

华中农业大学计算机网络实验报告

课程名称：计算机网络

专业班级：\_计科1403\_

姓 名：\_宋琦敏

学 号：2014317200325

指导老师：\_王建勇\_

报告日期：\_2016年12月31日\_

信息学院计算机科学与技术专业

**网络编程**

**实验一、获取主机名和IP地址**

1. 实验要求

利用winsock函数获取IP地址和计算机名，计算机MAC地址，网卡类型，子网掩码，网关地址等网卡基本信息。可以通过gethostname和gethostbyname两个函数直接获取目的主机的IP地址和计算机名。

1. 实验目的

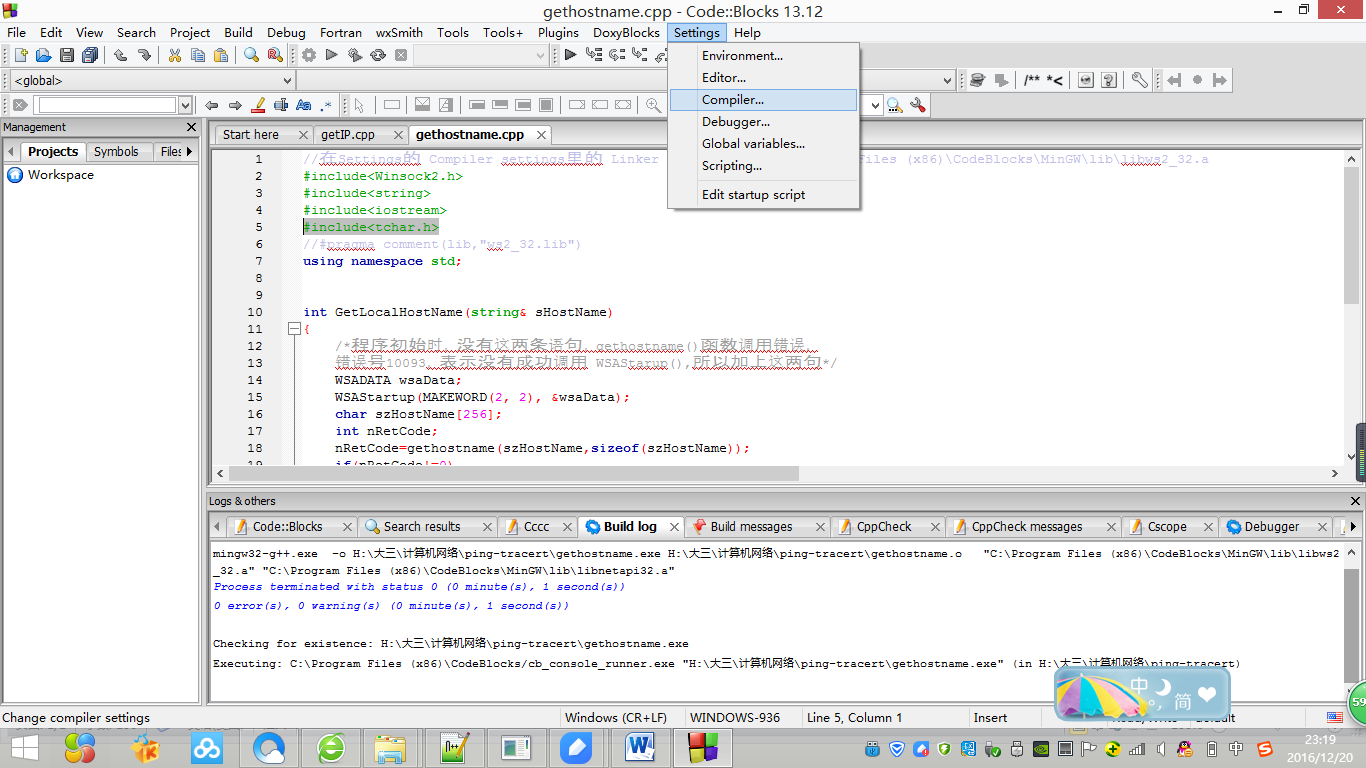
了解Windows系统下获取IP地址，主机名，物理与网络相关信息的方法

1. 实验环境

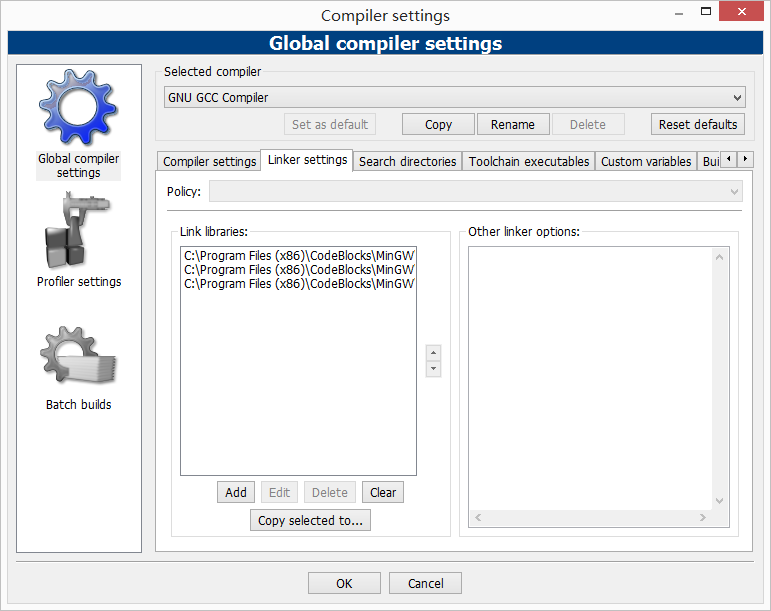
Windows8.1操作系统，codeblocks软件

1. 实验步骤
2. 配置codeblocks与网络编程有关的运行环境

打开对话框Setting->Compiler settings



选择Linker settings



添加相应的.a库文件

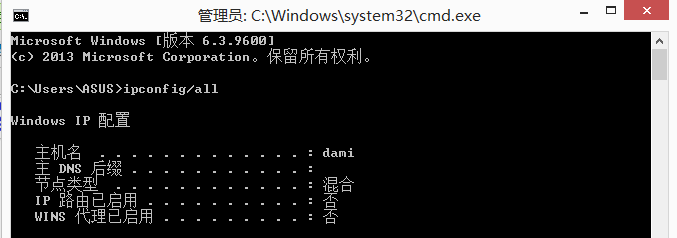
C:\Program Files (x86)\CodeBlocks\MinGW\lib\libws2\_32.a

C:\Program Files (x86)\CodeBlocks\MinGW\lib\libnetapi32.a

C:\Program Files (x86)\CodeBlocks\MinGW\lib\libiphlpapi.a

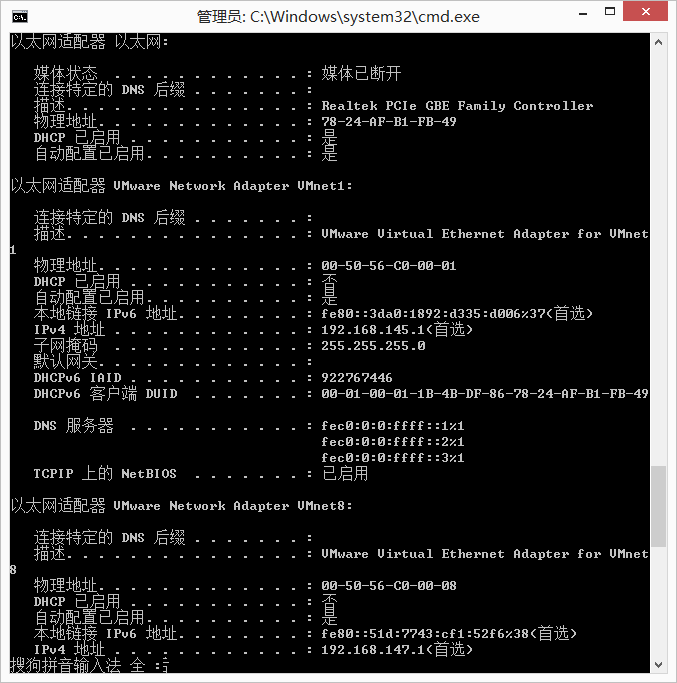
1. 通过cmd查看计算机的主机名和IP地址，以验证后续程序的正确性

主机名：dami



当本计算机连接宽带，显示所有网卡信息如下：





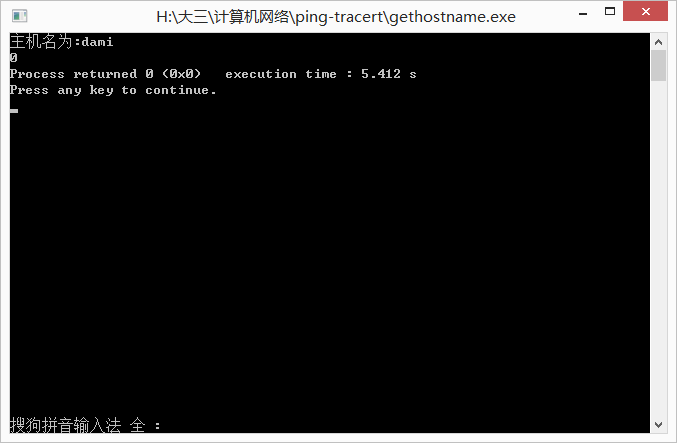


1. 执行获取主机名的程序：

核心代码如下：

|  |
| --- |
| #include<Winsock2.h>  #include<string>  #include<iostream>  #include<tchar.h>  using namespace std;  int GetLocalHostName(string& sHostName)  {  WSADATA wsaData;  WSAStartup(MAKEWORD(2, 2), &wsaData); /\*程序初始时，没有这两条语句，gethostname()函数调用错误，错误号10093，表示没有成功调用 WSAStarup(),所以加上这两句\*/  char szHostName[256];  int nRetCode;  nRetCode=gethostname(szHostName,sizeof(szHostName));  if(nRetCode!=0)  {  sHostName=\_T("Not available"); //\_T("")是一个宏,他的作用是让你的程序支持Unicode编码,在头文件#include<tchar.h>里  return WSAGetLastError();  }  sHostName=szHostName;  cout<<"主机名为:"<<sHostName<<endl;  return 0;  }  int main()  {  string s;  int n;  n=GetLocalHostName(s);  cout<<n; //检查gethostame()是否调用成功，正确：返回0，并显示主机名，错误：则返回错误号  ::WSACleanup();  } |

运行结果显示如下：



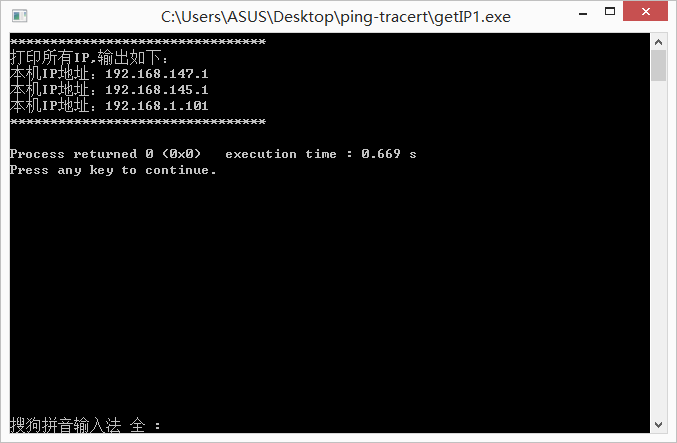
通过结果显示，可知程序运行正确，返回主机名，并输出0表示验证getlocalhostname（）调用正确

1. 执行获取IP地址程序

代码如下：

|  |
| --- |
| #include"stdio.h"  #include"string.h"  #include"Winsock2.h"  int main()  {  WSADATA wsaData;  if(WSAStartup(MAKEWORD(2,2),&wsaData)==SOCKET\_ERROR)  {  exit(0);  }  int nLen=256;  char hostname[20];  gethostname(hostname,nLen);  printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  printf("打印所有IP,输出如下：\n");  char szHost[256] = {0};  hostent \*lpHost = gethostbyname(szHost); //返回对于给定主机名的主机信息，即从返回的数据结构 struct hostent 中获取信息。返回非空指针则为正确调用    memset(szHost,0,sizeof(szHost)); //内存初始化，将szHost当前位置后面的sizeof(szHost)个字节用0代替  // 打印出所有IP地址  in\_addr addr;  for(int i = 0; ; i++)  {  char \*p = lpHost->h\_addr\_list[i]; //因为主机中可能存在多个网卡，所以通过数组获取所有IP地址  if(p == NULL)  break;  memcpy(&addr.S\_un.S\_addr, p, lpHost->h\_length); //从p所指的内存地址开始拷贝lpHost->h\_length个字节到&addr.S\_un.S\_addr所指的内存区域  char \*szIp = inet\_ntoa(addr); //将32位的二进制数转化为字符串;  printf("本机IP地址：%s \n", szIp);  }  printf("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");  ::WSACleanup(); //与WSAStartup(MAKEWORD(2,2),&wsaData)配对执行  } |

