WinPcap 教程：循序渐进教您使用 WinPcap

本节将向您展示如何使用 WinPcap API 的一些特性。 这部分教程细化成若干节课，以循序渐进的方式介绍给读者，让读者从最基本的部分(获得设备列表)到最复杂的部分(控制发送队列并收集和统计网络流量)来了解如何使用 WinPcap 进行程序开发。

有时，我们会给出一些简单使用的代码片断，但同时，我们提供完整程序的链接：所有的源代码包含一些指向手册其他地方的链接，这可以让您很方便地通过点击鼠标来跳转到您想查看的函数和数据结构的内容中去。

范例程序都是用纯 C 语言编写，所以，掌握基本的 C 语言编程知识是必须的，而且，这是一部关于处理原始网络数据包的教程，因为，我们希望读者拥有良好的网络及网络协议的知识。

用 Microsoft Visual C++ 创建一个使用 wpcap.dll 的应用程序，需要按一下步骤：

- 在每一个使用了库的源程序中，将 pcap.h 头文件包含(include)进来。
- 如果你在程序中使用了 WinPcap 中提供给 Win32 平台的特有的函数， 记得在预处理中加入 WPCAP 的定义。 （工程->设置->c/c++->预处理程序定义 中添加 WPCAP）
- 如果你的程序使用了 WinPcap 的远程捕获功能，那么在预处理定义中加入 HAVE_REMOTE。不要直接把 remote-ext.h 直接加入到你的源文件中去。 （工程->设置->c/c++->预处理程序定义 中添加 HAVE_REMOTE）
- 设置 VC++的链接器(Linker)，把 wpcap.lib 库文件包含进来。wpcap.lib 可以在 WinPcap 中找到。
- 设置 VC++的链接器(Linker)，把 ws2_32.lib 库文件包含进来。这个文件分布于 C 的编译器，并且包含了 Windows 的一些 socket 函数。本教程中的一些范例程序，会需要它。

获取设备列表

通常，编写基于 WinPcap 应用程序的第一件事情，就是获得已连接的网络适配器列表。libpcap 和 WinPcap 都提供了 **pcap_findalldevs_ex()** 函数来实现这个功能：这个函数返回一个 **pcap_if** 结构的链表， 每个这样的结构都包含了一个适配器的详细信息。值得注意的是，数据域 *name* 和 *description* 表示一个适配器名称和一个可以让人们理解的描述。

下列代码能获取适配器列表，并在屏幕上显示出来，如果没有找到适配器，将打印错误信息。

```
#include "pcap.h"


main()
{
    pcap_if_t *alldevs;
    pcap_if_t *d;
    int i=0;
    char errbuf[PCAP_ERRBUF_SIZE];


    /* 获取本地机器设备列表 */
```

```
    if (pcap_findalldevs_ex(PCAP_SRC_IF_STRING, NULL /*
auth is not needed */ , &alldevs, errbuf) == -1)
    {
```

```
        fprintf(stderr, "Error in pcap_findalldevs_ex: %s\n" ,
errbuf);
        exit(1);
    }

    /* 打印列表 */
    for(d= alldevs; d != NULL; d= d-> next)
    {
        printf( "%d. %s", ++i, d-> name);

        if (d-> description)
            printf( " (%s)\n", d->description);
        else
            printf( " (No description available)\n" );
    }

    if (i == 0)
    {
        printf( "\nNo interfaces found! Make sure WinPcap is
installed.\n");
        return;
    }

    /* 不再需要设备列表了，释放它 */
    pcap_freealldevs(alldevs);
}
```

有关这段代码的一些说明。

首先， **pcap_findalldevs_ex()** ，和其他 libpcap 函数一样，有一个 *errbuf* 参数。一旦发生错误，这个参数将会被 libpcap 写入字符串类型的错误信息。第二要记住，不是所有的操作系统都支持 libpcap 提供的网络程序接口，因此，如果我们想编写一个可移植的应用程序，我们就必须考虑在什么情况下， *description* 是 null。本程序中，我们遇到这种情况时，会打印提示语句"No description available"。

