

使用WinPcap

- 
- **Winpcap介绍**
 - **Winpcap安装**
 - **Winpcap应用程序结构**

什么是 Winpcap?

Libpcap (UNIX) 库 → Winpcap (Windows) 库

网络数据包捕获库函数

直接访问网络，免费、公用

工作于驱动层，网络操作高效

为应用程序提供了一组API接口

编程容易，源码级移植方便

WinPcap主要功能

- 捕获原始数据包
- 将数据包发送给应用程序之前，按照用户规定的规则过滤数据包
- 将捕获到的数据包输出到文件中，并可以对这些文件进行再分析
- 向网络发送原始数据包
- 搜集网络传输统计数据

哪些应用适合使用 WinPcap

- 网络协议分析
- 网络监控
- 流量记录统计
- 网络入侵检测
- 网络扫描
- 安全工具

WinPcap不能胜任的事情

WinPcap独立于主机的协议（如TCP/IP）收发数据包。这意味着它不能阻塞、过滤或者处理同一主机上其他程序产生的数据包：它仅仅嗅探网线上传输的数据包。所以它不适合应用于流量均衡、QoS调度和个人防火墙。

Winpcap的安装

- 下载安装包和开发包

<http://www.winpcap.org/archive/>

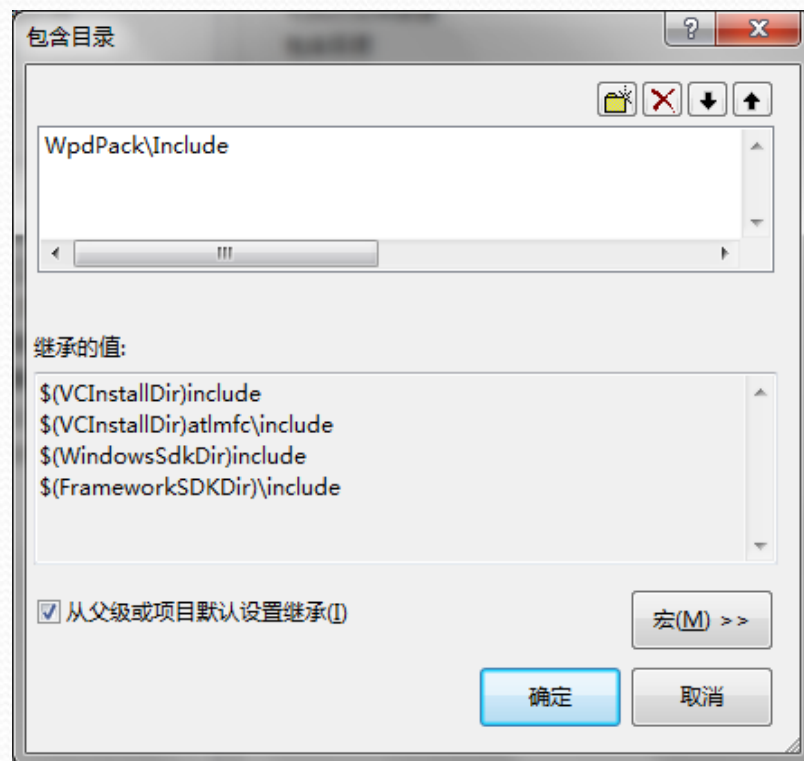
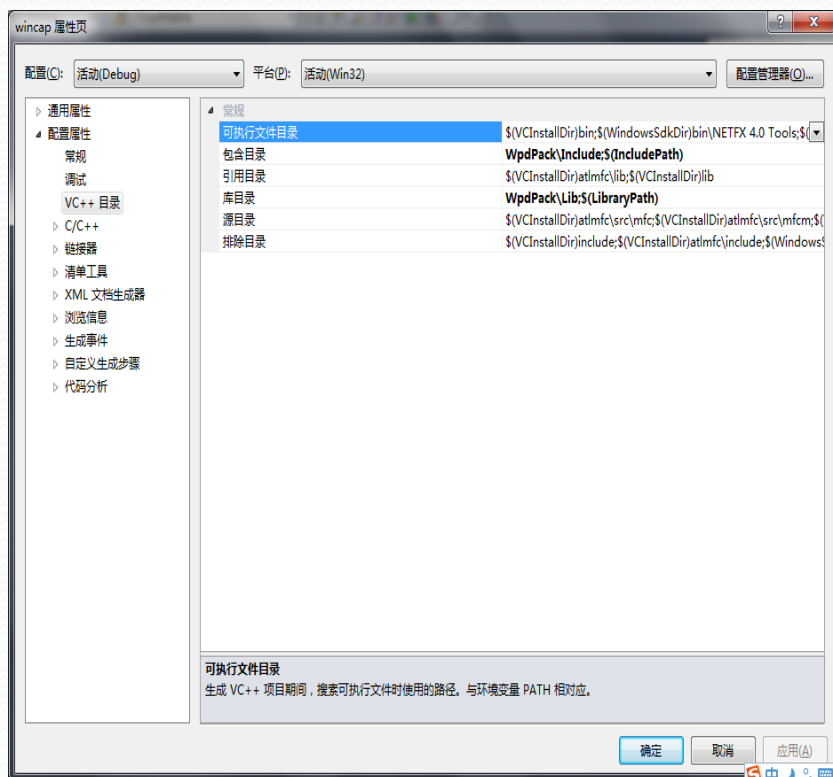
- **Winpcap**的安装包(*Winpcap_4_1.exe*)
 - 程序员开发包(*WpdPack_4_1.zip*)
-
- 运行**Winpcap_4_1.exe**
 - 测试安装结果