运行结果显示如下：



程序运行结果与之前终端命令行查看的IP地址相符，可知该程序能正确获取本机的IP地址。在本机中因安装有虚拟机，所以有两个虚拟网卡设置的IP地址如下

以太网适配器VMware Network Adapter VMnet1：192.168.145.1

以太网适配器VMware Network Adapter VMnet8：192.168.147.1

因为本计算机通过无线网接入，所以IP地址如下

无线局域网适配器 WLAN：10.164.12.1

从实验可以得知，适配器用于使PC机连接到局域网，IP地址存储在计算机的存储器中，所以可以按主机名获取计算机存储器中的所有IP地址

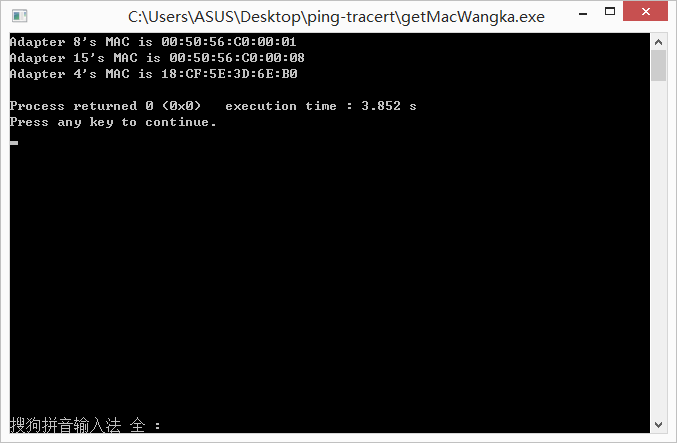
计算机的硬件地址即MAC地址，存储在适配器的ROM中，所以一个计算机有多个适配器，就会有多个MAC地址。

1. 获取计算机适配器中的MAC地址：

代码如下：

|  |
| --- |
| #include<Winsock2.h>  #include<string>  #include<iostream>  #include<tchar.h>  #include<Nb30.h>  #include <windows.h>  #include <stdlib.h>  #include<stdio.h>  using namespace std;  struct ASTAT  {  ADAPTER\_STATUS adapt;  NAME\_BUFFER NameBuff[30];  } Adapter;  UCHAR GetAddressByIndex(int lana\_num, ASTAT &Adapter)  {  NCB ncb;//网络控制块  UCHAR uRetCode;  memset(&ncb, 0, sizeof(ncb));  //指定网卡号，首先对选定的网卡发送一个NCBRESET命令，以便进行初始化  ncb.ncb\_command = NCBRESET;  ncb.ncb\_lana\_num = lana\_num;  uRetCode=Netbios(&ncb);//取得该网络控制块的指针  memset(&ncb, 0, sizeof(ncb));  ncb.ncb\_command = NCBASTAT;  ncb.ncb\_lana\_num = lana\_num;//指定网卡号  strcpy((char\*)ncb.ncb\_callname,"\* ");  ncb.ncb\_buffer=(unsigned char \*)&Adapter;  //指定返回的信息存放的变量  ncb.ncb\_length=sizeof(Adapter);  //发送NCBASTAT命令以获取网卡的信息  uRetCode=Netbios(&ncb);  return uRetCode;  }  int main()  {  NCB ncb;  UCHAR uRetCode;  int num=0;  LANA\_ENUM lana\_enum;  memset(&ncb,0,sizeof(ncb));  ncb.ncb\_command=NCBENUM;  ncb.ncb\_buffer=(unsigned char \*)&lana\_enum;  ncb.ncb\_length=sizeof(lana\_enum);  //向网卡发送NCBENUM命令，以获取当前机器的网卡信息，包括网卡的个数以及编号  uRetCode=Netbios(&ncb);  char acMAC[18];  if(uRetCode==0)  {  num=lana\_enum.length;  for(int i=0;i<num;i++)  {  ASTAT Adapter;  if(GetAddressByIndex(lana\_enum.lana[i],Adapter)==0)  {  sprintf(acMAC, "%02X:%02X:%02X:%02X:%02X:%02X",int (Adapter.adapt.adapter\_address[0]),  int (Adapter.adapt.adapter\_address[1]),  int (Adapter.adapt.adapter\_address[2]),  int (Adapter.adapt.adapter\_address[3]),  int (Adapter.adapt.adapter\_address[4]),  int (Adapter.adapt.adapter\_address[5]));//以指定的格式将MAC地址写进acMAC中  cout << "Adapter " << int (lana\_enum.lana[i]) <<//获取网卡号，和MAC地址  "'s MAC is " << acMAC << endl;  }  }  }  } |

运行结果显示如下：



从运行结果可发现，正确获取并输出本计算机的三个网卡的MAC地址。

1. 获取网卡类型，子网掩码，网关地址

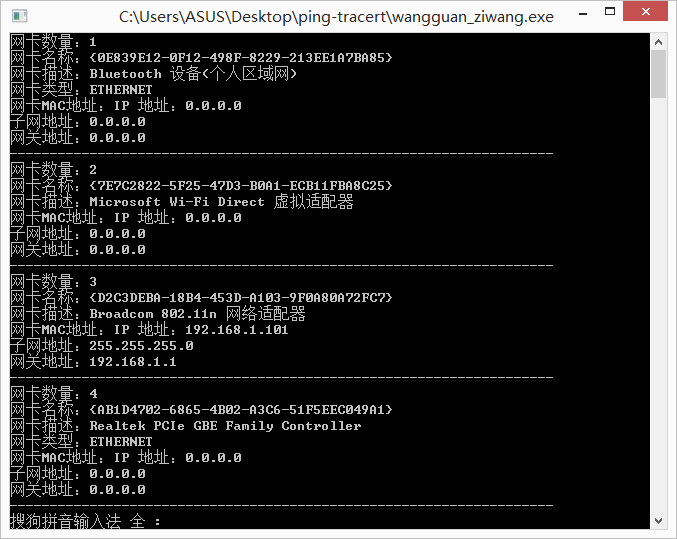
相关数据结构如下：

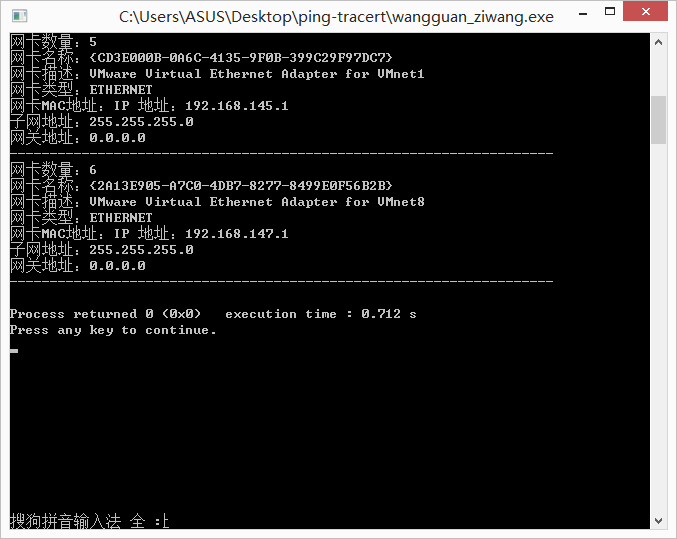
|  |
| --- |
| typedef struct \_IP\_ADDR\_STRING  {  struct \_IP\_ADDR\_STRING\* Next; //指向同类型节点，即下一个IP（如果有多IP的话）  IP\_ADDRESS\_STRING IpAddress; //IP地址信息  IP\_MASK\_STRING IpMask; //IP子网掩码  DWORD Context;// 网络表入口。这个值对应着AddIPAddredd和DeleteIPAddress函数中的NTEContext参数  } IP\_ADDR\_STRING; |
| typedef struct \_IP\_ADAPTER\_INFO  {  struct \_IP\_ADAPTER\_INFO\* Next;//指向链表中下一个适配器信息的指针  DWORD ComboIndex;//预留值  char AdapterName[MAX\_ADAPTER\_NAME\_LENGTH + 4];//使用ANSI字符串表示的适配器名称  char Description[MAX\_ADAPTER\_DESCRIPTION\_LENGTH + 4];//使用ANSI字符串表示的适配器描述  UINT AddressLength;//适配器硬件地址以字节计算的长度  BYTE Address[MAX\_ADAPTER\_ADDRESS\_LENGTH];//硬件地址以BYTE数组所表示  DWORD Index;//适配器索引  UINT Type;//适配器类型,主要有以下几种：  /\*  \* MIB\_IF\_TYPE\_OTHER 1  \* MIB\_IF\_TYPE\_ETHERNET 6  \* MIB\_IF\_TYPE\_TOKENRING 9  \* MIB\_IF\_TYPE\_FDDI 15  \* MIB\_IF\_TYPE\_PPP 23  \* MIB\_IF\_TYPE\_LOOPBACK 24  \* MIB\_IF\_TYPE\_SLIP 28  \*/  /\*UINT DhcpEnabled;//指定这个适配器是否开启DHCP  PIP\_ADDR\_STRING CurrentIpAddress;//预留值  IP\_ADDR\_STRING IpAddressList;//该适配器的IPv4地址链表  IP\_ADDR\_STRING GatewayList;//该适配器的网关IPv4地址链表  IP\_ADDR\_STRING DhcpServer;//该适配器的DHCP服务器的IPv4 地址链表  BOOL HaveWins;  IP\_ADDR\_STRING PrimaryWinsServer;  IP\_ADDR\_STRING SecondaryWinsServer;  time\_t LeaseObtained;  time\_t LeaseExpires;  } IP\_ADAPTER\_INFO,\*PIP\_ADAPTER\_INFO; |

源程序及注释如下：

|  |
| --- |
| #include <WinSock2.h>  #include <Iphlpapi.h>  #include <stdlib.h>  #include<stdio.h>  #include <iostream>  using namespace std;  int main()  {  //PIP\_ADAPTER\_INFO结构体指针存储本机网卡信息  PIP\_ADAPTER\_INFO pIpAdapterInfo = new IP\_ADAPTER\_INFO();  //得到结构体大小,用于GetAdaptersInfo参数  unsigned long stSize = sizeof(IP\_ADAPTER\_INFO);  //调用GetAdaptersInfo函数,填充pIpAdapterInfo指针变量;其中stSize参数既是一个输入量也是一个输出量  int nRel = GetAdaptersInfo(pIpAdapterInfo,&stSize);  //记录网卡数量  int netCardNum = 0;  if (ERROR\_BUFFER\_OVERFLOW == nRel)  {  //如果函数返回的是ERROR\_BUFFER\_OVERFLOW  //则说明GetAdaptersInfo参数传递的内存空间不够,同时其传出stSize,表示需要的空间大小  //这也是说明为什么stSize既是一个输入量也是一个输出量  //释放原来的内存空间  delete pIpAdapterInfo;  //重新申请内存空间用来存储所有网卡信息  pIpAdapterInfo = (PIP\_ADAPTER\_INFO)new BYTE[stSize];  //再次调用GetAdaptersInfo函数,填充pIpAdapterInfo指针变量  nRel=GetAdaptersInfo(pIpAdapterInfo,&stSize);  }  if (ERROR\_SUCCESS == nRel)  {  //输出网卡信息  //可能有多网卡,因此通过循环去判断  while (pIpAdapterInfo)  {  cout<<"网卡数量："<<++netCardNum<<endl;  cout<<"网卡名称："<<pIpAdapterInfo->AdapterName<<endl;  cout<<"网卡描述："<<pIpAdapterInfo->Description<<endl;  switch(pIpAdapterInfo->Type)  {  case MIB\_IF\_TYPE\_OTHER:  cout<<"网卡类型："<<"OTHER"<<endl;  break;  case MIB\_IF\_TYPE\_ETHERNET:  cout<<"网卡类型："<<"ETHERNET"<<endl;  break;  case MIB\_IF\_TYPE\_TOKENRING:  cout<<"网卡类型："<<"TOKENRING"<<endl;  break;  case MIB\_IF\_TYPE\_FDDI:  cout<<"网卡类型："<<"FDDI"<<endl;  break;  case MIB\_IF\_TYPE\_PPP:  cout<<"PP\n";  cout<<"网卡类型："<<"PPP"<<endl;  break;  case MIB\_IF\_TYPE\_LOOPBACK:  cout<<"网卡类型："<<"LOOPBACK"<<endl;  break;  case MIB\_IF\_TYPE\_SLIP:  cout<<"网卡类型："<<"SLIP"<<endl;  break;  default:  break;  }  cout<<"网卡MAC地址：";  char acMAC[18];  //可能网卡有多IP,因此通过循环去判断  IP\_ADDR\_STRING \*pIpAddrString1 =&(pIpAdapterInfo->IpAddressList);  do  {  IP\_ADDR\_STRING \*pIpAddrString =&(pIpAdapterInfo->IpAddressList);  cout<<"IP 地址："<<pIpAddrString->IpAddress.String<<endl;  cout<<"子网地址："<<pIpAddrString->IpMask.String<<endl;  cout<<"网关地址："<<pIpAdapterInfo->GatewayList.IpAddress.String<<endl;  pIpAddrString=pIpAddrString->Next;  pIpAddrString1= pIpAddrString;  } while (pIpAddrString1);  pIpAdapterInfo = pIpAdapterInfo->Next;  cout<<"--------------------------------------------------------------------"<<endl;  }  }  //释放内存空间  if (pIpAdapterInfo)  {  delete pIpAdapterInfo;  }  return 0;  } |

运行结果显示如下：





从上述结果可知，输出本计算机通过“ipconfig /all”指令在终端获取的所有网卡的名称，描述，类型，子网掩码，网关地址等信息。上述结果显示中的三个IP地址对应的各信息均正确输出