最后要记住，当我们完成了设备列表的使用，我们要调用 pcap_freealldevs() 函数将其占用的内存资源释放。

让我们编译并运行我们的第一个示例程序吧！ 为了能在 Unix 或 Cywin 平台上编译这段程序，需要简单输入：

```
gcc -o testprog testprog.c -lpcap
```

在 Windows 平台上，您需要创建一个工程，并按照 使用 **WinPcap** 编程 里的步骤做。 然而，我们建议您使用 WinPcap developer's pack (详情请访问 WinPcap 网站, http://www.winpcap.org), 因为它提供了很多已经配置好的范例，包括本教程中的所有示例代码，以及在编译运行时需要的 *包含文件(include)* 和 *库(libraries)*

假设我们已经完成了对程序的编译，那让我们来运行它吧。在某台 WinXP 的电脑上，我们得到的结果是：

1. \Device\NPF_{4E273621-5161-46C8-895A-48D0E52A0B83} (Realtek RTL8029(AS) Ethernet Adapter)

2. \Device\NPF_{5D24AE04-C486-4A96-83FB-8B5EC6C7F430} (3Com EtherLink PCI)

正如您看到的，Windows 平台下的网络适配器的名字读起来相当容易，可见，解释性的描述是多么有帮助阿！

获取已安装设备的高级信息

在第 1 讲中，（获取设备列表）我们展示了如何获取适配器的基本信息（如设备的名称和描述）。事实上，WinPcap 提供了其他更高级的信息。特别需要指出的是，由 **pcap_findalldevs_ex()** 返回的每一个 **pcap_if** 结构体，都包含一个 **pcap_addr** 结构体，这个结构体由如下元素组成：

- 一个地址列表
- 一个掩码列表（each of which corresponds to an entry in the addresses list).
- 一个广播地址列表（each of which corresponds to an entry in the addresses list).
- 一个目的地址列表（each of which corresponds to an entry in the addresses list).