编程环境设定 (VC++6.0)

1. 以 Administrator 身份登录 Windows (2000/XP) , 运行一次 Winpcap 自带例程, 此后可以一般用户身份使用
2. 运行 Visual C++ 6.0,
打开 WpdPack_4_1\WpdPack\Examples-pcap\下的任一项目
3. (假设用 basic_dump 目录下 basic_dump.dsw)
 - 在 “工程->设置-> Link->对象/库模块” 中加入
`wsock32.lib ws2_32.lib wpcap.lib`
 - 在 “工具->选择->目录” 的 include files 和 library files 设置中
引入 winpcap 开发包中的 **Include** 和 **Lib** 目录
4. 编译, 运行

编程环境设定 (VS2010)

1. 在项目选项卡中点击属性->点击配置属性中的VC++目录选项
 - 在包含目录和库目录中添加相应的Include的文件夹和Lib文件夹

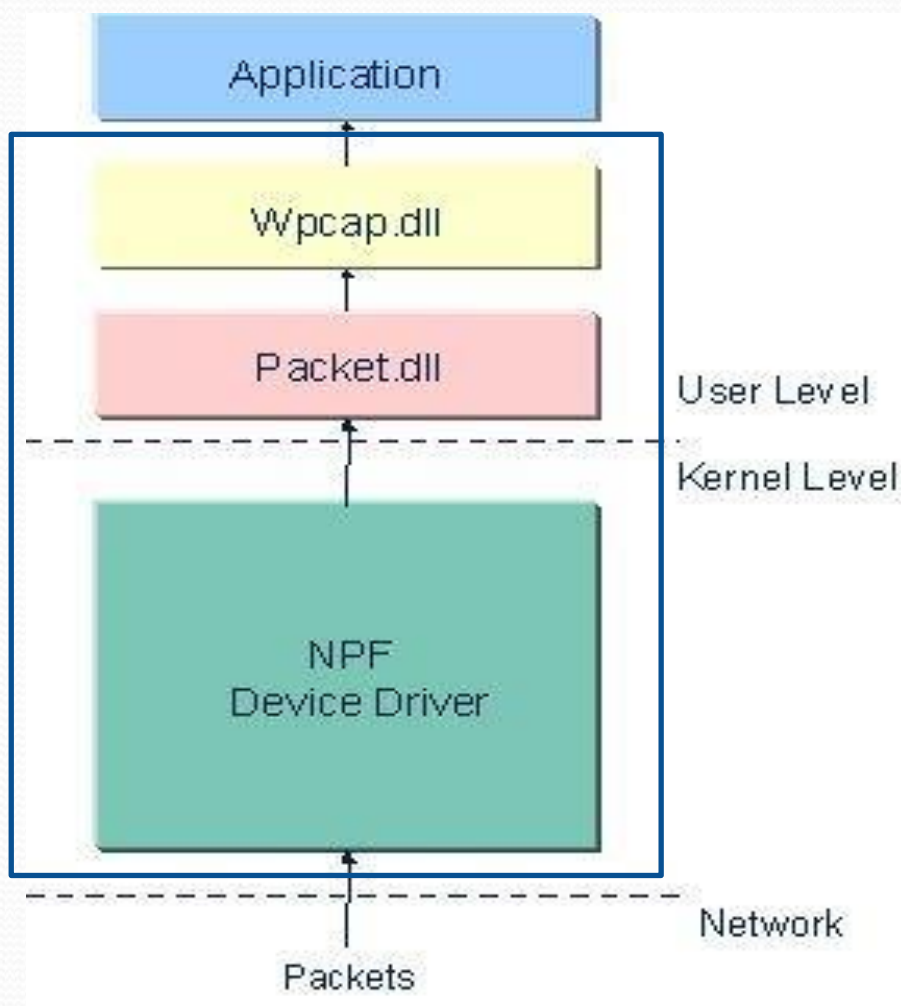


编程环境设定 (VS2010)