192.168.1.101为当前本机接入的无线网IP地址

192.168.145.1与192.168.147.1为虚拟机分配的网卡IP地址

五． 实验总结

1. 通过主机名获取IP地址，最简单的方法为利用gethostname和gethostbyname两个函数

2. gethostbyname()函数的调用需要引入两个头文件“#include<netdb.h>、#include<sys/socket.h>”

3. 获取域名、子网掩码功能的实现需要对注册表中保存有网卡信息的相应键进行读取。

4. 获取网卡类型和MAC地址，需要了解网络层提供的是面向连接的可靠虚电路服务，还是无连接的不可靠服务。

**实验二、简化的ping程序和tracert程序实现**

1. 实验要求

在Windows环境下，编写程序，实现ping(即发送一个ICMP的echo报文并对目标返回的回应报文进行正确解析)、tracert程序（利用TTL递减的ICMP报文，逐跳返回从本地主机到目的主机之间的所有路由器IP地址信息）的基本功能。

1. 实验目的

掌握ICMP协议的原理，熟悉ping,tracert程序思想

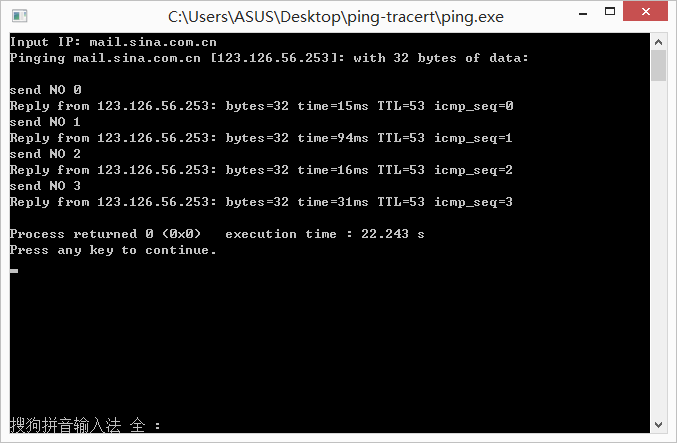
1. 实验环境

Windows8.1操作系统、codeblocks软件

1. 实验步骤
2. 配置Codeblocks与网络编程有关的运行环境，即添加相应的动态链接库。实现同实验一
3. Ping程序的实现

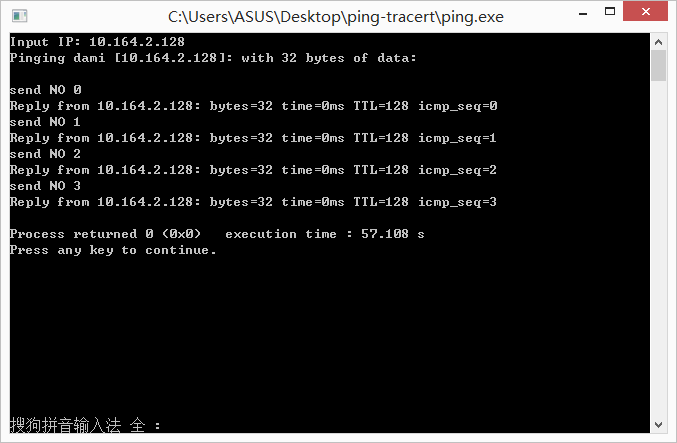
|  |
| --- |
| #include<Winsock2.h>  #include<process.h>  #include<stdio.h>  #include<Winsock2.h>  #include<process.h>  #include<stdio.h>  #define SEND\_SIZE 32  #define PACKET\_SIZE 4096  #define ICMP\_ECHO 8  #define ICMP\_ECHOREPLY 0  struct icmp  {  unsigned char icmp\_type;//类型  unsigned char icmp\_code;//编码  unsigned short icmp\_chksum;//校验和  unsigned short icmp\_id;//标示符  unsigned short icmp\_seq;//顺序号  unsigned long icmp\_data;//数据  };  struct ip  {  unsigned char ip\_hl:4;//报头长度  unsigned char ip\_v:4;//版本号  unsigned char ip\_tos;//服务类型  unsigned short ip\_len;//总长度  unsigned short ip\_id;//标识  unsigned short ip\_off;//标志  unsigned char ip\_ttl;//生存时间  unsigned char ip\_p;//协议  unsigned short ip\_sum;//报头校验和  unsigned long ip\_src;//源IP地址  unsigned long ip\_dst;//目的IP地址  };  char sendpacket[PACKET\_SIZE];  char recvpacket[PACKET\_SIZE];  struct sockaddr\_in dest\_addr;  struct sockaddr\_in from\_addr;  int sockfd;  int pid;  unsigned short cal\_chksum(unsigned short \*addr,int len);  int pack(int pack\_no);  int unpack(unsigned char \*buf,int len);  void send\_packet(void);  void recv\_packet(void);  int main(int argc, CHAR\* argv[])  {  struct hostent \*host;  struct protoent \*protocol;  int timeout=1000;  int SEND\_COUNT=4;  int i;  char \*par\_host;  char m\_Input[100];  printf("Input IP: ");//提示用户输入IP地址  gets(m\_Input);  par\_host=m\_Input;  WORD wVersionRequested;  WSADATA wsaData;  int err;  wVersionRequested = MAKEWORD( 2, 2 );//返回无符号的16位整型数  err = WSAStartup( wVersionRequested, &wsaData );//本函数必须是应用程序或DLL调用的第一个Windows Sockets函数  if ( err != 0 )  if(LOBYTE( wsaData.wVersion ) != 2 || HIBYTE( wsaData.wVersion ) != 2 )  {  WSACleanup( );  }  if( (protocol=getprotobyname("icmp") )==NULL)//返回对应于给定协议名的包含名字和协议号的protoent结构指针  {  printf("getprotobyname error\n");  exit(1);  }  if( (sockfd=socket(AF\_INET,SOCK\_RAW,protocol->p\_proto) )<0)  {  printf("socket error\n");  exit(1);  }  if(setsockopt(sockfd,SOL\_SOCKET,SO\_RCVTIMEO,(char\*)&timeout,sizeof(timeou))<0) //设置套接口的选项  fprintf(stderr,"failed to set recv timeout: %d\n",WSAGetLastError());  if(setsockopt(sockfd,SOL\_SOCKET,SO\_SNDTIMEO,(char\*)&timeout,sizeof(timeout))<0)  fprintf(stderr,"failed to set send timeout: %d\n",WSAGetLastError());  memset(&dest\_addr,0,sizeof(dest\_addr));//内存初始化，将dest\_addr当前位置后面的sizeof(dest\_addr)个字节用0代替  dest\_addr.sin\_family=AF\_INET;  if(host=gethostbyname(par\_host))// 返回对应于给定主机名的主机信息  {  memcpy( (char \*)&dest\_addr.sin\_addr,host->h\_addr,host->h\_length);  //从host->h\_addr所指的内存地址开始拷贝host->h\_length个 字节到dest\_addr.sin\_addr所指的内存区域  if(host=gethostbyaddr(host->h\_addr,4,PF\_INET))//返回给定主机地址的主机信息  par\_host=host->h\_name;  }  else if(dest\_addr.sin\_addr.s\_addr=inet\_addr(par\_host)==INADDR\_NONE)  //将点分十进制的IP转换成一个长整数型数，并判断  {  printf("Unkown host %s\n",par\_host);  exit(1);  }  pid=\_getpid();  printf("Pinging %s [%s]: with %d bytes of data:\n\n",par\_host,inet\_ntoa(dest\_addr.sin\_addr),SEND\_SIZE);  //将32位的二进制数转化为字符串;  for(i=0; i<SEND\_COUNT; i++) //循环发送4个报文  {  send\_packet();  recv\_packet();  Sleep(1000);  }  }  /\*产生网际校验和-校验算法  把ICMP报头二进制数据以2字节为单位累加起来  若ICMP报头为奇数个字节，会剩下最后一字节。  把最后一个字节视为一个2字节数据的高字节，  这个2字节数据的低字节为0，继续累加\*/  unsigned short cal\_chksum(unsigned short \*addr,int len)  {  int nleft=len;  int sum=0;  unsigned short \*w=addr;  unsigned short answer=0;  while(nleft>1)  {  sum+=\*w++;  nleft-=2;  }  if( nleft==1)  {  \*(unsigned char \*)(&answer)=\*(unsigned char \*)w;  sum+=answer;  }  sum=(sum>>16)+(sum&0xffff);  sum+=(sum>>16);  answer=~sum;  return answer;  }  //填充ICMP数据包各字段  int pack(int pack\_no)  {  int packsize;  struct icmp \*icmp;  packsize=8+SEND\_SIZE;  icmp=(struct icmp\*)sendpacket;  icmp->icmp\_type=ICMP\_ECHO;  icmp->icmp\_code=0;  icmp->icmp\_chksum=0;  icmp->icmp\_seq=pack\_no;  icmp->icmp\_id=pid;  icmp->icmp\_data=GetTickCount();  icmp->icmp\_chksum=cal\_chksum( (unsigned short \*)icmp,packsize); /\*校验算法\*/  return packsize;  }  //解包  int unpack(char \*buf,int len)  {  struct ip \*ip;  struct icmp \*icmp;  double rtt;  int iphdrlen;  ip=(struct ip \*)buf;  iphdrlen=ip->ip\_hl\*4;  icmp=(struct icmp \*)(buf+iphdrlen);  if( (icmp->icmp\_type==ICMP\_ECHOREPLY) &&(icmp->icmp\_id==pid) )  {  len=len-iphdrlen-8;  rtt=GetTickCount()-icmp->icmp\_data;  printf("Reply from %s: bytes=%d time=%.0fms TTL=%d icmp\_seq=%u\n",inet\_ntoa(from\_addr.sin\_addr), len, rtt, ip->ip\_ttl, icmp->icmp\_seq);  return 1;  }  return 0;  }  //发送数据包  void send\_packet()  {  int packetsize;  static int pack\_no=0;  packetsize=pack(pack\_no++);//调用ICMP填充函数  if( sendto(sockfd,sendpacket,packetsize,0,(struct sockaddr \*)&dest\_addr,sizeof(dest\_addr) )<0 )  printf("Destination host unreachable.\n");  printf("send NO %d\n",pack\_no-1);  }  //接收数据包  void recv\_packet()  {  int n,fromlen;  int success;  fromlen=sizeof(from\_addr);  do  {  if( (n=recvfrom(sockfd,recvpacket,sizeof(recvpacket),0,(struct sockaddr \*)&from\_addr,&fromlen)) >=0)  success=unpack(recvpacket,n);//调用接收ICMP解包函数  else if (WSAGetLastError() == WSAETIMEDOUT)  {  printf("Request timed out.\n");  return;  }  }  while(!success);  } |

运行结果显示如下：



从运行结果可知，向mail.sina.com.cn发送4个ICMP数据报，程序将域名解析为IP地址123.126.56.253，并从目的主机获取到正确的发送信息。