另外，函数 pcap_findalldevs_ex() 还能返回远程适配器信息和一个位于所给的本地文件夹的 pcap 文件列表。

下面的范例使用了 ifprint()函数来打印出 pcap_if 结构体中所有的内容。程序对每一个由 pcap_findalldevs_ex() 函数返回的 pcap_if，都调用 ifprint()函数来实现打印。

```
#include "pcap.h"


#ifndef WIN32
   #include <sys/socket.h>
   #include <netinet/in.h>
#else
   #include <winsock.h>
#endif
```

```
// 函数原型
void ifprint(pcap_if_t *d);
char *iptos(u_long in);
char* ip6tos(struct sockaddr *sockaddr, char *address, int addrlen);


int main()
{
  pcap_if_t *alldevs;
  pcap_if_t *d;
  char errbuf[PCAP_ERRBUF_SIZE+1];
  char source[PCAP_ERRBUF_SIZE+1];

  printf("Enter the device you want to list:\n"
            "rpcap://           ==> lists interfaces in the local machine\n"
            "rpcap://hostname:port ==> lists interfaces in a remote machine\n"
```

```
            "                    (rpcapd daemon must be up and
running\n"
            "                    and it must accept 'null'
authentication)\n"
            "file://foldername    ==> lists all pcap files in the give
folder\n\n"
            "Enter your choice: " );

  fgets(source,  PCAP_ERRBUF_SIZE, stdin);
  source[ PCAP_ERRBUF_SIZE] = '\0';

  /* 获得接口列表 */
  if (pcap_findalldevs_ex(source, NULL, &alldevs, errbuf)
== -1)
  {
    fprintf(stderr, "Error in pcap_findalldevs: %s\n" ,errbuf);
    exit(1);
  }

  /* 扫描列表并打印每一项 */
  for(d=alldevs;d;d=d->next)
  {
    ifprint(d);
  }
  pcap_freealldevs(alldevs);
  return 1;
}

/* 打印所有可用信息 */
void ifprint(pcap_if_t *d)
{
  pcap_addr_t *a;
  char ip6str[128];

  /* 设备名(Name) */
  printf("%s\n",d->name);

  /* 设备描述(Description) */
  if (d->description)
    printf( "\tDescription: %s\n",d->description);

  /* Loopback Address*/
  printf("\tLoopback: %s\n",(d->flags &
PCAP_IF_LOOPBACK)?"yes":"no");
```

```
  /* IP addresses */
  for(a=d->addresses;a;a=a->next) {
    printf( "\tAddress Family: #%d\n",a->addr->sa_family);

    switch(a->addr->sa_family)
    {
      case AF_INET:
       printf( "\tAddress Family Name: AF_INET\n" );
        if (a->addr)
         printf( "\tAddress: %s\n",iptos(((struct sockaddr_in
*)a->addr)->sin_addr.s_addr));
        if (a->netmask)
         printf( "\tNetmask: %s\n",iptos(((struct sockaddr_in
*)a->netmask)->sin_addr.s_addr));
        if (a->broadaddr)
         printf( "\tBroadcast Address: %s\n",iptos(((struct
sockaddr_in *)a->broadaddr)->sin_addr.s_addr));
        if (a->dstaddr)
         printf( "\tDestination Address: %s\n",iptos(((struct
sockaddr_in *)a->dstaddr)->sin_addr.s_addr));
        break;

      case AF_INET6:
       printf( "\tAddress Family Name: AF_INET6\n" );
        if (a->addr)
         printf( "\tAddress: %s\n", ip6tos(a->addr, ip6str,
sizeof(ip6str)));
       break;

      default:
       printf( "\tAddress Family Name: Unknown\n" );
        break;
    }
  }
  printf("\n");
}

/* 将数字类型的 IP 地址转换成字符串类型的 */

#define IPTOSBUFFERS    12
char *iptos(u_long in)
{
    static char output[IPTOSBUFFERS][3*4+3+1];
    static short which;

    u_char *p;
```

```
    p = (u_char *)&in;
    which = (which + 1 == IPTOSBUFFERS ? 0 : which + 1);
    sprintf(output[which], "%d.%d.%d.%d", p[0], p[1], p[2],
p[3]);
    return output[which];
```

```
}

char* ip6tos(struct sockaddr *sockaddr, char *address, int
addrlen)
{

    socklen_t sockaddrlen;

   #ifdef WIN32
    sockaddrlen =  sizeof(struct sockaddr_in6);
   #else
    sockaddrlen =  sizeof(struct sockaddr_storage);
   #endif


    if(getnameinfo(sockaddr,
        sockaddrlen,
       address,
       addrlen,
       NULL,
       0,
        NI_NUMERICHOST) != 0) address = NULL;
    return address;
}
```

打开适配器并捕获数据包

现在，我们已经知道如何获取适配器的信息了，那我们就开始一项更具意义的工作，打开适配器并捕获数据包。在这讲中，我们会编写一个程序，将每一个通过适配器的数据包打印出来。

打开设备的函数是 **pcap_open()**。下面是参数 *snaplen*, *flags* 和 *to_ms* 的解释说明

snaplen 制定要捕获数据包中的哪些部分。 在一些操作系统中 (比如 xBSD 和 Win32)， 驱动可以被配置成只捕获数据包的初始化部分： 这样可以减少应用程序间复制数据的量，从而提高捕获效率。本例中，我们将值定为 65535，它比我们能遇到的最大的 MTU 还要大。因此，我们确信我们总能收到完整的数据包。

flags: 最最重要的 flag 是用来指示适配器是否要被设置成混杂模式。 一般情况下，适配器只接收发给它自己的数据包， 而那些在其他机器之间通讯的数据包，将会被丢弃。 相反，如果适配器是混杂模式，那么不管这个数据包是不是发给我的，我都会去捕获。也就是说，我会去捕获所有的数据包。这意味着在一个共享媒介(比如总线型以太网)，WinPcap 能捕获其他主机的所有的数据包。 大多数