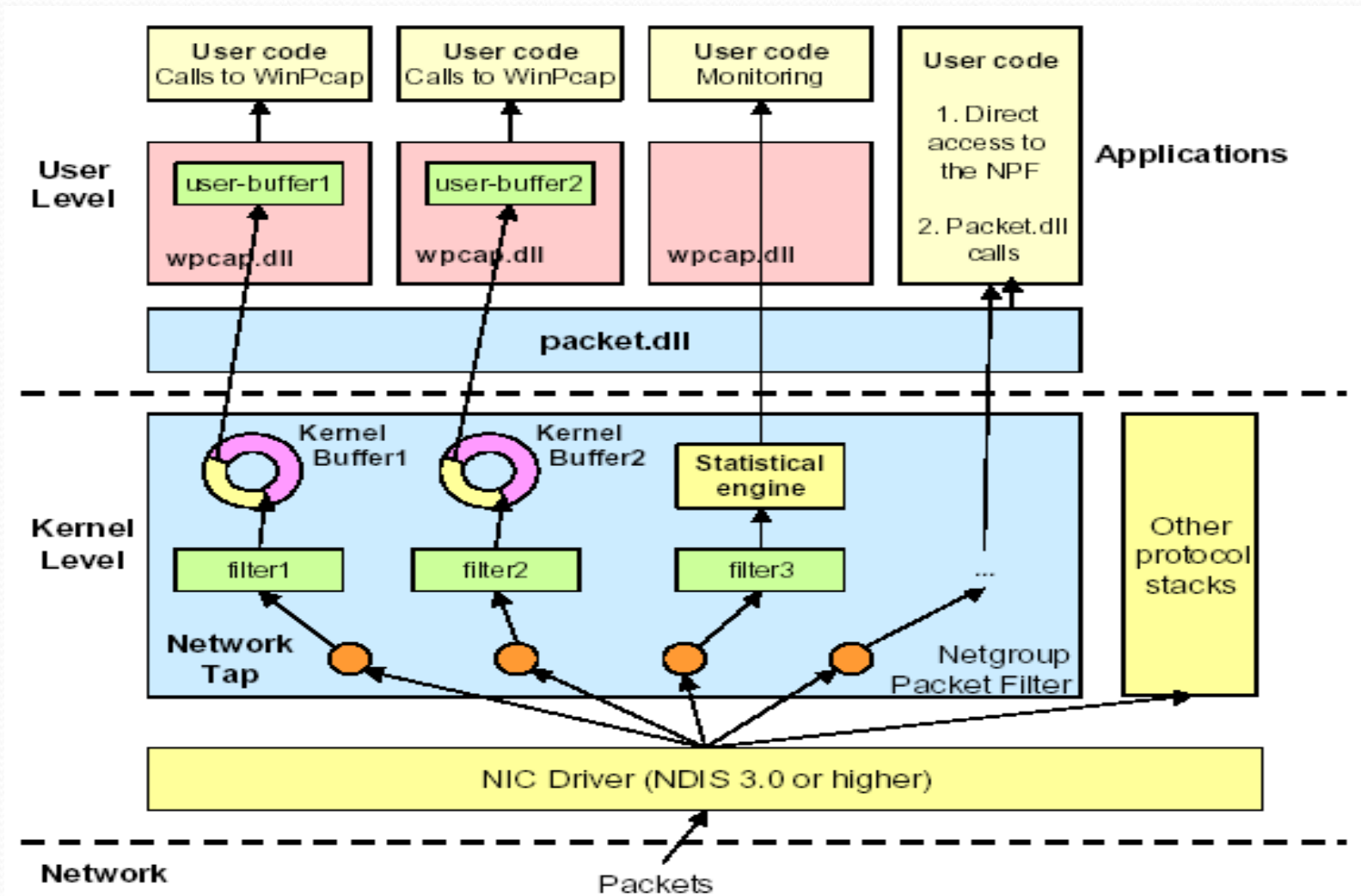
2. 代码中添加 `#pragma comment(lib, "wpcap.lib")` 即可

```
#include "stdafx.h"  
#include "pcap.h"  
#include "inc.h"  
#include "windows.h"  
  
#pragma comment(lib, "wpcap.lib")  
#pragma comment(lib, "ws2_32")  
...
```