当向本机发送ICMP数据报显示如下

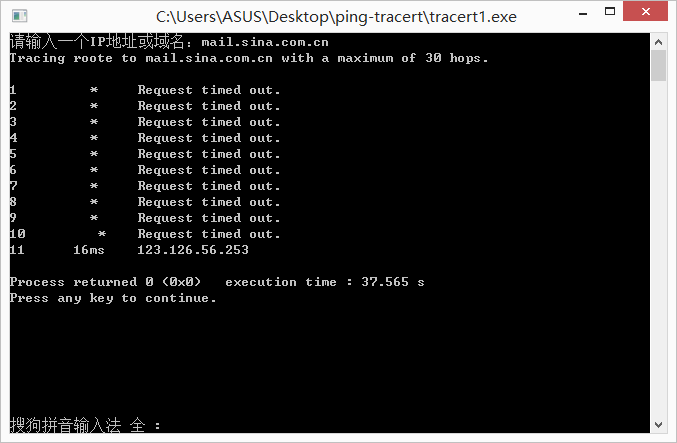


从运行结果可发现，time=0ms,数据报成功发送。

1. Tracert程序实现

|  |
| --- |
| #include <iostream>  #include <winsock2.h>  #include <ws2tcpip.h>  using namespace std;  //IP报头  typedef struct IP\_HEADER  {  unsigned char hdr\_len:4; //4位头部长度  unsigned char version:4; //4位版本号  unsigned char tos; //8位服务类型  unsigned short total\_len; //16位总长度  unsigned short identifier; //16位标识符  unsigned short frag\_and\_flags; //3位标志加13位片偏移  unsigned char ttl; //8位生存时间  unsigned char protocol; //8位上层协议号  unsigned short checksum; //16位校验和  unsigned long sourceIP; //32位源IP地址  unsigned long destIP; //32位目的IP地址  } IP\_HEADER;  //ICMP报头  typedef struct ICMP\_HEADER  {  BYTE type; //8位类型字段  BYTE code; //8位代码字段  USHORT cksum; //16位校验和  USHORT id; //16位标识符  USHORT seq; //16位序列号  } ICMP\_HEADER;  //报文解码结构  typedef struct DECODE\_RESULT  {  USHORT usSeqNo; //序列号  DWORD dwRoundTripTime; //往返时间  in\_addr dwIPaddr; //返回报文的IP地址  } DECODE\_RESULT;  //计算网际校验和函数  USHORT checksum( USHORT \*pBuf, int iSize )  {  unsigned long cksum = 0;  while( iSize > 1 )  {  cksum += \*pBuf++;  iSize -= sizeof(USHORT);  }  if( iSize )//如果 iSize 为正，即为奇数个字节  {  cksum += \*(UCHAR \*)pBuf; //则在末尾补上一个字节，使之有偶数个字节  }  cksum = ( cksum >> 16 ) + ( cksum&0xffff );  cksum += ( cksum >> 16 );  return (USHORT)( ~cksum );  }  //对数据包进行解码  BOOL DecodeIcmpResponse(char \* pBuf, int iPacketSize, DECODE\_RESULT &DecodeResult,BYTE ICMP\_ECHO\_REPLY, BYTE ICMP\_TIMEOUT)  {  //检查数据报大小的合法性  IP\_HEADER\* pIpHdr = ( IP\_HEADER\* )pBuf;  int iIpHdrLen = pIpHdr->hdr\_len \* 4; //ip报头的长度是以4字节为单位的  //若数据包大小 小于 IP报头 + ICMP报头，则数据报大小不合法  if ( iPacketSize < ( int )( iIpHdrLen + sizeof( ICMP\_HEADER ) ) )  return FALSE;  //根据ICMP报文类型提取ID字段和序列号字段  ICMP\_HEADER \*pIcmpHdr = ( ICMP\_HEADER \* )( pBuf+iIpHdrLen );//ICMP报头 = 接收到的缓冲数据 + IP报头  USHORT usID, usSquNo;  if( pIcmpHdr->type == ICMP\_ECHO\_REPLY ) //ICMP回显应答报文  {  usID = pIcmpHdr->id; //报文ID  usSquNo = pIcmpHdr->seq; //报文序列号  }  else if( pIcmpHdr->type == ICMP\_TIMEOUT )//ICMP超时差错报文  {  char \* pInnerIpHdr = pBuf + iIpHdrLen + sizeof( ICMP\_HEADER ); //载荷中的IP头  int iInnerIPHdrLen = ( ( IP\_HEADER \* )pInnerIpHdr )->hdr\_len \* 4; //载荷中的IP头长  ICMP\_HEADER \* pInnerIcmpHdr = ( ICMP\_HEADER \* )( pInnerIpHdr + iInnerIPHdrLen );//载荷中的ICMP头  usID = pInnerIcmpHdr->id; //报文ID  usSquNo = pInnerIcmpHdr->seq; //序列号  }  else  {  return false;  }  //检查ID和序列号以确定收到期待数据报  if( usID != ( USHORT )GetCurrentProcessId() || usSquNo != DecodeResult.usSeqNo )  {  return false;  }  //记录IP地址并计算往返时间  DecodeResult.dwIPaddr.s\_addr = pIpHdr->sourceIP;  DecodeResult.dwRoundTripTime = GetTickCount() - DecodeResult.dwRoundTripTime;  //处理正确收到的ICMP数据报  if ( pIcmpHdr->type == ICMP\_ECHO\_REPLY || pIcmpHdr->type == ICMP\_TIMEOUT )  {  //输出往返时间信息  if(DecodeResult.dwRoundTripTime)  cout<<" "<<DecodeResult.dwRoundTripTime<<"ms"<<flush;  else  cout<<" "<<"<1ms"<<flush;  }  return true;  }  int main()  {  //初始化Windows sockets网络环境  WSADATA wsa;  WSAStartup( MAKEWORD(2,2), &wsa );  char IpAddress[255];  cout<<"请输入一个IP地址或域名：";  cin>>IpAddress;  //得到IP地址  u\_long ulDestIP = inet\_addr( IpAddress );  //转换不成功时按域名解析  if( ulDestIP == INADDR\_NONE )  {  hostent \* pHostent = gethostbyname( IpAddress );  if( pHostent )  {  ulDestIP = ( \*( in\_addr\* )pHostent->h\_addr).s\_addr;  }  else  {  cout<<"输入的IP地址或域名无效!"<<endl;  WSACleanup();  }  }  cout<<"Tracing roote to "<<IpAddress<<" with a maximum of 30 hops.\n"<<endl;  //填充目的端socket地址  sockaddr\_in destSockAddr;  ZeroMemory( &destSockAddr, sizeof( sockaddr\_in ) );  destSockAddr.sin\_family = AF\_INET;  destSockAddr.sin\_addr.s\_addr = ulDestIP;  //创建原始套接字  SOCKET sockRaw = WSASocket( AF\_INET, SOCK\_RAW, IPPROTO\_ICMP, NULL, 0, WSA\_FLAG\_OVERLAPPED );  //超时时间  int iTimeout = 3000;  //设置接收超时时间  setsockopt( sockRaw, SOL\_SOCKET, SO\_RCVTIMEO, (char \*)&iTimeout,  sizeof( iTimeout ) );  //设置发送超时时间  setsockopt(sockRaw,SOL\_SOCKET,SO\_SNDTIMEO,(char \*)&iTimeout,sizeof(iTimeout));  //构造ICMP回显请求消息，并以TTL递增的顺序发送报文  //ICMP类型字段  const BYTE ICMP\_ECHO\_REQUEST = 8; //请求回显  const BYTE ICMP\_ECHO\_REPLY = 0; //回显应答  const BYTE ICMP\_TIMEOUT = 11; //传输超时  //其他常量定义  const int DEF\_ICMP\_DATA\_SIZE = 32; //ICMP报文默认数据字段长度  const int MAX\_ICMP\_PACKET\_SIZE = 1024; //ICMP报文最大长度（包括报头）  const DWORD DEF\_ICMP\_TIMEOUT = 3000; //回显应答超时时间  const int DEF\_MAX\_HOP = 30; //最大跳站数  //填充ICMP报文中每次发送时不变的字段  char IcmpSendBuf[ sizeof( ICMP\_HEADER ) + DEF\_ICMP\_DATA\_SIZE ];//发送缓冲区  memset( IcmpSendBuf, 0, sizeof( IcmpSendBuf ) ); //初始化发送缓冲区  char IcmpRecvBuf[ MAX\_ICMP\_PACKET\_SIZE ]; //接收缓冲区  memset( IcmpRecvBuf, 0, sizeof( IcmpRecvBuf ) ); //初始化接收缓冲区  ICMP\_HEADER \* pIcmpHeader = ( ICMP\_HEADER\* )IcmpSendBuf;  pIcmpHeader->type = ICMP\_ECHO\_REQUEST; //类型为请求回显  pIcmpHeader->code = 0; //代码字段为0  pIcmpHeader->id = (USHORT)GetCurrentProcessId(); //ID字段为当前进程号  memset( IcmpSendBuf + sizeof( ICMP\_HEADER ), 'E', DEF\_ICMP\_DATA\_SIZE );//数据字段  USHORT usSeqNo = 0; //ICMP报文序列号  int iTTL = 1; //TTL初始值为1  BOOL bReachDestHost = FALSE; //循环退出标志  int iMaxHot = DEF\_MAX\_HOP; //循环的最大次数  DECODE\_RESULT DecodeResult; //传递给报文解码函数的结构化参数  while( !bReachDestHost && iMaxHot-- )  {  //设置IP报头的TTL字段  setsockopt( sockRaw, IPPROTO\_IP, IP\_TTL, (char \*)&iTTL, sizeof(iTTL) );  cout<<iTTL<<flush; //输出当前序号,flush表示将缓冲区的内容马上送进cout,把输出缓冲区刷新  //填充ICMP报文中每次发送变化的字段  ((ICMP\_HEADER \*)IcmpSendBuf)->cksum = 0; //校验和先置为0  ((ICMP\_HEADER \*)IcmpSendBuf)->seq = htons(usSeqNo++); //填充序列号  ((ICMP\_HEADER \*)IcmpSendBuf)->cksum =  checksum( ( USHORT \* )IcmpSendBuf, sizeof( ICMP\_HEADER ) + DEF\_ICMP\_DATA\_SIZE ); //计算校验和  //记录序列号和当前时间  DecodeResult.usSeqNo = ( ( ICMP\_HEADER\* )IcmpSendBuf )->seq; //当前序号  DecodeResult.dwRoundTripTime = GetTickCount(); //当前时间  //发送TCP回显请求信息  sendto( sockRaw, IcmpSendBuf, sizeof(IcmpSendBuf), 0, (sockaddr\*)&destSockAddr, sizeof(destSockAddr) );  //接收ICMP差错报文并进行解析处理  sockaddr\_in from; //对端socket地址  int iFromLen = sizeof(from);//地址结构大小  int iReadDataLen; //接收数据长度  while(1)  {  //接收数据  iReadDataLen = recvfrom( sockRaw, IcmpRecvBuf, MAX\_ICMP\_PACKET\_SIZE, 0, (sockaddr\*)&from, &iFromLen );  if( iReadDataLen != SOCKET\_ERROR )//有数据到达  {  //对数据包进行解码  if(DecodeIcmpResponse( IcmpRecvBuf, iReadDataLen, DecodeResult, ICMP\_ECHO\_REPLY, ICMP\_TIMEOUT ) )  {  //到达目的地，退出循环  if( DecodeResult.dwIPaddr.s\_addr == destSockAddr.sin\_addr.s\_addr )  bReachDestHost = true;  //输出IP地址  cout<<'\t'<<inet\_ntoa( DecodeResult.dwIPaddr )<<endl;  break;  }  }  else if( WSAGetLastError() == WSAETIMEDOUT ) //接收超时，输出\*号  {  cout<<" \*"<<'\t'<<"Request timed out."<<endl;  break;  }  else  {  break;  }  }  iTTL++; //递增TTL值  }  } |

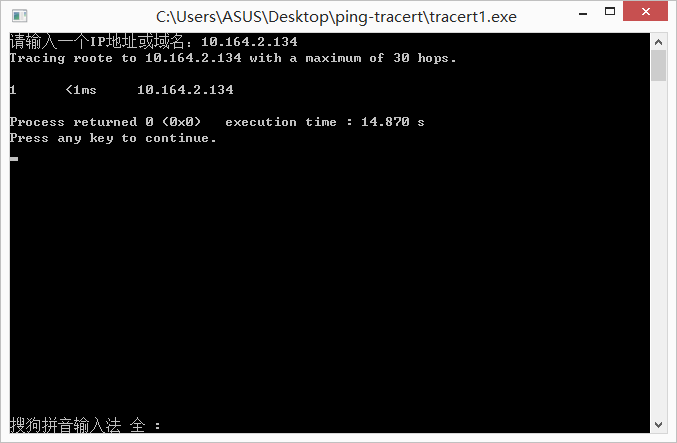
运行结果显示如下：



从运行结果可以分析，tracert实现的是源主机与目的主机之间的站点跟踪，从最后一跳的显示信息可以知道，报文成功发送到目的端，但前10跳都显示”Request timed out”，通过网上相关资料说明，解释为路由器屏蔽了tracert功能。

ICMP提供一种把通信服务中的差错向原站点报告的机制。

当tracert当前计算机所连校园网，分配到的WLAN IP时，显示如下：



因为本机的WLAN IP即为10.164.2.134，所以当在跟踪这条链路上的路由器的情况时，只有有且仅有一跳显示信息。

注：ping程序与tracert程序的运行，也可以通过Windows的终端命令行cmd实现，但前期需要配置一些动态链接库。本实验在采用该方式运行程序时，根据计算机的系统属性（64位操作系统）需要配置如下文件到C:\Windows\SysWOW64\文件夹下：

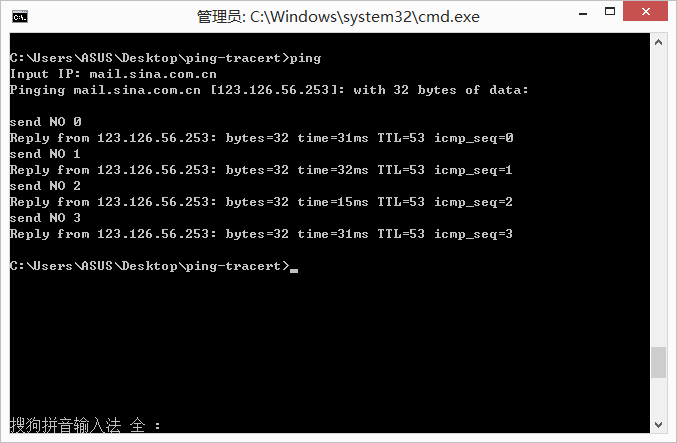
Mfc42d.dll

Mfcn42d.dll

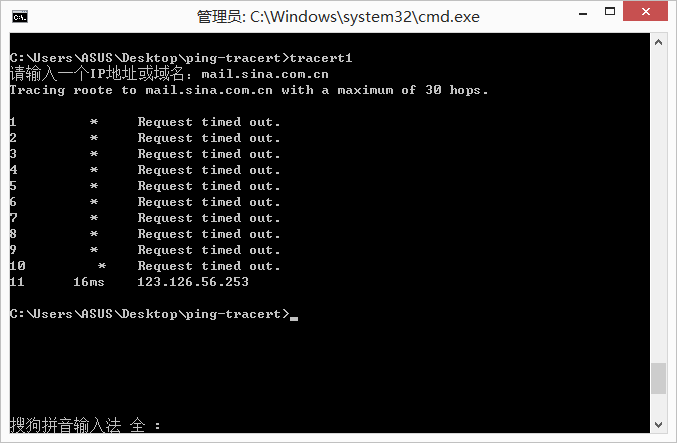
Msvcrtd.dll

Mfco42d.dll

命令行运行ping程序显示如下：



命令行运行tracert程序显示如下：



1. 实验总结
2. ICMP网际控制协议两个典型的应用为“ping、tracert”，ICMP允许向主机或路由器报告差错情况或提供有关异常情况的报告。当一个分组消息无法到达目的站点或TTL超时后，路由器就会丢弃此分组，并向原站点返回一个目的站点不可到达的ICMP报文
3. Ping 程序的实现是通过主机向目的主机发出ICMP回显请求以后，目的主机会拦截这个请求，然后生成一条回显应答请求，再通过网络传回主机。若因为某些原因，不能抵达主机，就会生成对应的ICMP错误消息，由通信路径上的某台路由器返回。当与目的主机的连接不存在问题，但目的主机已经关机或没有设置对网络事件作出响应，仅需要由原程序执行时间超时判断。
4. Tracert程序的设计时通过像目的主机发送一个端口不可达的UDP报文，并重复递增IP地址的TTL字段。初始时，ttl等于1，一旦抵达第一个路由器ttl就会变为0,这时路由器生成ICMP超时报文，然后再把TTL递增，本地主机将返回的每一条ICMP消息收集下来后进行解析，便可以读出到目的主机的路径上的每一台路由器的IP地址。