用于数据捕获的应用程序都会将适配器设置成混杂模式，所以，我们也会在下面的范例中，使用混杂模式。

to_ms 指定读取数据的超时时间，以毫秒计(1s=1000ms)。在适配器上进行读取操作(比如用 **pcap_dispatch()** 或 **pcap_next_ex()**) 都会在 *to_ms* 毫秒时间内响应，即使在网络上没有可用的数据包。 在统计模式下，*to_ms* 还可以用来定义统计的时间间隔。 将 *to_ms* 设置为 0 意味着没有超时，那么如果没有数据包到达的话，读操作将永远不会返回。 如果设置成-1，则情况恰好相反，无论有没有数据包到达，读操作都会立即返回。

```
#include "pcap.h"

/* packet handler 函数原型 */
void packet_handler(u_char *param, const struct pcap_pkthdr
*header, const u_char *pkt_data);

main()
{
pcap_if_t *alldevs;
pcap_if_t *d;
int inum;
int i=0;
pcap_t *adhandle;
char errbuf[PCAP_ERRBUF_SIZE];

    /* 获取本机设备列表 */
    if (pcap_findalldevs_ex(PCAP_SRC_IF_STRING, NULL,
&alldevs, errbuf) == -1)
   {
       fprintf(stderr, "Error in pcap_findalldevs: %s\n" ,
errbuf);
       exit(1);
   }

    /* 打印列表 */
    for(d=alldevs; d; d=d->next)
   {
       printf( "%d. %s", ++i, d->name);
       if (d-> description)
          printf( " (%s)\n", d->description);
       else
          printf( " (No description available)\n" );
```

```
    }

    if(i==0)
    {
        printf( "\nNo interfaces found! Make sure WinPcap is
installed.\n");
        return -1;
    }

    printf( "Enter the interface number (1-%d):" ,i);
    scanf( "%d", &inum);

    if(inum < 1 || inum > i)
    {
        printf( "\nInterface number out of range.\n" );
        /* 释放设备列表 */
        pcap_freealldevs(alldevs);
        return -1;
    }

    /* 跳转到选中的适配器 */
    for(d=alldevs, i=0; i< inum-1 ;d=d-> next, i++);

    /* 打开设备 */
    if ( (adhandle= pcap_open(d->name,          // 设备名
                        65536,                  // 65535 保证能捕获
到不同数据链路层上的每个数据包的全部内容
PCAP_OPENFLAG_PROMISCUOUS,    // 混杂模式
                        1000,                   // 读取超时时间
                        NULL,                   // 远程机器验证
                        errbuf                  // 错误缓冲池
                        ) ) == NULL)
    {
        fprintf(stderr, "\nUnable to open the adapter. %s is not
supported by WinPcap\n", d->name);
        /* 释放设备列表 */
        pcap_freealldevs(alldevs);
        return -1;
    }

    printf( "\nlistening on %s...\n", d->description);

    /* 释放设备列表 */
    pcap_freealldevs(alldevs);
```

```
    /* 开始捕获 */
    pcap_loop(adhandle, 0, packet_handler, NULL);

    return 0;
}


/* 每次捕获到数据包时，libpcap 都会自动调用这个回调函数 */
void packet_handler(u_char *param, const struct pcap_pkthdr
*header, const u_char *pkt_data)
{
    struct tm *ltime;
    char timestr[16];
    time_t local_tv_sec;

    /* 将时间戳转换成可识别的格式 */
    local_tv_sec = header-> ts.tv_sec;
    ltime=localtime(&local_tv_sec);
    strftime( timestr,  sizeof timestr, "%H:%M:%S", ltime);

   printf( "%s,%.6d len:%d\n", timestr, header-> ts.tv_usec,
header->len);

}
```

当适配器被打开，捕获工作就可以用 **pcap_dispatch()** 或 **pcap_loop()**进行。 这两个函数非常的相似，区别就是 pcap_ dispatch() 当超时时间到了(timeout expires)就返回 (尽管不能保证)，而 **pcap_loop()** 不会因此而返回，只有当 *cnt* 数据包被捕获，所以，pcap_loop()会在一小段时间内，阻塞网络的利用。**pcap_loop()**对于我们这个简单的范例来说，可以满足需求，不过， **pcap_dispatch()** 函数一般用于比较复杂的程序中。

这两个函数都有一个 *回调* 参数， *packet_handler* 指向一个可以接收数据包的函数。 这个函数会在收到每个新的数据包并收到一个通用状态时被 libpcap 所调用 (与函数 **pcap_loop()** 和 **pcap_dispatch()** 中的 *user* 参数相似)，数据包的首部一般有一些诸如时间戳，数据包长度的信息，还有包含了协议首部的实际数据。 注意：冗余校验码 CRC 不再支持，因为帧到达适配器，并经过校验确认以后，适配器就会将 CRC 删除，与此同时，大部分适配器会直接丢弃 CRC 错误的数据包，所以，WinPcap 没法捕获到它们。