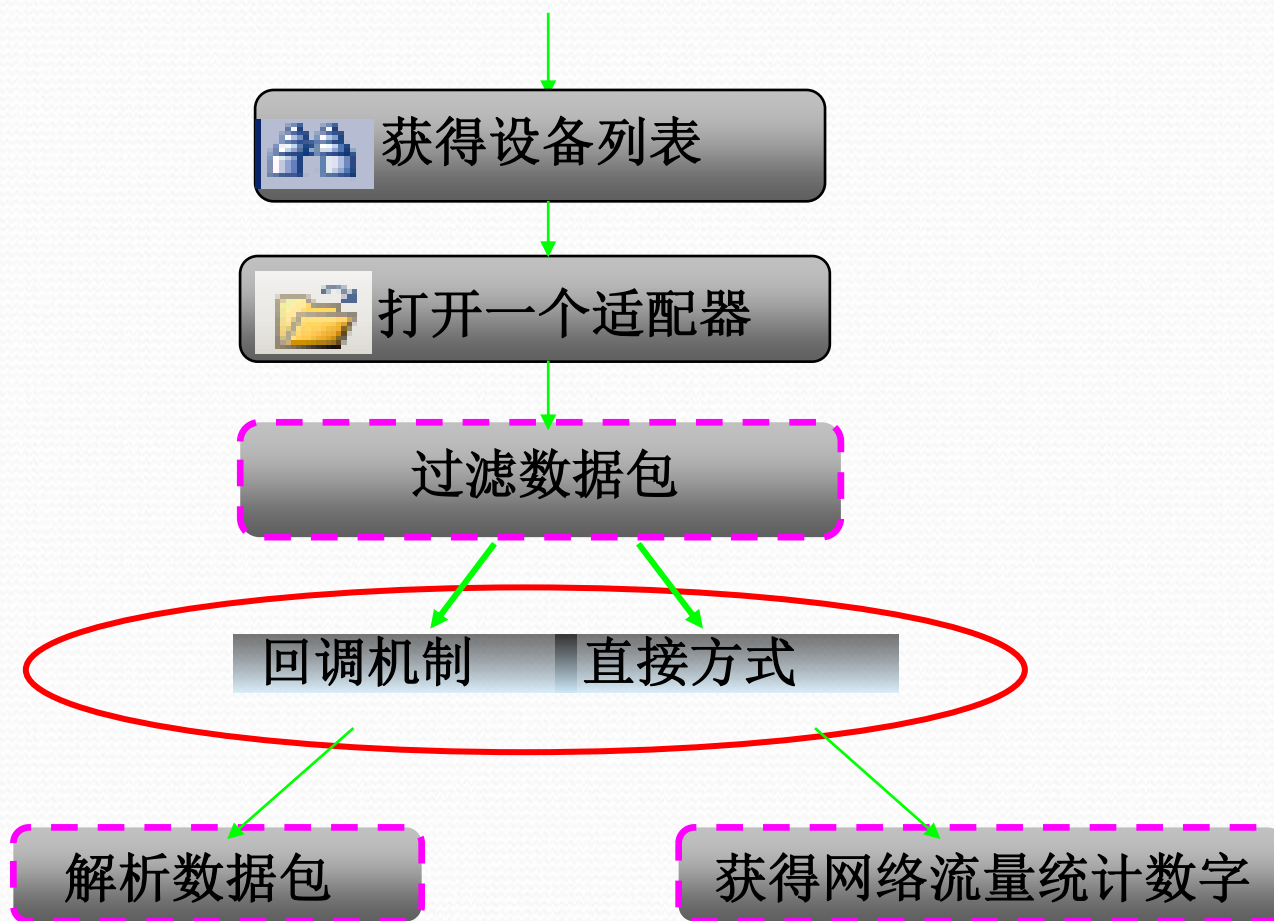
Winpcap的各个组成部分



Winpcap体系结构



WinPcap的典型应用流程（接收）



获得设备列表（一）

一个基本的WinPcap应用程序所需的第一步就是获得合适的网络适配器。

完成获得设备功能的函数：

- `pcap_findalldevs()`
- `pcap_findalldevs_ex()`

这2个函数都会返回一个的`pcap_if_t`结构的列表，列表的每一项包含关于适配器的复杂的信息。特别是`name`和`description`域数据包含设备的名称和可读的描述。

- **pcap_findalldevs()函数原型:**

int pcap_findalldevs(pcap_if_t **alldev, char *errbuf)

参数: alldev指向列表的第一个元素, 列表都是pcap_if_t类型, 如果没有已连接并打开的适配器, 则为NULL;

errbuf存储错误信息

返回值: 函数调用成功返回0, 失败返回-1。

pcap_if_t结构用来描述网络适配器, 定义如下:

```
typedef struct pcap_if_t
{
    struct pcap_if_t *next;           //如果不为空, 则指向下一个元素
    char *name;                       //设备名称
    char *description;                //描述设备
    struct pcap_addr *addresses;       //接口地址列表
    bpf_u_int32 flags;                //PCAP_IF_接口标志
};
```

获得设备列表 (二)

每个pcap_findalldevs() 返回的 pcap_if_t 结构包含了一个pcap_addr结构的列表:

- ❖ 该接口的IP地址
- ❖ 网络掩码
- ❖ 广播地址
- ❖ 目标地址

```
struct pcap_addr
{
    struct pcap_addr *next;
    struct sockaddr *addr;
    struct sockaddr *netmask;
    struct sockaddr *broadaddr;
    struct sockaddr *dstaddr;
};
```



```

pcap_if_t *alldevs, *d;
int i=0;
char errbuf [PCAP_ERRBUF_SIZE];
if (pcap_findalldevs(&alldevs, errbuf) == -1)
    { fprintf(stderr, "Error in pcap_findalldevs: %s\n", errbuf);
      exit(1); }
for(d=alldevs; d ; d=d->next)                /* Print the list */
    { 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; }
pcap_freealldevs(alldevs);

```

```

E:\课件\WpdPack\Examples-pcap\basic_dump\Debug\basic_dump.exe
1. \Device\NPF_GenericDialupAdapter (Generic dialup adapter)
2. \Device\NPF_{A354877C-EAD2-40A8-BDA9-DFE5E5A4593F} (Intel(R) PRO/100 VE Network Connection)
Enter the interface number (1-2):2