**实验三、 端口扫描**

1. 实验要求

在Windows环境下编写程序，实现对目标主机的指定端口进行扫描，测试端口是否为工作状态

1. 实验目的

掌握端口扫描的原理、熟练编写端口扫描程序

1. 实验环境

Windows8.1操作系统、VC++软件

1. 程序流程

初始化

清空列表框

输入主机地址并判断是否合法

N

弹出提示对话框

Y

IP地址格式转化

是否为单端口扫描

Y

向端口发起连接

N

连接是否打开

最大端口是否大于最小端口

Y

N

向报表视图循环插入扫描结果

弹出提示对话框

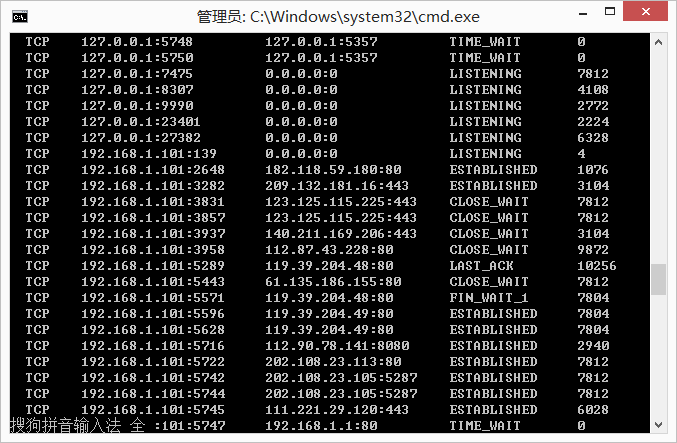
结束

结果保存成文件

1. 实验步骤
2. 通过“netstat -ano”命令在终端查看本计算机各端口的状态情况，用于后续程序运行结果的验证。

部分截图入下：





当前计算机所连无线网IP地址为：192.168.1.101

1. 配置程序运行环境

该程序实现框架为MFC模式，在VC++6.0集成开发环境下运行。且工程需要引用的头文件及动态链接库文件默认在C：\Program files\Microsoft Visual Studio\VC98。因 此确保VC++的安装路径与之匹配。

需要在C:\Windows\SysWOW64\文件夹下导入如下文件：

Mfc42d.dll

Mfcn42d.dll

Msvcrtd.dll

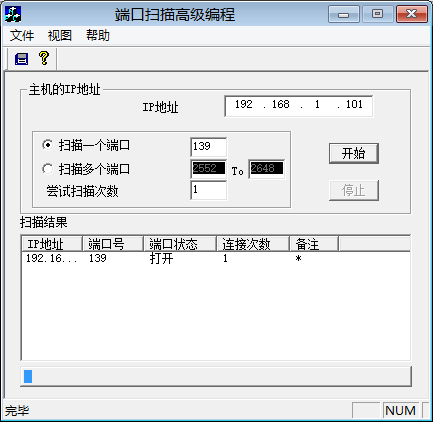
Mfco42d.dll

1. 核心代码注释如下：

|  |
| --- |
| **//开始扫描按钮处理程序**  void CPortScanView::OnButtonScan()  {  CString btnTxt,IP;  UINT nSinglePort;  BYTE f1,f2,f3,f4;//表标识IP地址的4个字节  TCHAR temp[10] = "\0";  m\_cProgress.SetPos(0);  m\_cResult.DeleteAllItems();  //清空列表框  POSITION p = m\_pStatusList->GetHeadPosition();  while (p)  {  POSITION temp = p;  DATA\* pNode = (DATA\*)m\_pStatusList->GetNext(p);  m\_pStatusList->RemoveAt(temp);  //循环删除  if (pNode)  delete pNode;  }  //验证IP地址是否为空  if (m\_cIP.IsBlank())  {  MessageBox(\_T("请输入IP地址."),  \_T("Error"),  MB\_OK | MB\_ICONEXCLAMATION);  return;  }    //验证IP地址是否正确  if (m\_cIP.GetAddress(f1,f2,f3,f4) < 4)  {  MessageBox(\_T("请确认IP地址."),  \_T("Invalid IP address"),MB\_OK | MB\_ICONEXCLAMATION);  return;  }  //进行IP地址转化  IP = \_itoa(f1,temp,10);  IP += \_T('.');  IP += \_itoa(f2,temp,10);  IP += \_T('.');  IP += \_itoa(f3,temp,10);  IP += \_T('.');  IP += \_itoa(f4,temp,10);  m\_cBtnStop.EnableWindow();  m\_cBtnScan.EnableWindow(FALSE);  //是否是单个端口扫描  if (m\_bSinglePort)  {  CString port;  m\_cSinglePort.GetWindowText(port);  m\_minPort = m\_maxPort = nSinglePort = atoi(port);  }  else//多个端口进行扫描  {  CString port1,port2;  m\_cPortFrom.GetWindowText(port1);  m\_cPortTo.GetWindowText(port2);  m\_minPort = atoi(port1);  m\_maxPort = atoi(port2);  m\_cProgress.SetRange32(0,m\_maxPort-m\_minPort+1);  m\_cProgress.SetStep(1);  }    if (!m\_bSinglePort && m\_maxPort < m\_minPort)  {  MessageBox(\_T("最大端口要大于最小端口."),  \_T("Caution"),  MB\_OK | MB\_ICONINFORMATION);  return;  }  UINT m\_nMaxAttempts = GetDlgItemInt(IDC\_EDIT\_ATTEMPTS);  for (m\_nCounter = m\_minPort; m\_nCounter <= m\_maxPort; m\_nCounter++)  {  BOOL bIsOpen = FALSE;  UINT nAttempt = 1;    while(nAttempt <= m\_nMaxAttempts && !bIsOpen)  {  TCHAR temp[10]="\0";  CString str = \_T("连接端口# ");  #ifdef \_UNICODE  str += \_itow(m\_nCounter,temp,10);  #else  str += itoa(m\_nCounter,temp,10);  #endif  str += \_T(", IP地址=");  str += IP;  str += \_T(", Attempt=");  #ifdef \_UNICODE  str += \_itow(nAttempt,temp,10);  #else  str += itoa(nAttempt,temp,10);  #endif  //设定状态栏  m\_parent->SetStatusBarText(str);  str.Empty();  //打开连接，测试端口状态  bIsOpen = TestConnection(IP,m\_nCounter);  //判断是否打开  if (bIsOpen)  {  DATA\* pNode = new DATA;  ASSERT(pNode);  strcpy(pNode->IPAddress,IP.GetBuffer(IP.GetLength()));  strcpy(pNode->port,\_itoa(m\_nCounter,temp,10));  pNode->bStatus = 1; //open  pNode->nAttempts = nAttempt;  m\_pStatusList->AddTail(pNode);  }  //连接次数  nAttempt++;  }  //如果还是无法扫描成功  if (!bIsOpen)  {  DATA\* pNode = new DATA;  ASSERT(pNode);  strcpy(pNode->IPAddress,IP.GetBuffer(IP.GetLength()));  strcpy(pNode->port,\_itoa(m\_nCounter,temp,10));  pNode->bStatus = 0; //关闭状态  pNode->nAttempts = nAttempt-1;  m\_pStatusList->AddTail(pNode);  }    MSG message;  if (::PeekMessage(&message,NULL,0,0,PM\_REMOVE))  {  ::TranslateMessage(&message);  ::DispatchMessage(&message);  }  m\_cProgress.StepIt();  }  //设定状态栏  m\_parent->SetStatusBarText((CString)\_T("完毕"));  m\_cBtnScan.EnableWindow();  m\_cBtnStop.EnableWindow(FALSE);  //填充报表视图  UINT nIndex = 0;  POSITION pos = m\_pStatusList->GetHeadPosition();  //循环插入扫描结果  while (pos)  {  DATA\* pNode = (DATA\*)m\_pStatusList->GetNext(pos);  AddItem(nIndex,0,pNode->IPAddress);  AddItem(nIndex,1,pNode->port);  if (pNode->bStatus)  {  AddItem(nIndex,2,\_T("打开"));  AddItem(nIndex,4,\_T("\*"));  }  else  {  AddItem(nIndex,2,\_T("关闭"));  AddItem(nIndex,4,\_T(" "));  }  AddItem(nIndex++,3,\_itoa(pNode->nAttempts,temp,10));  }  } |
| **//打开连接**  BOOL CPortScanView::TestConnection(CString IP, UINT nPort)  {  CTheSocket\* pSocket;  pSocket = new CTheSocket;  ASSERT(pSocket);  //创建socket  if (!pSocket->Create())  {  //如果创建失败，则删除，返回 false  delete pSocket;  pSocket = NULL;  return FALSE;  }  //连接被连接的主机地址和指定端口  while (!pSocket->Connect(IP , nPort))  {  //如果失败返回false  delete pSocket;  pSocket = NULL;  return FALSE;  }  //关闭socket  pSocket->Close();  delete pSocket;  return TRUE;  }  void CPortScanView::OnButtonStop()  {  m\_nCounter = m\_maxPort+1;  m\_cBtnStop.EnableWindow(FALSE);  m\_cBtnScan.EnableWindow();  m\_parent->SetStatusBarText((CString)\_T("Ready"));  } |
|  |
| **//保存成文件**  void CPortScanView::OnFileSave()  {  CFileDialog\* pDlg = new CFileDialog(FALSE,  \_T("txt"),NULL,  OFN\_OVERWRITEPROMPT |  OFN\_EXPLORER | OFN\_LONGNAMES,  \_T("Scanned ports files (port.txt)"),this);  ASSERT(pDlg);  if (pDlg->DoModal() == IDOK)  {  int nHandle,retVal;  nHandle = \_open(pDlg->GetPathName(),\_O\_BINARY | \_O\_CREAT | \_O\_TRUNC | \_O\_RDWR);  if (nHandle == -1)  {  MessageBox(\_T("Unable to open output file to write."),  \_T("Error"),  MB\_OK | MB\_ICONEXCLAMATION);  delete pDlg;  return;  }  POSITION pos = m\_pStatusList->GetHeadPosition();  while (pos)  {  CString buffer;  DATA\* pNode = (DATA\*)m\_pStatusList->GetNext(pos);  buffer = pNode->IPAddress;  buffer += \_T(',');  buffer += pNode->port;  buffer += \_T(',');  if (pNode->bStatus)  buffer += \_T("Open");  else  buffer += \_T("Close");  buffer += \_T("\r\n\0");  retVal = \_write(nHandle,  (void\*)buffer.GetBuffer(buffer.GetLength()),  buffer.GetLength());    if (retVal != buffer.GetLength())  {  MessageBox(\_T("An error occured while writing records."),  \_T("Error"),MB\_OK | MB\_ICONEXCLAMATION);  delete pDlg;  return;  }  buffer.Empty();  }  \_close(nHandle);  }  delete pDlg;  } |

程序运行结果

1. 测试本机IP地址的一个端口



上述结果与实际端口状态相符



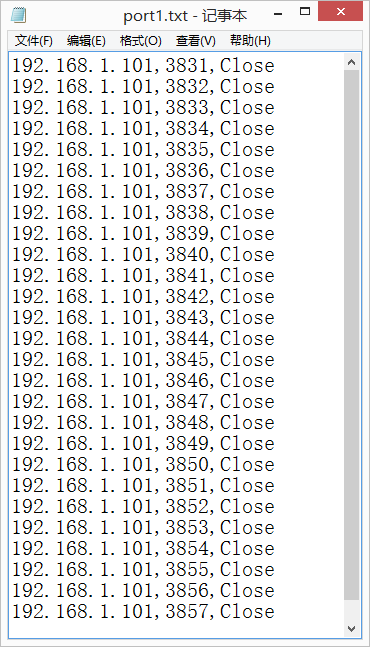
1. 测试本机IP地址的多个端口，并保存成文件



上述结果与实际端口状态相符



保存的端口状态扫描文件如下：



从上述运行结果可体会到端口扫描程序的实现原理，通过指定的扫描IP地址和端口信息，创建一个Socket，并向指定的IP地址与端口发起连接请求，如果连接成功建立，说明端口处于“打开状态”,否则端口关闭，无法连接。

该程序在实现时，用户点击“开始”按钮，便出发“OnButtonScan()”函数，在该函数中要调用“TestConnect()”函数用于检测端口状态。当扫描完所有待测端口后，选择是否将扫描结果保存成文件。