上面的程序将每一个数据包的时间戳和长度从 **pcap_pkthdr** 的首部解析出来，并打印在屏幕上。

请注意，使用 **pcap_loop()** 函数可能会遇到障碍，主要因为它直接由数据包捕获驱动所调用。因此，用户程序是不能直接控制它的。另一个实现方法(也是提高可读性的方法)，是使用 **pcap_next_ex()** 函数。有关这个函数的使用，我们将在下一讲为您展示。 (不用回调方法捕获数据包).

不用回调方法捕获数据包

本讲的范例程序所实现的功能和效果和上一讲的非常相似（打开适配器并捕获数据包），但本讲将用 **pcap_next_ex()** 函数代替上一讲的 **pcap_loop()**函数。

pcap_loop()函数是基于回调的原理来进行数据捕获，这是一种精妙的方法，并且在某些场合中，它是一种很好的选择。 然而，处理回调有时候并不实用 -- 它会增加程序的复杂度，特别是在拥有多线程的 C++程序中。

可以通过直接调用 **pcap_next_ex()** 函数来获得一个数据包 -- 只有当编程人员使用了 **pcap_next_ex()** 函数才能收到数据包。

这个函数的参数和捕获回调函数的参数是一样的 -- 它包含一个网络适配器的描述符和两个可以初始化和返回给用户的指针（一个指向 **pcap_pkthdr** 结构体，另一个指向数据报数据的缓冲）。

在下面的程序中，我们会再次用到上一讲中的有关回调方面的代码，只是我们将它放入了 main()函数，之后调用 **pcap_next_ex()**函数。

```
#include "pcap.h"

main()

{
pcap_if_t *alldevs;

pcap_if_t *d;
int inum;
int i=0;
```

```
pcap_t *adhandle;
int res;
char errbuf[PCAP_ERRBUF_SIZE];
struct tm *ltime;
char timestr[16];
struct pcap_pkthdr *header;
const u_char *pkt_data;
time_t local_tv_sec;

    /* 获取本机设备列表 */
    if (pcap_findalldevs_ex(PCAP_SRC_IF_STRING, NULL,
&alldevs, errbuf) == -1)
    {
        fprintf(stderr, "Error in pcap_findalldevs: %s\n" , errbuf);

        exit(1);
    }

    /* 打印列表 */
    for(d=alldevs; d; d=d->next)
    {
        printf( "%d. %s", ++i, d->name);
        if (d-> description)
            printf( " (%s)\n", d->description);
         else
```

```
            printf(   " (No description available)\n" );
    }

    if(i==0)
    {
        printf(  "\nNo interfaces found! Make sure WinPcap is installed.\n");
        return -1;
    }

    printf( "Enter the interface number (1-%d):" ,i);
    scanf( "%d", &inum);

    if(inum < 1 || inum > i)
    {
        printf(  "\nInterface number out of range.\n" );
        /* 释放设备列表 */
        pcap_freealldevs(alldevs);

        return -1;
    }

    /* 跳转到已选中的适配器 */
    for(d=alldevs, i=0; i< inum-1 ;d=d-> next, i++);
        /* 打开设备 */
    if ( (adhandle= pcap_open(d->name,          // 设备名
                        65536,                  // 要捕捉的数据包的部分
                                                // 65535 保证能捕获到不同数据
链路层上的每个数据包的全部内容
                            PCAP_OPENFLAG_PROMISCUOUS,    // 混
杂模式
                        1000,                   // 读取超时时间
                        NULL,                   // 远程机器验证
                        errbuf                  // 错误缓冲池
                        ) ) == NULL)

    {
        fprintf(stderr, "\nUnable to open the adapter. %s is not supported
by WinPcap\n", d->name);
        /* 释放设列表 */
        pcap_freealldevs(alldevs);
        return -1;
    }

    printf( "\nlistening on %s...\n", d->description);

    /* 释放设备列表 */
```

```
    pcap_freealldevs(alldevs);

    /* 获取数据包 */
    while((res = pcap_next_ex( adhandle, &header, &pkt_data)) >=
0){
        ____________________

        if(res == 0)
```

```
            /* 超时时间到 */
            continue;

        /* 将时间戳转换成可识别的格式 */
        local_tv_sec = header->  ts.tv_sec;
        ltime=localtime(&local_tv_sec);
        strftime( timestr,   sizeof timestr, "%H:%M:%S", ltime);

       printf( "%s,%.6d len:%d\n", timestr, header->ts.tv_usec,
header->len);
    }

    if(res == -1){
       printf( "Error reading the packets: %s\n" ,
pcap_geterr(adhandle));
        return -1;
    }
    return 0;
}
```