```


打开一个适配器开始捕获数据包

获得一个网络适配器后必须打开它才能进行工作。打开适配器函数原型：

```
pcap_t * pcap_open_live ( const char * device,    //适配器的设备标识
                           int snaplen,           //数据包长度限制
                           int promisc,           //是否混杂模式捕获
                           int to_ms,             //能容忍的超时时间
                           char * ebuf )          //出错信息缓存
```

```
pcap_t *adhandle=  
    pcap_open_live(d->name, 65535, 1, 1000, errbuf );
```

设备标识
(字符串)

抓包长度

混杂模式

超时时间

错误信息

函数pcap_open_live () 的返回值是pcap_t类型的指针，pcap_t结构体对用户透明，提供了对一个已打开的适配器实例的描述。windows平台上pcap_t的主要成员有：

```
typedef struct pcap pcap_t;
struct  pcap
{
    ADAPTER      *adapter;
    LPPACKET      Packet;
    int           linktype;           //数据链路层类型
    int           linktype_ext;      // linktype成员扩展信息
    int           offset;            //时区偏移
    int           activated;         //捕获准备好否
    struct pcap_sf sf;
    struct pcap_md md;
    struct pcap_opt opt;
    ...
};
```

捕获数据包（直接方式）

```
int pcap_next_ex ( pcap_t * p,                // 适配器名称
                  struct pcap_pkthdr ** pkt_header, // 捕获数据包首部指针
                  const u_char ** pkt_data ) // 捕获数据包的数据
```

该函数从接口或者文件读取一个数据包。 **pcap_next_ex**用下一个数据包的指向数据包头和数据的指针填充**pkt_header**和**pkt_data**参数。

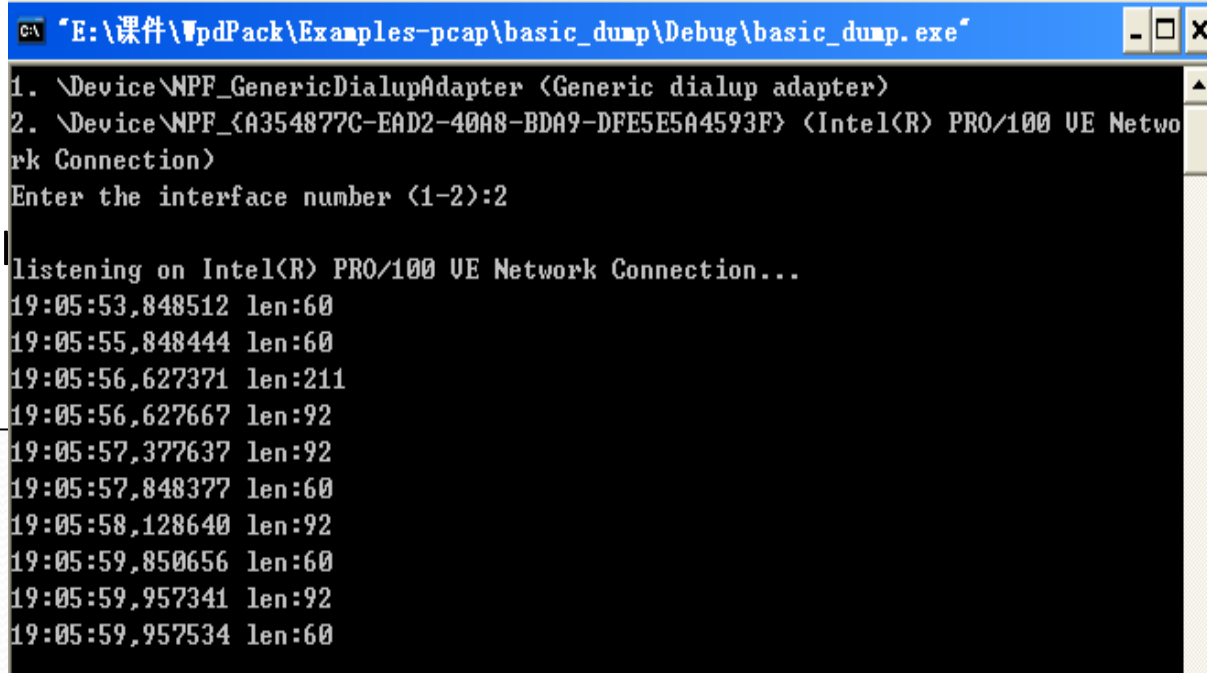
pcap_next_ex() 目前只在Win32下可用，因为它不是属libpcap原始的API。这意味着含有这个函数的代码将不能被移植到Unix上。


```

while ((res = pcap_next_ex(adhandle, &header, &pkt_data)) >= 0)
{
    if (res == 0)
    {
        continue;                /* Timeout elapsed */
    }

    /* convert the timestamp to readable format */
    ltime = localtime(&header->ts.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 packet\n");
    return -1;
}

