); } *//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Method: ParseHttpHead // FullName: ParseHttpHead // Access: public // Returns: void // Qualifier: 解析 TCP 报文中的 HTTP 头部，将url host等信息存入httpHeader // Parameter: char \* buffer // Parameter: HttpHeader \* httpHeader //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\** void ParseHttpHead(char \*buffer,HttpHeader \* httpHeader){  char \*p;  const char \* delim = "\r\n";  p = strtok(buffer,delim); *// 第一次调用，第一个参数为被分解的字符串* if(p[0] == 'G'){  *//GET 方式* memcpy(httpHeader->method,"GET",3);  memcpy(httpHeader->url,&p[4],strlen(p) -13); *//'Get' 和 'HTTP/1.1' 各占 3 和 8 个，再加上俩空格，一共13个* }  else if(p[0] == 'P'){  *//POST 方式* memcpy(httpHeader->method,"POST",4);  memcpy(httpHeader->url,&p[5],strlen(p) - 14); *//'Post' 和 'HTTP/1.1' 各占 4 和 8 个，再加上俩空格，一共14个* }  printf("访问的url是 ： %s\n",httpHeader->url);  *// 第二次调用，需要将第一个参数设为 NULL* p = strtok(**NULL**,delim);  while(p){  switch(p[0]){  case 'H':*//Host* 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(**NULL**,delim);  } } *//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // 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**; }  cache.cpp:  #include <iostream> #include <winsock2.h> #include <string.h> using namespace std; #define **MAXSIZE** 65507 *//发送数据报文的最大长度   // 将本地缓存文件的Data字段存入tempDate* boolean ParseDate(char \*buffer, char \*field, char \*tempDate) {  char \*p, \*ptr, temp[5];  const char \*delim = "\r\n";  **ZeroMemory**(temp, 5);  p = strtok(buffer, delim);  int len = strlen(field) + 2;  while (p) {  if (strstr(p, field) != **NULL**) {  *// 获取日期Date* memcpy(tempDate, &p[len], strlen(p) - len);  return **TRUE**;  }  p = strtok(**NULL**, delim);  }  return **TRUE**; }  *//改造HTTP请求报文，增加“If-Modified-Since”字段* void makeNewHTTP(char \*buffer, char \*value) {  const char \*field = "Host";  const char \*newfield = "If-Modified-Since: ";  *//const char \*delim = "\r\n";* char temp[**MAXSIZE**];  **ZeroMemory**(temp, MAXSIZE);  char \*pos = strstr(buffer, field);  int i = 0;  for (i = 0; i < strlen(pos); i++) {  temp[i] = pos[i];  }  \*pos = '\0';  *//插入If-Modified-Since字段* while (\*newfield != '\0') {  \*pos++ = \*newfield++;  }  while (\*value != '\0') {  \*pos++ = \*value++;  }  \*pos++ = '\r';  \*pos++ = '\n';  for (i = 0; i < strlen(temp); i++) {  \*pos++ = temp[i];  } }  *//根据url构造本地缓存文件名* void makeFilename(char \*url, char \*filename) {  while (\*url != '\0') {  if (\*url != '/' && \*url != ':' && \*url != '.') {  \*filename++ = \*url;  }  url++;  }  *//本地缓存的文件名* strcat(filename, ".txt"); }  *//将内容写入本地缓存文件，以备下一次访问时直接调用* void makeCache(char \*buffer, char \*url) {  char \*p, \*ptr, num[10], tempBuffer[**MAXSIZE** + 1];  const char \* delim = "\r\n";  **ZeroMemory**(num, 10);  **ZeroMemory**(tempBuffer, MAXSIZE + 1);  *// 将buffer内容转入tempBuffer，以准备存入本地文件* memcpy(tempBuffer, buffer, strlen(buffer));  p = strtok(tempBuffer, delim);*//提取第一行* memcpy(num, &p[9], 3);  *//如果缓存过期 or 没有缓存，服务器返回状态码200，因此更新本地缓存* if (strcmp(num, "200") == 0) {  char filename[100] = { 0 };  *//构造文件名* makeFilename(url, filename);  FILE \*out;  *//写入本地缓存文件* out = fopen(filename, "w");  fwrite(buffer, sizeof(char), strlen(buffer), out);  fclose(out);  cout << "-------------------------------------"<< endl;  cout << "该网页缓存成功！" << endl;  } }  *//从本地缓存文件中获取对象* void getCache(char \*buffer, char \*filename) {  char \*p, num[10], tempBuffer[**MAXSIZE** + 1];  const char \* delim = "\r\n";  **ZeroMemory**(num, 10);  **ZeroMemory**(tempBuffer, MAXSIZE + 1);  memcpy(tempBuffer, buffer, strlen(buffer));  p = strtok(tempBuffer, delim);*//提取第一行* memcpy(num, &p[9], 3);  *// 服务器返回状态码304，表示本地有缓存文件且未过期  // 因此proxy直接将本地缓存文件的内容发送给客户端* if (strcmp(num, "304") == 0) {  cout << "-------------------------------------"<< endl;  cout << "已经从本地缓存文件获取对象"<< endl;  **ZeroMemory**(buffer, strlen(buffer));  FILE \*in = **NULL**;  if ((in = fopen(filename, "r")) != **NULL**) {  fread(buffer, sizeof(char), **MAXSIZE**, in);  fclose(in);  }  }  } |