为什么我们要用 **pcap_next_ex()** 代替以前的 **pcap_next()**？ 因为 **pcap_next()** 有一些不好的地方。首先，它效率低下，尽管它隐藏了回调的方式，但它依然依赖于函数 **pcap_dispatch()**。第二，它不能检测到文件末尾这个状态(EOF)，因此，如果数据包是从文件读取来的，那么它就不那么有用了。

值得注意的是， **pcap_next_ex()** 在成功，超时，出错或 EOF 的情况下，会返回不同的值。

过滤数据包

WinPcap 和 Libpcap 的最强大的特性之一，是拥有过滤数据包的引擎。 它提供了有效的方法去获取网络中的某些数据包，这也是 WinPcap 捕获机制中的一个组成部分。 用来过滤数据包的函数是 **pcap_compile()** 和 **pcap_setfilter()** 。

pcap_compile() 它将一个高层的布尔过滤表达式编译成一个能够被过滤引擎所解释的低层的字节码。有关布尔过滤表达式的语法可以参见 **Filtering expression syntax** 这一节的内容。

pcap_setfilter() 将一个过滤器与内核捕获会话向关联。当 **pcap_setfilter()** 被调用时，这个过滤器将被应用到来自网络的所有数据包，并且，所有的符合要求的数据包 (即那些经过过滤器以后，布尔表达式为真的包) ，将会立即复制给应用程序。

以下代码展示了如何编译并设置过滤器。 请注意，我们必须从 **pcap_if** 结构体中获得掩码，因为一些使用 **pcap_compile()** 创建的过滤器需要它。

在这段代码片断中，传递给 **pcap_compile()** 的过滤器是"ip and tcp"，这说明我们只希望保留 IPv4 和 TCP 的数据包，并把他们发送给应用程序。

```
    if (d->addresses != NULL)
        /* 获取接口第一个地址的掩码 */
       netmask=(( struct sockaddr_in
*)(d->addresses->netmask))->sin_addr.S_un.S_addr;
    else
        /* 如果这个接口没有地址，那么我们假设这个接口在 C 类网络中 */
       netmask=0xffffff;


compile the filter
   if ( pcap_compile(adhandle, &fcode, "ip and tcp", 1,
netmask) < 0)
   {
       fprintf(stderr, "\nUnable to compile the packet filter. Check
the syntax.\n");
        /* 释放设备列表 */
        pcap_freealldevs(alldevs);
        return -1;
   }

set the filter
    if (pcap_setfilter(adhandle, &fcode) < 0)

   {
       fprintf(stderr, "\nError setting the filter.\n" );
        /* 释放设备列表 */
        pcap_freealldevs(alldevs);
        return -1;
   }
```

如果你想阅读过滤函数的范例代码，请阅读下一讲 分析数据包的例子。

本讲中，我们将会利用上一讲的一些代码，来建立一个更实用的程序。 本程序的主要目标是展示如何解析所捕获的数据包的协议首部。这个程序可以称为 UDPdump，打印一些网络上传输的 UDP 数据的信息。