```



```

C:\E:\课件\WpdPack\Examples-pcap\basic_dump\Debug\basic_dump.exe
1. \Device\NPF_{Generic dialup adapter}
2. \Device\NPF_{A354877C-EAD2-40A8-BDA9-DFE5E5A4593F} <Intel(R) PRO/100 UE Network Connection>
Enter the interface number <1-2>:2

listening on Intel(R) PRO/100 UE Network Connection...
19:05:53,848512 len:60
19:05:55,848444 len:60
19:05:56,627371 len:211
19:05:56,627667 len:92
19:05:57,377637 len:92
19:05:57,848377 len:60
19:05:58,128640 len:92
19:05:59,850656 len:60
19:05:59,957341 len:92
19:05:59,957534 len:60

```

捕获数据包（回调机制）

```
int pcap_loop ( pcap_t * p,                // 适配器名称
                int cnt,
                pcap_handler callback,
                u_char * user )
```

例如：

```
pcap_loop(adhandle, 0, packet_handler, NULL);
typedef void(* pcap_handler)
    ( u_char *user,
      const struct pcap_pkthdr *pkt_header,
      const u_char *pkt_data)
```

过滤数据包

```
int pcap_compile ( pcap_t * p,  
                  struct bpf_program * fp,  
                  char * str,           //过滤表达式  
                  int optimize,  
                  bpf_u_int32 netmask) //网络掩码
```

```
int pcap_setfilter ( pcap_t * p,  
                    struct bpf_program * fp )
```

pcap_compile() 编译一个包过滤器。将一个高级的、布尔形式表示的字符串转换成低级的、二进制过滤语句，以便被包驱动使用。pcap_setfilter() 在核心驱动中将过滤器和捕获过程结合在一起。从这一时刻起，所有网络的数据包都要经过过滤，通过过滤的数据包将被传入应用程序。

过滤表达式

表达式由一个或多个原语组成。原语通常由一个id（名称或者数字）和在它前面的一个或几个修饰符组成。有3种不同的修饰符：

类型 指明id名称或者数字指的是哪种类型。

可能是host，net和port。

例如 “host foo”、“net 128.3”、“port 20”。如果没有类型修饰符，缺省为host。

方向 指明向 和/或 从id传输等方向的修饰符。

可能的方向有src、dst、src or dst 和 src and dst。

例如 “src foo”、“dst net 128.3”、“src or dst port ftp-data”。如果缺省为src or dst。

协议 指明符合特定协议的修饰符。

目前的协议包括ether、fddi、ip、ip6、arp、rarp、tcp和udp等。

例如 “ether src foo”、“arp net 128.3”。

如果没有协议修饰符，则表示声明类型的所有协议。

例如 “src foo”表示 “（ip or arp or rarp） src foo”