1. 实验总结
2. 该程序的实现原理为，对每一个指定需要扫描的ip地址和端口，创建一个socket，然后向指定的IP地址和端口发起连接请求，如果成功建立连接，说明该端口处于工作状态，如果连接不成功，说明端口已经被关闭。
3. 基于socket的编程时网络层面向连接的虚电路服务，它属于可靠连接的服务
4. Socket套接字也叫“插口”，由IP地址和端口构成，标识一种应用进程。

**实验四、基于TCP的客户机/服务器的实现**

1. 实验要求

利用Socket套接字实现基于TCP协议的客户端/服务器端的双向通信。要求双方通信互不干扰。可以连续发送多条信息。

1. 实验目的

理解TCP协议的工作原理，了解面向连接的C/S程序的工作模式，掌握基于Socket编程的思想

1. 实验环境

Windoes8.1操作系统，VC++6.0j集成开发环境

1. 实验原理

本实验利用Socket套接字编程实现客户端与服务器端的聊天程序。程序设计思想流程如下：

创建Socket套接字

Connect()将套接字与服务器连接

创建Socket套接字

Recv()接收数据

Close()关闭套接字

Recv()接收数据

Send()发送数据

Close()关闭套接字

Send()发送数据

等待客户端连接请求的到来

Accept()接收连接

Listen()侦听是否有客户端连接请求

Bind()套接字与本地地址绑定

数据同步

数据同步

1. 实验步骤
2. 运行环境配置：

需要在C:\Windows\SysWOW64\文件夹下导入如下文件：

Mfc42d.dll

Mfcn42d.dll

Msvcrtd.dll

Mfco42d.dll

1. 核心代码及注释如下：

服务器端

|  |
| --- |
| **对话框初始化函数**  BOOL CCSocketDlg::OnInitDialog()  {  CDialog::OnInitDialog();  ASSERT((IDM\_ABOUTBOX & 0xFFF0) == IDM\_ABOUTBOX);  ASSERT(IDM\_ABOUTBOX < 0xF000);  CMenu\* pSysMenu = GetSystemMenu(FALSE);  if (pSysMenu != NULL)  {  CString strAboutMenu;  strAboutMenu.LoadString(IDS\_ABOUTBOX);  if (!strAboutMenu.IsEmpty())  {  pSysMenu->AppendMenu(MF\_SEPARATOR);  pSysMenu->AppendMenu(MF\_STRING, IDM\_ABOUTBOX, strAboutMenu);  }  }  SetIcon(m\_hIcon, TRUE); // Set big icon  SetIcon(m\_hIcon, FALSE); // Set small icon    count=0;  m\_list.InsertColumn(0,"消息");  m\_list.SetColumnWidth(0,435);  m\_edit.SetLimitText(99);  for (int i=0;i<50;i++)  msgsock[i]=NULL;  //设定地址，INADDR\_ANY代表有系统随机指定一块网卡地址  serv.sin\_addr.s\_addr=htonl(INADDR\_ANY);  serv.sin\_family=AF\_INET;  serv.sin\_port=5000;//htons(5000);//htons函数进行字节顺序转换  addlen=sizeof(serv);  m\_button.EnableWindow(FALSE);  //创建socket  sock=socket(AF\_INET,SOCK\_STREAM,0);  //将套接字和地址serv指定的网卡绑定在一起  if (bind(sock,(sockaddr\*)&serv,addlen))  {  m\_edit.SetWindowText("绑定错误");  }else  {  m\_edit.SetWindowText("服务器创建成功");  //开始侦听  listen(sock,5);  //调用线程  AfxBeginThread(&thread,0);//通过该函数创建一个线程，线程名为thread  }  return TRUE;  } |

|  |
| --- |
| **服务器线程函数**  UINT thread(LPVOID p)  {  char buff[100];  CSize size;  size.cx=0;  size.cy=30;  int s=1,msgcount,loop=1,flag=0;  //dlg执向主对话框  CCSocketDlg \*dlg=(CCSocketDlg\*)AfxGetApp()->GetMainWnd();  //获得客户端数量  msgcount=dlg->getcount();//通过getcount函数获得空闲的socket数组成员  if (msgcount==-1)  loop=0;  if(loop)  {  s=1;  //通过accept()函数接收客户端的连接请求 dlg->msgsock[msgcount]=accept(dlg->sock,(sockaddr\*)&(dlg->serv),&(dlg->addlen));  if (dlg->msgsock[msgcount]==INVALID\_SOCKET)  {  dlg->m\_edit.SetWindowText("Error accept");  }  else  {  //启动线程  AfxBeginThread(thread,0);  dlg->SetForegroundWindow();  dlg->m\_list.InsertItem(dlg->count++,"连接成功"); dlg->m\_list.InsertItem(dlg->count++,inet\_ntoa(dlg->serv.sin\_addr));  dlg->m\_list.Scroll(size);  dlg->m\_button.EnableWindow(TRUE);  while(s!=SOCKET\_ERROR)  {  //循环接收数据，通过recv函数将从客户端发送过来的数据保存在buff里面  s=recv(dlg->msgsock[msgcount],buff,100,0);  dlg->SetForegroundWindow();  if (s!=SOCKET\_ERROR)  {  dlg->m\_list.InsertItem(dlg->count++,buff);  dlg->m\_list.Scroll(size);  dlg->sendtoall(dlg->msgsock[msgcount],buff);  }  }  //当退出上面的循环，说明客户端已经断开了与服务器的连接  send(dlg->msgsock[msgcount],"Disconnected",100,0);  dlg->m\_list.InsertItem(dlg->count++,"Disconnected");  dlg->m\_list.Scroll(size);  dlg->msgsock[msgcount]=NULL;  for (int i=0;i<50;i++)  if (dlg->msgsock[i]!=NULL)  flag=1;  if (flag!=1)  dlg->m\_button.EnableWindow(FALSE);  closesocket(dlg->msgsock[msgcount]);    }  }  //终止线程  AfxEndThread(0);  return 0;  } |

|  |
| --- |
| **发送数据到界面显示函数**  void CCSocketDlg::OnButton1()  {  char buff[100];  m\_edit.GetWindowText(buff,99);  m\_edit.SetWindowText("");  m\_list.InsertItem(count++,buff);  CSize size;  size.cx=0;  size.cy=30;  m\_list.Scroll(size);  //循环向所有客户发送信息  for (int i=0;i<50;i++)  {  if (msgsock[i]!=NULL)  send(msgsock[i],buff,100,0);  }  } |
| **向所有客户发送数据到缓冲区----------当有多个客户端连接时**  void CCSocketDlg::sendtoall(SOCKET s,char \*buff)  {  for (int i=0;i<50;i++)  {  if (msgsock[i]!=NULL && msgsock[i]!=s)  //发送  send(msgsock[i],buff,100,0);  }  } |

|  |
| --- |
| **获得还没有使用的socket数组号函数**  int CCSocketDlg::getcount()  {  for (int i=0;i<50;i++)  {  if (msgsock[i]==NULL)  return i;  }  return -1;    } |

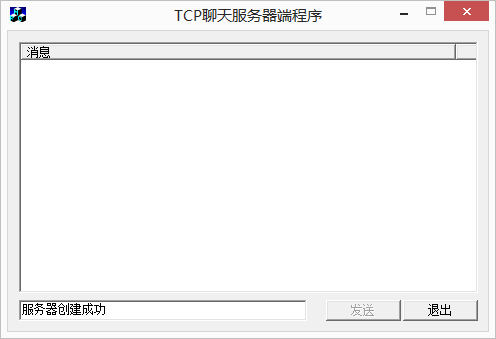
客户端

|  |
| --- |
| **连接服务器**  void CCSocketcliDlg::OnButton2()  {  char ipaddress[35];  m\_edit2.GetWindowText(ipaddress,30);  cli.sin\_addr.s\_addr=inet\_addr(ipaddress);  cli.sin\_family=AF\_INET;  cli.sin\_port=5000;//htons(5000);  //创建socket  clisock=socket(AF\_INET,SOCK\_STREAM,0);  ee=1;  //启动线程  AfxBeginThread(thread,0);    } |

|  |
| --- |
| **客户端线程**  UINT thread(LPVOID v)  {  char buff[100];  char array[25][30]=  {"155.245.160.151", "155.245.160.152","155.245.160.153",  "155.245.160.154", "155.245.160.155","155.245.160.156",  "155.245.160.157", "155.245.160.158","155.245.160.159",  "155.245.160.160","155.245.160.161","155.245.160.162",  "155.245.160.163", "155.245.160.164","155.245.160.165",  "155.245.160.166","155.245.160.167","155.245.160.168",  "155.245.160.169","155.245.160.170","155.245.160.171",  "155.245.160.172","155.245.160.173","155.245.160.174",  "155.245.160.175"};  CSize size;  size.cx=0;  size.cy=30;  int s=1,addcount=0;  CCSocketcliDlg \*dlg=(CCSocketcliDlg\*) AfxGetApp()->GetMainWnd();  dlg->m\_connect.EnableWindow(FALSE);  dlg->m\_disconnect.EnableWindow(TRUE);  //连接到服务器，利用Connect（）函数向服务器发起连接请求  while(connect(dlg->clisock,(sockaddr\*)&(dlg->cli),sizeof(dlg->cli)) && dlg->ee!=0)  {  dlg->m\_edit.SetWindowText("等待.....");  //空循环  for (int i=0;i<=65000;i++)  for(int j=0;j<=200;j++);  if (addcount==25)  addcount=0;  dlg->cli.sin\_addr.s\_addr=inet\_addr(array[addcount++]);  }  if (dlg->ee==1)  dlg->m\_list.InsertItem(dlg->count++,"连接成功");  dlg->m\_button1.EnableWindow(TRUE);  dlg->SetForegroundWindow();  //循环获得数据  while(s!=SOCKET\_ERROR && dlg->ee!=0)  {  //调用recv函数接收数据  s=recv(dlg->clisock,buff,100,0);  dlg->SetForegroundWindow();  if (s!=SOCKET\_ERROR && dlg->ee!=0)  dlg->m\_list.InsertItem(dlg->count++,buff);  dlg->m\_list.Scroll(size);  }  //发送断开命令  send(dlg->clisock,"Disconnected",100,0);  dlg->m\_button1.EnableWindow(FALSE);  dlg->m\_connect.EnableWindow(TRUE);  dlg->m\_disconnect.EnableWindow(FALSE);  closesocket(dlg->clisock);  AfxEndThread(0);  return 0;  }  CCSocketcliDlg::~CCSocketcliDlg()  {  send(clisock,"Disconnected",100,0);  } |

|  |
| --- |
| **发送信息到界面显示**  void CCSocketcliDlg::OnButton1()  {  char buff[100];  CSize size;  size.cx=0;  size.cy=30;  //获得发送信息  m\_edit.GetWindowText(buff,99);  m\_edit.SetWindowText("");  m\_list.InsertItem(count++,buff);  m\_list.Scroll(size);  //发送数据  send(clisock,buff,100,0);  } |

1. 启动服务端程序，直接双击CSocket.exe运行，结果如下：

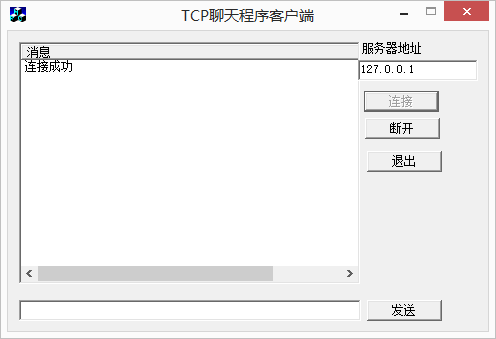


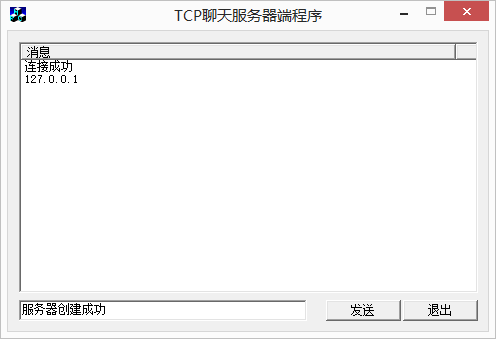
1. 启动客户端程序，直接双击CSocketcli.exe，结果如下：



服务器地址栏“192.168.0.3”显示为一个随机的IP地址

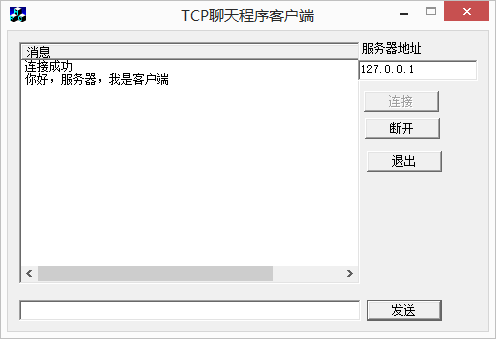
1. 指定客户端连接的服务器端的IP地址，因为本实验的客户端与服务器端程序都在本机运行，所以指定的服务器地址为“127.0.0.1”建立连接显示如下：