我们选择分析和现实 UDP 协议而不是 TCP 等其它协议，是因为它比其它的协议更简单，作为一个入

我们选择分析和实现 UDP 协议而不是 TCP 等其它协议，是因为它比其它的协议更简单，作为一个入门程序范例，是很不错的选择。让我们看看代码：

```
#include "pcap.h"

/* 4 字节的 IP 地址 */
typedef struct ip_address{
    u_char byte1;
    u_char byte2;
    u_char byte3;
    u_char byte4;
}ip_address;

/* IPv4 首部 */
typedef struct ip_header{
    u_char  ver_ihl;          // 版本 (4 bits) + 首部长度 (4 bits)
    u_char  tos;              // 服务类型(Type of service)
    u_short tlen;             // 总长(Total length)
    u_short identification;   // 标识(Identification)
    u_short flags_fo;         // 标志位(Flags) (3 bits) + 段偏移量(Fragment offset) (13 bits)
    u_char  ttl;              // 存活时间(Time to live)
    u_char  proto;            // 协议(Protocol)
    u_short crc;              // 首部校验和(Header checksum)
    ip_address  saddr;        // 源地址(Source address)
    ip_address  daddr;        // 目的地址(Destination address)
    u_int   op_pad;           // 选项与填充(Option + Padding)
}ip_header;

/* UDP 首部*/
typedef struct udp_header{
    u_short sport;            // 源端口(Source port)
    u_short dport;            // 目的端口(Destination port)
    u_short len;              // UDP 数据包长度(Datagram length)
    u_short crc;              // 校验和(Checksum)
}udp_header;

/* 回调函数原型 */
void packet_handler(u_char *param, const struct pcap_pkthdr *header, const u_char *pkt_data);

main()
{
pcap_if_t *alldevs;
pcap_if_t *d;
```

```
int inum;
int i=0;
pcap_t *adhandle;
char errbuf[PCAP_ERRBUF_SIZE];
u_int netmask;
char packet_filter[] = "ip and udp";
struct bpf_program fcode;

    /* 获得设备列表 */
    if (pcap_findalldevs_ex(PCAP_SRC_IF_STRING, NULL,
&alldevs, errbuf) == -1)
    {
        fprintf(stderr, "Error in pcap_findalldevs: %s\n", errbuf);
        exit(1);
    }

    /* 打印列表 */
    for(d=alldevs; d; d=d->next)
    {
        printf( "%d. %s", ++i, d->name);
        if (d-> description)
            printf( " (%s)\n", d->description);
        else
            printf( " (No description available)\n" );
    }

    if(i==0)
    {
        printf( "\nNo interfaces found! Make sure WinPcap is installed.\n");
        return -1;
    }

    printf( "Enter the interface number (1-%d):" ,i);
    scanf( "%d", &inum);

    if(inum < 1 || inum > i)
    {
        printf( "\nInterface number out of range.\n" );
        /* 释放设备列表 */
        pcap_freealldevs(alldevs);
        return -1;
    }

    /* 跳转到已选设备 */
```

```
    for(d=alldevs, i=0; i< inum-1 ;d=d-> next, i++);

    /* 打开适配器 */
    if ( (adhandle= pcap_open(d->name,  // 设备名
                        65536,          // 要捕捉的数据包的部分
                                        // 65535 保证能捕获到不同数据链路层上的
每个数据包的全部内容
                            PCAP_OPENFLAG_PROMISCUOUS,
// 混杂模式
                        1000,           // 读取超时时间
                        NULL,           // 远程机器验证
                        errbuf          // 错误缓冲池
                        ) ) == NULL)
    {
        fprintf(stderr, "\nUnable to open the adapter. %s is not supported
by WinPcap\n");
        /* 释放设备列表 */
        pcap_freealldevs(alldevs);
        return -1;
    }

    /* 检查数据链路层，为了简单，我们只考虑以太网 */
    if(pcap_datalink(adhandle) != DLT_EN10MB)
    {
        fprintf(stderr, "\nThis program works only on Ethernet
networks.\n");
        /* 释放设备列表 */
        pcap_freealldevs(alldevs);
        return -1;
    }

    if(d->addresses != NULL)
        /* 获得接口第一个地址的掩码 */
        netmask=(( struct sockaddr_in
*)(d->addresses->netmask))->sin_addr.S_un.S_addr;
    else
        /* 如果接口没有地址，那么我们假设一个 C 类的掩码 */
        netmask=0xffffff;

    //编译过滤器
    if ( pcap_compile(adhandle, &fcode, packet_filter, 1,
netmask) <0 )
    {
        fprintf(stderr, "\nUnable to compile the packet filter. Check the
syntax.\n");
```

```
        /* 释放设备列表 */
        pcap_freealldevs(alldevs);

        return -1;
    }

    //设置过滤器
    if (pcap_setfilter(adhandle, &fcode)<0)

    {
        fprintf(stderr, "\nError setting the filter.\n");
        /* 释放设备列表 */
        pcap_freealldevs(alldevs);
        return -1;
    }

    printf("\nlistening on %s...\n", d->description);
```

```
    /* 释放设备列表 */
    pcap_freealldevs(alldevs);


    /* 开始捕捉 */
    pcap_loop(adhandle, 0, packet_handler, NULL);


    return 0;
}

/* 回调函数，当收到每一个数据包时会被 libpcap 所调用 */
void packet_handler(u_char *param, const struct pcap_pkthdr
*header, const u_char *pkt_data)
{
    struct tm *ltime;
    char timestr[16];
    ip_header *ih;
    udp_header *uh;
    u_int ip_len;
    u_short sport,dport;
    time_t local_tv_sec;

    /* 将时间戳转换成可识别的格式 */
    local_tv_sec = header-> ts.tv_sec;
    ltime=localtime(&local_tv_sec);

    strftime( timestr, sizeof timestr, "%H:%M:%S", ltime);

    /* 打印数据包的时间戳和长度 */
    printf("%s.%.6d len:%d ", timestr, header->ts.tv_usec,
header->len);
```

```
    /* 获得 IP 数据包头部的位置 */
    ih = (ip_header *) (pkt_data +
        14);    //以太网头部长度

    /* 获得 UDP 首部的位置 */
    ip_len = (ih->ver_ihl & 0xf) * 4;
```

```
    uh = (udp_header *) ((u_char*)ih + ip_len);

    /* 将网络字节序列转换成主机字节序列 */
    sport = ntohs( uh->sport );
    dport = ntohs( uh->dport );

    /* 打印 IP 地址和 UDP 端口 */
    printf( "%d.%d.%d.%d.%d -> %d.%d.%d.%d.%d\n",
        ih->saddr.byte1,
        ih->saddr.byte2,
        ih->saddr.byte3,
        ih->saddr.byte4,
        sport,
        ih->daddr.byte1,
        ih->daddr.byte2,
        ih->daddr.byte3,
        ih->daddr.byte4,
        dport);
}
```