“port 53”表示 “（tcp or udp） port 53”。

过滤设置举例

```
char packet_filter[] = "ip and udp";
struct bpf_program fcode;
/* 获取接口地址的掩码，如果没有掩码，认为该接口属于一个C类网络 */
if(d->addresses != NULL)
    netmask=((struct sockaddr_in *)
            (d->addresses->netmask))->sin_addr.S_un.S_addr;
else netmask=0xffffffff;

if(pcap_compile(adhandle, &fcode, packet_filter, 1, netmask) <0 ){
    fprintf(stderr, "\nUnable to compile the filter. Check the syntax.\n");
    pcap_freealldevs(alldevs);          return -1;
}

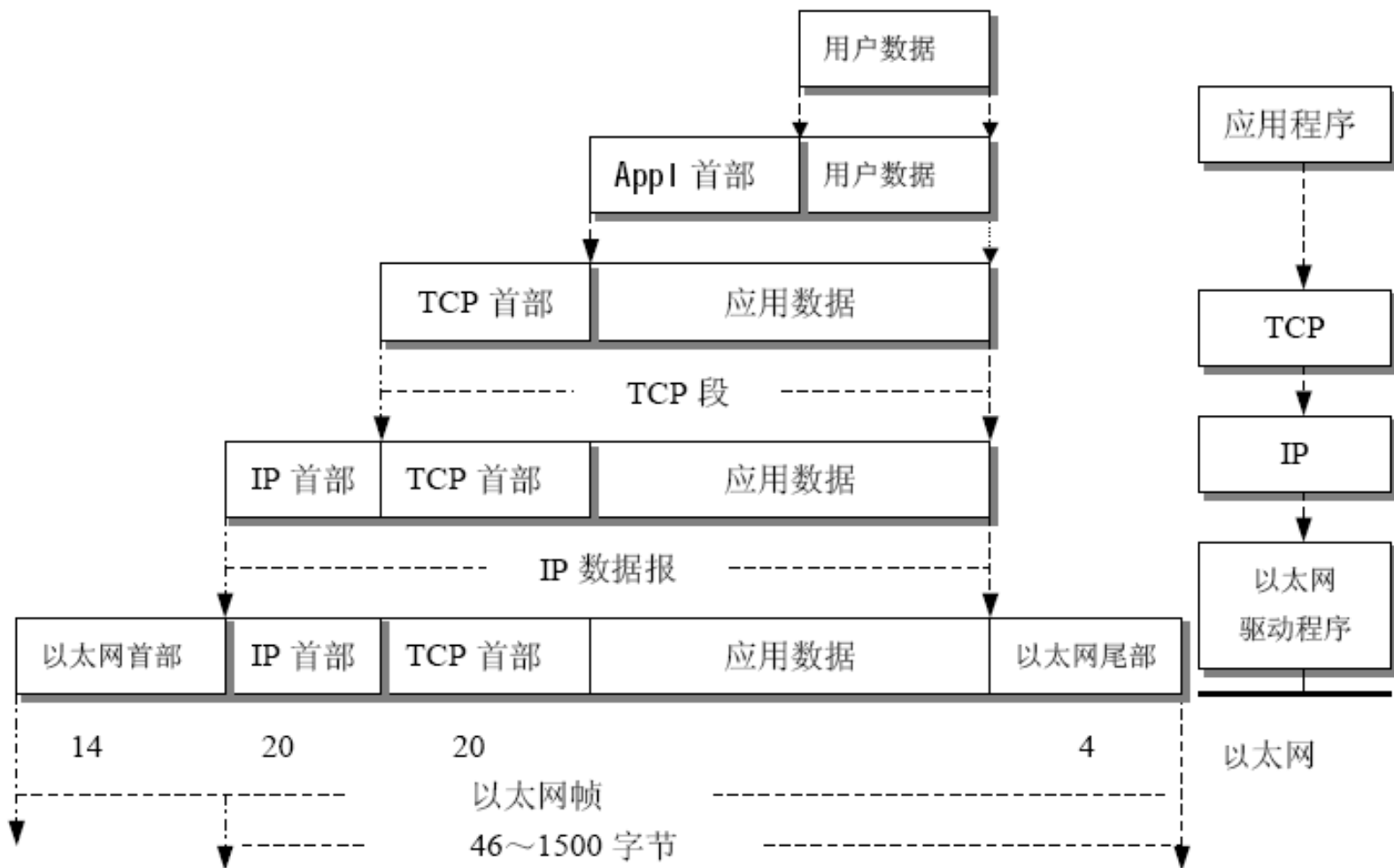
if(pcap_setfilter(adhandle, &fcode)<0){
    fprintf(stderr, "\nError setting the filter.\n");
    pcap_freealldevs(alldevs);          return -1;
}
```


解析数据包

不同的网络使用不同的链路层协议，不知道网络类型就无法定位数据帧（frame），WINPCAP使用如下函数对网络类型进行判断：

```
int pcap_datalink(pcap_t *p)
```

它返回适配器的链路层标志，例如DLT_EN10MB表示以太网（10Mb, 100Mb, 1000Mb及以上），DLT_IEEE802表示令牌环网。可以在设置过滤条件之前先对各个设备的网络类型进行探测，以获知物理层的数据帧格式，接下来就可以针对不同的帧格式解析出封装在其中的报文（packet），例如IP报文，继而拆封IP报文得到TCP或者UDP等报文，再深层次的拆封就是应用程序数据了。



解析数据包

前面我们提到，用户在程序中调用的`pcap_loop()`（或者`pcap_next_ex()`）函数可以进行数据包的捕获，而每一个数据包到达时该函数会调用`pcap_handler()`函数进行数据包处理，返回一个指向**捕获器头部**和一个指向**帧数据**的指针（包含协议头）。

常用的以太网IP、TCP的头部格式如下，其他数据包的格式请参考RFC文档。

注意：在数据包解析时要检查校验和及包的顺序。

Ethernet帧头

8 Bytes	6 Bytes	6 Bytes	2 Bytes
Preamble	Dest Addr	Src Addr	Type

Struct ethernet

```
{  u_char ether_dhost[ETHER_ADDR_LEN];  
    u_char ether_shost[ETHER_ADDR_LEN];  
    u_short ether_type; /* IP? ARP? RARP? 等*/  
};
```


IP报文头

20 个 字 节 固 定 长 度		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
		Version				HLen				Service Type								Total Length															
		Identification																Flags				Fragment Offset											
		Time To Live								Protocol								Header Checksum															
		Source IP Address																															
		Destination IP Address																															