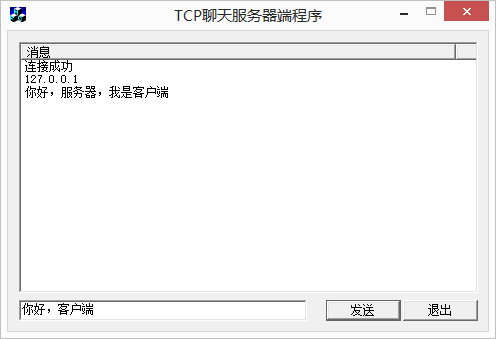


1. 客户端与服务器成功建立连接后，双方可进行通信

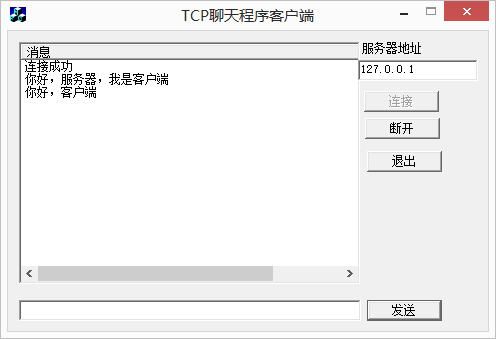
客户端发送信息



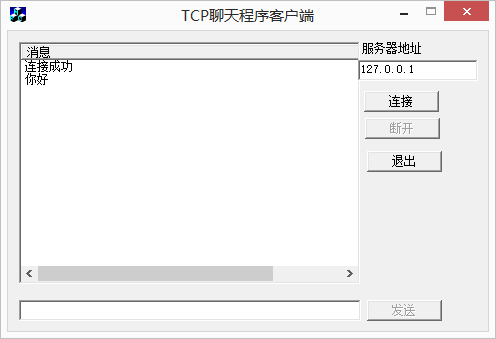
服务器端接收信息后，向客户端发送信息

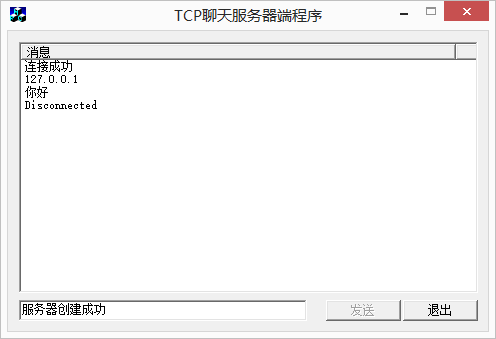


客户端接收服务器端发送的信息



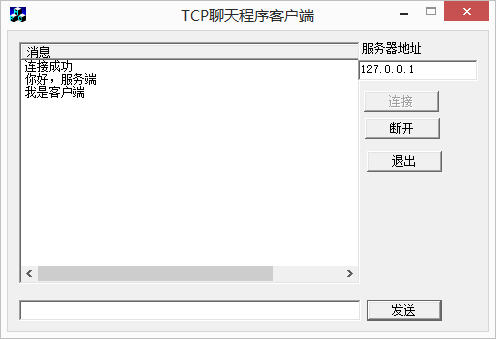
当客户端断开连接时，显示如下：



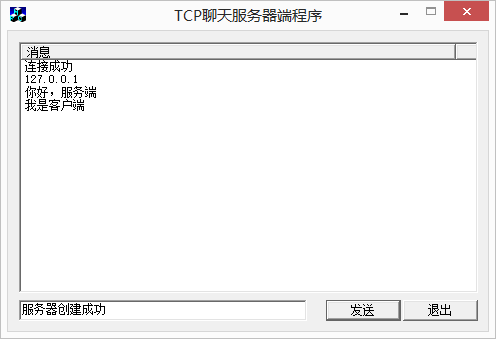


双发都不可以再发送消息

1. 客户端可向服务端连续发送多条信息

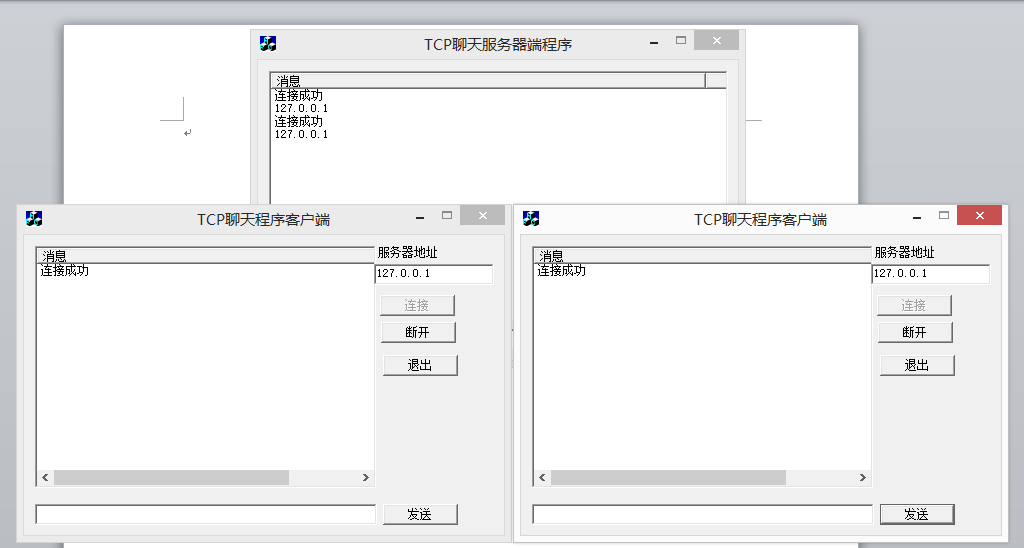


服务端可正确接收所有信息



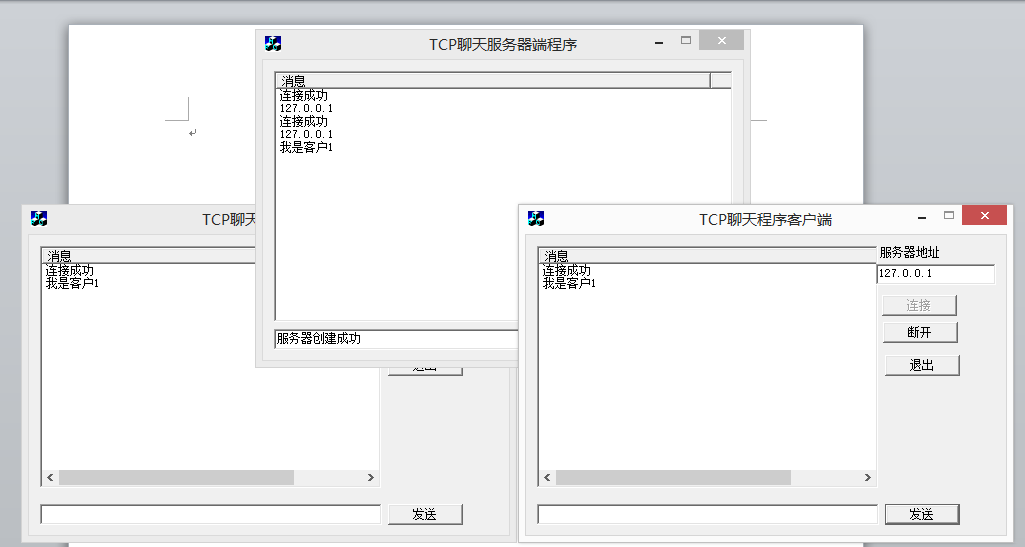
1. 当有多个客户端与服务器端连接时

连接两个客户端时，两个客户端通过中间服务器端可以进行通信，当服务器端发送消息时，所有与之相连的客户端都可以接收

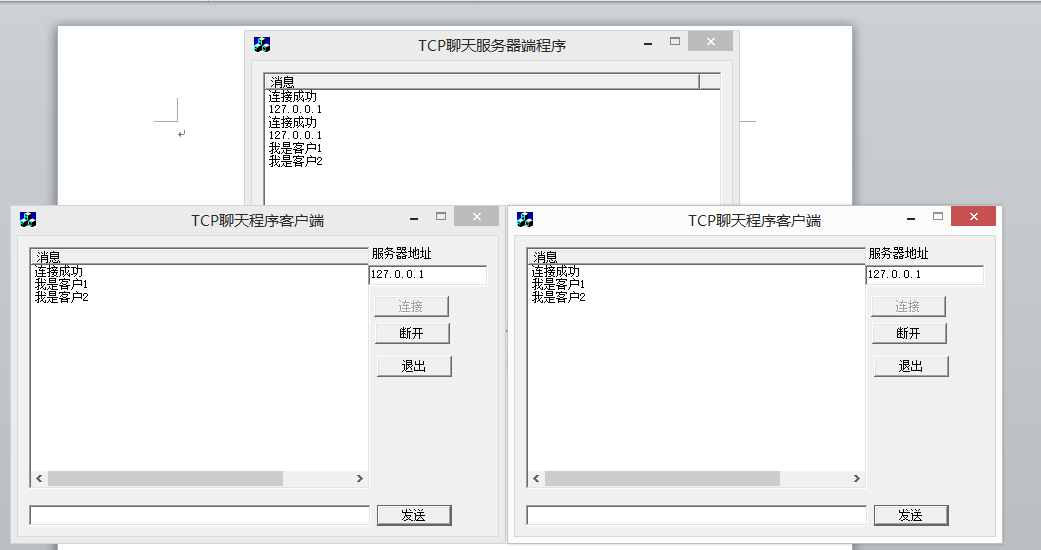


两个客户端可以进行通信：

客户1发送消息“我是客户1”

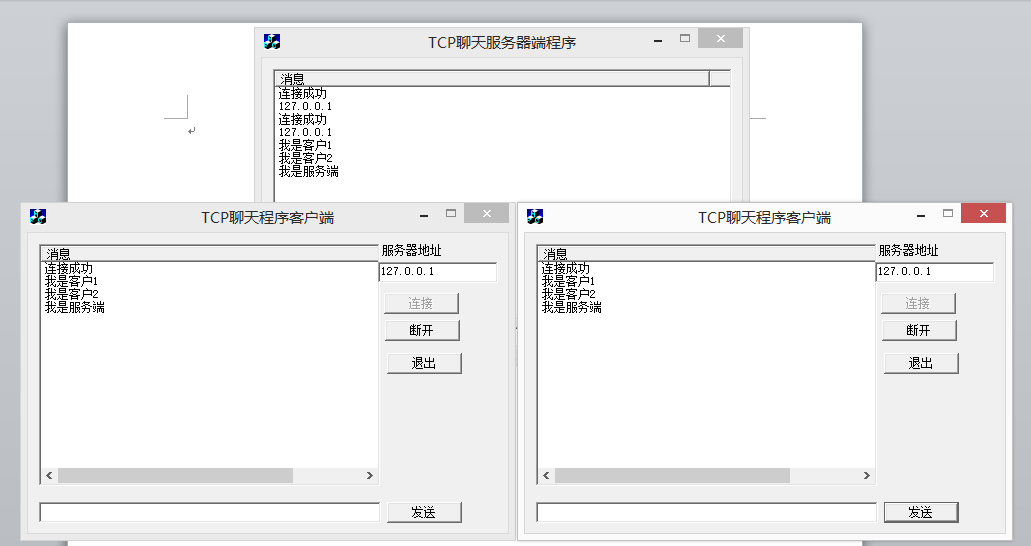


客户2发送消息“我是客户2”



服务器端发送消息，所有与之相连的客户端都可以接收：

服务端发送消息“我是服务端”



1. 实验总结
2. 服务器端需要先启动，以监听是否有客户端的连接请求。客户端需要主动发起连接请求，所有客户端主动向服务端发起连接时，需要调用connnect函数。
3. 服务端接收来自客户端的请求后，需要先判断是否有可以的Socket资源，如果没有，则不能响应客户端的连接请求。如果有可用的资源，通过Accept（）函数接收客户端的连接。
4. 服务端接收客户端的信息，需要循环实现，以便继续接收。服务端接收一个客户端的请求后，需要立即启动另个线程准备接收可能后续发起连接请求的客户端。
5. 本实验中，服务器端socket的初始化放在程序主对话框的OnInitDialog函数中，随主对话框的创建，socket也同时被创建。
6. 本实验中，客户端在点击“连接”时，相应的处理函数需要先读取服务器地址信息，将其转换成socketaddr\_in的地址信息，然后创建一个套接字，通过connect函数向服务器发起连接请求，并通过循环接收服务器端的信息。