首先，我们将过滤器设置成"ip and udp"。在这种方式下，我们确信 packet_handler()只会收到基于 IPv4 的 UDP 数据包；这将简化解析过程，提高程序的效率。

我们还分别创建了用于描述 IP 首部和 UDP 首部的结构体。这些结构体中的各种数据会被 packet_handler()合理地定位。

packet_handler()，尽管只受限于单个协议的解析（比如基于 IPv4 的 UDP），不过它展示了捕捉器(sniffers)是多么的复杂，就像 TcpDump 或 WinDump 对网络数据流进行解码那样。 因为我们对 MAC 首部不感兴趣，所以我们跳过它。为了简洁，我们在开始捕捉前，使用了 **pcap_datalink()** 对 MAC 层进行了检测，以确保我们是在处理一个以太网络。这样，我们就能确保 MAC 首部是 14 位的。

IP 数据包的首部就位于 MAC 首部的后面。我们将从 IP 数据包的首部解析到源 IP 地址和目的 IP 地址。

处理 UDP 的首部有一些复杂，因为 IP 数据包的首部的长度并不是固定的。然而，我们可以通过 IP 数据包的 length 域来得到它的长度。一旦我们知道了 UDP 首部的位置，我们就能解析到源端口和目的端口。

被解析出来的值被打印在屏幕上，形式如下所示：

```
1. \Device\Packet_{A7FD048A-5D4B-478E-B3C1-34401AC3B72F} (Xircom t 10/100
Adapter)
Enter the interface number (1-2):1

listening on Xircom CardBus Ethernet 10/100 Adapter...
16:13:15.312784 len:87 130.192.31.67.2682 -> 130.192.3.21.53
16:13:15.314796 len:137 130.192.3.21.53 -> 130.192.31.67.2682

16:13:15.322101 len:78 130.192.31.67.2683 -> 130.192.3.21.53
```

最后 3 行中的每一行，分别代表了一个数据包。