20 个
字节
固定
长度

```

Struct ip {
    #if BYTE_ORDER == LITTLE_ENDIAN
        u_int ip_hl:4,
        ip_v:4;
    #endif
    #if BYTE_ORDER == BIG_ENDIAN
        u_int ip_v:4,
        ip_hl:4;
    #endif
    u_char ip_tos;
    u_short ip_len;
    u_short ip_id;
    u_short ip_off;
    #define IP_RF 0x8000
    #define IP_DF 0x4000
    #define IP_MF 0x2000
    #define IP_OFFMASK 0x1fff
    u_char ip_ttl;
    u_char ip_p;
    u_short ip_sum;
    struct in_addr ip_src, ip_dst;
};

/* header length */
/* version */

/* version */
/* header length */
/* not_IP_VHL */
/* type of service */
/* total length */
/* identification */
/* fragment offset field */
/* reserved fragment flag */
/* don't fragment flag */
/* more fragments flag */
/* mask for fragmenting bits */
/* time to live */
/* protocol */
/* checksum */
/* source and dest address */

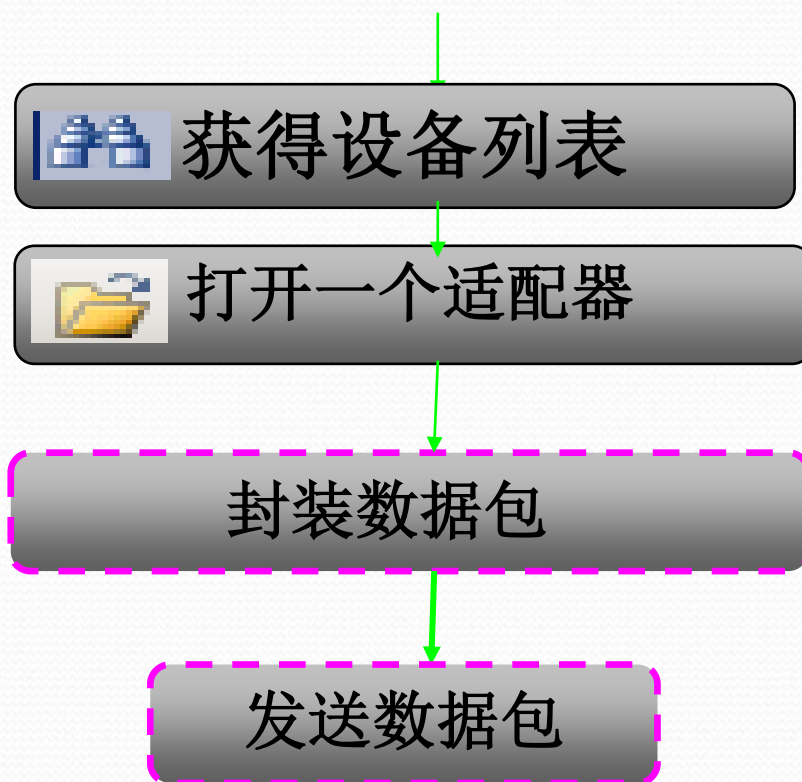
```

TCP报文头

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Source Port																Destination Port															
Sequence Number																															
Acknowledgement Number																															
D. Offset		Reserved				Control										Window															
Checksum																Urgent Pointer															
Options																														Padding	


```
Struct tcp {
    u_short th_sport;           /* source port */
    u_short th_dport;          /* destination port */
    tcp_seq th_seq;             /* sequence number */
    tcp_seq th_ack;             /* acknowledgement number */
    #if BYTE_ORDER == LITTLE_ENDIAN
    u_int th_x2:4,              /* (unused) */
    th_off:4;                   /* data offset */
    #endif
    #if BYTE_ORDER == BIG_ENDIAN
    u_int th_off:4,             /* data offset */
    th_x2:4;                   /* (unused) */
    #endif
    u_char th_flags;
    #define TH_FIN 0x01
    #define TH_SYN 0x02
    #define TH_RST 0x04
    #define TH_PUSH 0x08
    #define TH_ACK 0x10
    #define TH_URG 0x20
    #define TH_ECE 0x40
    #define TH_CWR 0x80
    #define TH_FLAGS (TH_FIN|TH_SYN|TH_RST|TH_ACK|TH_URG|TH_ECE|TH_CWR)
    u_short th_win; /* window */
    u_short th_sum; /* checksum */
    u_short th_urp; /* urgent pointer */
};
```

WinPcap的典型应用流程（发送）



发送数据包

- 发送单个数据包 **pcap_sendpacket()**
- 发送队列 **pcap_sendqueue_transmit()** （查看 winpcap 手册）

pcap_sendpacket发送单个数据包

打开适配器后，调用pcap_sendpacket() 函数来发送一个手写的数据包。

pcap_sendpacket()用一个包含要发送的数据的缓冲区、该缓冲区的长度和发送它的适配器作为参数。注意该缓冲区是不经任何处理向外发出的，应用程序必须产生正确的协议头。

int pcap_sendpacket (pcap_t * p,	//已经打开的适配器
u_char * str,	//要发送数据包的内容
int optimize)	//发送数据包的长度

```
u_char packet[100];
if((fp = pcap_open_live(argv[1], 100, 1, 1000, error) ) == NULL)
{
    fprintf(stderr, "\nError opening adapter: %s\n", error);
    return;
}
/* Supposing to be on ethernet, set mac destination to 1:1:1:1:1:1 */
packet[0...5]=1;
/* set mac source to 2:2:2:2:2:2 */
packet[6...11]=2;
/* Fill the rest of the packet */
for(i=12;i<100;i++){
    packet[i]=i%256;
}
/* Send down the packet */
pcap_sendpacket ( fp, packet, 100);
```