**实验五、DNS域名解析**

1. 实验要求

利用DNS协议，设计简单的域名解析程序，实现域名到IP地址的映射解析

1. 实验目的

理解DNS协议的工作原理、熟悉DNS报文的格式及其各字段的含义，编写DNS客户端程序发送DNS解析请求以及解析DNS服务器回复报文

1. 实验环境

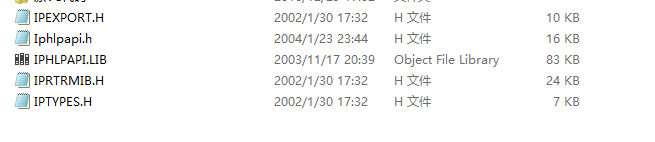
Windows8.1操作系统，VC++6.0集成开发环境

1. 实验步骤
2. 环境配置

本计算机的VC++6.0的安装路径如下：

F:\software1\VC++6.0\vc++6.0\_win8\VC98

将如下文件进行拷贝



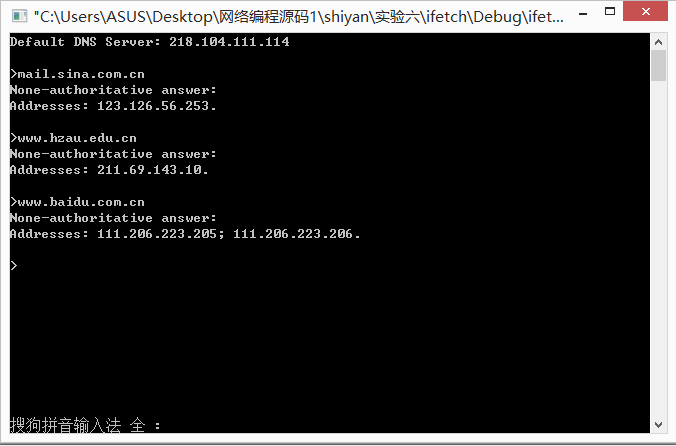
其中”.h”文件copy到VC98文件夹下的include文件夹下

其中”.lib”文件copy到VC98文件夹下的lib文件夹下

1. VC++6.0导入工程的工作空间，编译运行程序
2. 核心代码

|  |
| --- |
| **主函数** |
| int main()  {  char sendBuff[1024]; //发送缓冲区  char recvBuff[1024\*10]; //接收缓冲区  char hostname[100]; //保存DNS报文中的“域名”字段，这个字段在报文中长度为可变的  WSADATA wsaData;  WSAStartup(MAKEWORD(2,2),&wsaData);  SOCKET SocketFd;  int iServerLen;  struct sockaddr\_in ServerAddr,HostAddr;  ZeroMemory(&ServerAddr,sizeof(struct sockaddr\_in));  SocketFd = socket(AF\_INET,SOCK\_DGRAM,0); //创建一个数据报(SOCK\_DGRAM)套接字  int iTimeout = TIMEOUT;  if(setsockopt(SocketFd,SOL\_SOCKET,SO\_RCVTIMEO,(char \*)&iTimeout,sizeof(iTimeout)) == SOCKET\_ERROR) //设置接收超时时间为3000ms  {  printf("setsockopt(SO\_RCVTIMEO)  error: %d",WSAGetLastError());  return -1;  }  HostAddr.sin\_family = AF\_INET;  HostAddr.sin\_addr.s\_addr = htonl(INADDR\_ANY);  if(bind(SocketFd,(struct sockaddr \*)&HostAddr,sizeof(sockaddr\_in)) == SOCKET\_ERROR) //将创建的socket和本地网卡绑定在一起  {  printf("binding error: %d", WSAGetLastError());  return -1;  }    ServerAddr.sin\_family = AF\_INET; //获得DNS服务器地址并保存在ServerAddr里面  char dnsServer[16];  getDnsServer(dnsServer); //获取本机DNS服务器地址，作为字符串形式保存在dnsServer里面，如字符串"202.103.24.68"  printf("Default DNS Server: %s\n\n",dnsServer);  unsigned long ulDestIP = inet\_addr(dnsServer); //将点分十进制计法的字符串IP地址转换为无符号32位长整形  ServerAddr.sin\_addr.s\_addr = ulDestIP;  ServerAddr.sin\_port = htons(PORT);  iServerLen = sizeof(ServerAddr);  pDNSHDR pDnsHdr = (pDNSHDR)malloc(sizeof(DNSHDR));  pQUERYHDR pQueryHdr = (pQUERYHDR)malloc(sizeof(QUERYHDR));  while(1)  {  printf(">");  scanf("%s",hostname);  int iSendByte = genDNSPacket(pDnsHdr,pQueryHdr,hostname,sendBuff); //组装一个DNS请求报文，保存在sendBuff里面，并返回这个报文的长度保存在变量iSendBuff里面  if (iSendByte == -1)  {  return -1;  }  int iRes;  iRes = sendto(SocketFd,sendBuff,iSendByte,0,(struct sockaddr\*)&ServerAddr,iServerLen); //将sendBuff中的DNS请求报文发往DNS服务器  if(iRes == -1)  {  printf("sendto error: %d", WSAGetLastError());  return -1;  }  iRes = recvfrom(SocketFd,recvBuff,sizeof(recvBuff),0,(struct sockaddr\*)&ServerAddr,&iServerLen);  decodeDNSPacket(recvBuff); //将DNS应答报文接收到recvBuff，并进行解析  ZeroMemory(recvBuff,sizeof(recvBuff)); //清空接收缓存里面的数据  ZeroMemory(hostname,sizeof(hostname));  }  free(pDnsHdr); //释放之前malloc()申请的内存空间  free(pQueryHdr);  WSACleanup();  return 0;  } |
| **生成DNS查询报文函数** |
| //生成DNS查询报文，保存在DNSsendBuff里面，hostname为需要查询的域名字符串  int genDNSPacket(pDNSHDR pDnsHdr,pQUERYHDR pQueryHdr,char \*hostname,char \*DNSsendBuff)  {  if(!(strcmp(hostname,"exit"))) //如果在命令行输入“exit”则函数结束返回-1，程序结束  {  return -1;  }  else //正常的DNS查询请求  {  int iSendByte = 0;  ZeroMemory(DNSsendBuff,sizeof(DNSsendBuff));  pDnsHdr->id = htons(0x0000); //“标识”字段设置为0  pDnsHdr->flags = htons(0x0100); //“标志”字段设置为0x0100，即RD位为1，期望递归查询  pDnsHdr->questNum = htons(0x0001); //1个查询记录  pDnsHdr->answerNum = htons(0x0000); //没有回答记录和其他的记录  pDnsHdr->authorNum = htons(0x0000);  pDnsHdr->additionNum = htons(0x0000);  memcpy(DNSsendBuff,pDnsHdr,sizeof(DNSHDR)); //把生成的DNS查询报文首部复制到DNSsendBuff里面  iSendByte += sizeof(DNSHDR); //记录目前DNSsendBuff里面的数据量    //对域名字符串进行解析并进行形式的变换，例如将"www.wust.edu.cn\0"变成"3www4wust3edu2cn0x00"  char \*pTrace = hostname;  char \*pHostname = hostname;  int iStrLen = strlen(hostname);  unsigned char iCharNum = 0;  while(\*pTrace != '\0') //指针移到最后并从最后一个字符'\0'开始，每个字符往后移一个字节  {  pTrace++;  }  while(pTrace != hostname)  {  \*(pTrace+1) = \*pTrace;  pTrace--;  }  \*(pTrace+1) = \*pTrace; //把第一个字符移到第二个字符位置  pTrace++; //此时第一个字符没有实际意义，将指针指向原字符串中第二个字符的位置  while(\*pTrace != '\0') //从第一个字符开始扫描，iCharNum统计每两个字符'.'之间的字符数，然后填入原来字符'.'的位置  {  if(\*pTrace == '.')  {  \*pHostname = iCharNum;  iCharNum = 0;  pHostname = pTrace;  }  else  {  iCharNum ++;  }  pTrace ++;  }  \*pHostname = iCharNum; //最后一个字符'.'之后的字符数写入，例如"3www6google3com.hk"中的".hk"  memcpy(DNSsendBuff + sizeof(DNSHDR),hostname,iStrLen+2); //将解析好的“域名”字段填入 相应的位置  iSendByte += (iStrLen + 2); //之所以为iStrLen+2，是因为解析后的结果比原来多了2 个字符  //即最开始的位置需要填入一个数字和最后需要填入0x00。  pQueryHdr->type = htons(0x0001);  pQueryHdr->queryclass = htons(0x0001);  memcpy(DNSsendBuff + sizeof(DNSHDR) + iStrLen + 2,pQueryHdr,sizeof(QUERYHDR)); //在“域名”字段之后填入“查询类型”和“查询类”的值  iSendByte += sizeof(QUERYHDR); //累加生成的DNSsendBuff里面的数据的字节数  return iSendByte; //返回最终生成的DNS查询报文的字节数  }  } |
| **解析收到的DNS报文函数** |
| //解析收到的DNS应答报文DNSrecvBuff，应答记录字段中类型值为5即为别名则直接移动指针跳过此记录，类型值为1即为查询域名对应的IP地址信息就显示出来  void decodeDNSPacket(char \*DNSrecvBuff)  {  pDNSHDR pDnsHdr = (pDNSHDR)DNSrecvBuff; //指针pDnsHdr指向接收到的DNS应答报文首部  int iQueryNum,iRespNum,iAuthRespNum,iAddtionNum;  iQueryNum = ntohs(pDnsHdr->questNum); //将查询记录数目  iRespNum = ntohs(pDnsHdr->answerNum); // 应答记录数目  iAuthRespNum = ntohs(pDnsHdr->authorNum); // 授权服务器地址数目  iAddtionNum = ntohs(pDnsHdr->additionNum); // 附加信息数目均保存起来，本程序主要使用查询记录数目和应答记录数目  if(pDnsHdr->flags >> 15) //将DNS应答报文的“标志”字段右移15位即取最高位，为0为DNS查询报文，为1则为DNS应答报文  {    if((pDnsHdr->flags & 0x0007) == 3)  {  printf("No corresponding domain name entry.\n\n");  return;  }  if((pDnsHdr->flags >> 10) & 0x0001) //查看“标志”字段的AA位，看此应答信息是否为授权应答  {  printf("Authoritative answer:\n");  }  else  {  printf("None-authoritative answer:\n");  }  char \*pTraceResponse;  pTraceResponse=DNSrecvBuff+sizeof(DNSHDR); //指针移向应答报文中的第一个查询记录，因为一般情况下应答报文均会首先附带一个对应的查询记录  while(\*pTraceResponse) //因为查询记录的“域名”字段是变长的，以0x00结尾，所以通过移动指针将指针移到“域名”字段最后  pTraceResponse++;  pTraceResponse++; //把指针移动到查询记录的“域名”字段之后  pTraceResponse += sizeof(long); //跳过“查询类型”和“查询类”两个字段，指针指向第一个应答记录  in\_addr address;  pRESPONSE pResponse;  printf("Addresses: ");  for(int i=1;i<=iRespNum;i++)  {  pTraceResponse += sizeof(short); //指针跳过应答记录的“域名”字段，此“域名”字段一般为一个域名指针，以0xC0开始。  pResponse = (pRESPONSE)pTraceResponse;  if(ntohs(pResponse->type) == 1) //这条应答记录返回的是与之前查询所对应的IP地址  {  pTraceResponse += sizeof(RESPONSE);  unsigned long ulIP = \*(unsigned long \*)pTraceResponse;  address.s\_addr = ulIP; //获取IP地址信息保存到ulIP，并写入address里面  if(i == iRespNum) //最后一条记录显示句号，否则显示分号  {  printf("%s. ",inet\_ntoa(address));  }  else  {  printf("%s; ",inet\_ntoa(address));  }  pTraceResponse += sizeof(long); //指针移过应答记录的IP地址字段，指向下一个应答记录  }  else if (ntohs(pResponse->type) == 5) //这条应答记录为所查询主机的一个别名，这里本程序直接跳过这条记录  {  pTraceResponse += sizeof(RESPONSE);  pTraceResponse += ntohs(pResponse->length);  }  }  printf("\n\n");  }  else //“标志”字段最高位不为1，表示不是一个DNS应答报文，不做任何处理  {  printf("Invalid DNS resolution!\n\n");  }  } |
| **获取本机DNS服务器地址** |
| //此函数获取本机DNS服务器地址（为点分十进制计法的字符串形式），并保存到dnsServer中，这里需要Iphlpapi.h和IPHLPAPI.LIB  void getDnsServer(char \*dnsServer)  {  //获得需要的缓冲区大小  DWORD nLength = 0;  if (GetNetworkParams(NULL, &nLength) != ERROR\_BUFFER\_OVERFLOW)  {  return;  }  FIXED\_INFO\* pFixedInfo = (FIXED\_INFO\*)new BYTE[nLength];  //获得本地计算机网络参数  if (GetNetworkParams(pFixedInfo, &nLength) != ERROR\_SUCCESS)  {  delete[] pFixedInfo;  return;  }  IP\_ADDR\_STRING\* pCurrentDnsServer = &pFixedInfo->DnsServerList;  if(pCurrentDnsServer != NULL)  {  char \*tmp = pCurrentDnsServer->IpAddress.String; //pCurrentDnsServer->IpAddress.String即为我们所需要的字符串形式的DNS服务器IP地址  strcpy(dnsServer,tmp);  }  } |

1. 运行结果



本人运行程序时，测试了三个域名，如上，观察结果可知，程序正确解析出相应的IP地址

1. 实验总结
2. 本实验的DNS域名解析流程如下

创建socket套接字并将其与本地网卡绑定在一起

获取本机DNS服务器并作为默认的DNS服务器地址，如上图显示：

Default DNS Server：218.104.111.114

生成一个DNS的域名请求报文

将DNS讲求报文发往DNS服务器

接收DNS应答报文，并进行解析成IP地址

1. DNS域名请求报文的生成即为对DNS报文各字段进行装配，并将域名的字符串进行格式转化
2. 接收DNS回应报文，即为读取应答记录字段的类型值，当类型值为1时即为查询域名对应的IP地址。
3. 域名解析的过程中，应用进程成为DNS的一个客户，把待解析的域名放在DNS请求报文中，以UDP数据报方式发给本地域名服务器，本地域名服务器在查找域名后，将对应的IP地址放在回答报文中返回。若本地域名服务器不能回答报文，则此域名服务器就暂时成为DNS中的另一个客户，并向其他域名服务器发出查询请求，直到找到能够回答该请求域名的服务器为止。