目　　录

第 1 章　网络安全 …… (1)
1.1　网络安全问题 …… (1)
1.1.1　网络安全重要性 …… (1)
1.1.2　网络安全目标 …… (2)
1.1.3　网络安全策略 …… (3)
1.2　网络安全技术 …… (3)
1.2.1　网络数据包捕获技术 …… (4)
1.2.2　网络协议分析技术 …… (5)
1.2.3　网络数据包生成技术 …… (6)
1.2.4　网络安全扫描技术 …… (7)
1.2.5　网络防火墙技术 …… (8)
1.2.6　网络入侵检测技术 …… (9)
1.3　网络安全开发 …… (9)
1.3.1　网络安全开发的主要内容 …… (9)
1.3.2　网络安全开发所需的知识 …… (11)
第 2 章　网第安全开发包 …… (13)
2.1　网络安全开发包技术 …… (13)
2.1.1　网络安全开发包的作用 …… (13)
2.1.2　网络安全开发包的种类 …… (14)
2.1.3　如何使用开发包 …… (14)
2.2　常用的网络安全开发包简介 …… (15)
2.2.1　Libpcap …… (15)
2.2.2　WinPcap …… (16)
2.2.3　数据包构造和发送开发包 Libnet …… (17)
2.2.4　通用网络安全开发包 Libdnet …… (18)
2.2.5　网络入侵检测开发包 Libnids …… (19)
第 3 章　网络数据包捕获开发包 Libpcap …… (21)
3.1　Libpcap 概述 …… (21)
3.1.1　Libpcap 简介 …… (21)
3.1.2　Libpcap 的作用 …… (21)
3.1.3　Libpcap 的安装 …… (22)
3.2　Libpcap 的组成 …… (24)
3.2.1　BPF 捕获机制 …… (24)
3.2.2　过滤规则 …… (25)
3.2.3　网卡设置 …… (27)

3.2.4 文件 …… (27)
3.3 Libpcap 数据结构 …… (31)
3.3.1 pcap …… (31)
3.3.2 pcap_file_header …… (32)
3.3.3 pcap_pkthdr …… (33)
3.3.4 pcap_stat …… (33)
3.3.5 pcap_if …… (34)
3.3.6 pcap_addr …… (34)
3.4 Libpcap 函数 …… (35)
3.4.1 网络接口函数 …… (35)
3.4.2 规则函数 …… (35)
3.4.3 数据包捕获函数 …… (36)
3.4.4 文件相关函数 …… (37)
3.4.5 错误处理函数 …… (38)
3.4.6 辅助函数 …… (38)
3.5 Libpcap 的使用 …… (40)
3.5.1 最简单的应用程序 …… (41)
3.5.2 以太网数据包的捕获 …… (46)
3.5.3 ARP 数据包的捕获 …… (53)
3.5.4 IP 数据包的捕获 …… (59)
3.5.5 TCP 数据包的捕获 …… (65)
3.5.6 UDP 数据包的捕获 …… (74)
3.5.7 ICMP 数据包的捕获 …… (80)
3.5.8 综合程序 …… (88)
3.6 小结 …… (107)
第 4 章 Windows 网络数据包捕获开发包 WinPcap …… (108)
4.1 WinPcap 介绍 …… (108)
4.2 WinPcap 的组成 …… (110)
4.3 WinPcap 数据结构 …… (112)
4.4 WinPcap 函数 …… (113)
4.5 WinPcap 开发包的使用 …… (117)
4.5.1 使用 WinPcap 的一般流程 …… (117)
4.5.2 捕获数据包 …… (122)
4.5.3 发送数据包 …… (136)
4.6 小结 …… (139)
第 5 章 数据包构造和发送开发包 Libnet …… (141)
5.1 Libnet 介绍 …… (141)
5.1.1 Libnet 的由来 …… (142)
5.1.2 Libnet 的作用 …… (143)
5.2 Libnet 的组成 …… (144)

5.2.1 文件 …… (144)
5.2.2 函数构成 …… (146)
5.2.3 协议部分 …… (149)
5.2.4 Libnet 的安装 …… (150)
5.3 Libnet 数据结构 …… (160)
5.3.1 常量 …… (160)
5.3.2 协议头的数据结构 …… (164)
5.3.3 核心数据结构 …… (192)
5.4 Libnet 函数 …… (199)
5.4.1 核心函数 …… (199)
5.4.2 地址函数 …… (200)
5.4.3 数据包构造函数 …… (201)
5.4.4 数据包发送函数 …… (218)
5.4.5 高级处理函数 …… (218)
5.4.6 Libnet 句柄队列操作函数 …… (219)
5.4.7 辅助函数 …… (220)
5.5 使用 Libnet 的开发流程 …… (223)
5.5.1 初始化 …… (223)
5.5.2 构造数据包 …… (225)
5.5.3 数据包检测和发送 …… (226)
5.6 使用 Libnet 开发包 …… (226)
5.6.1 构造 ARP 数据包 …… (227)
5.6.2 构造 IP 数据包 …… (236)
5.6.3 构造 TCP 数据包 …… (245)
5.6.4 构造 UDP 数据包 …… (252)
5.6.5 构造 ICMP 数据包 …… (258)
5.6.6 构造 BGP 数据包 …… (287)
5.6.7 构造 DHCP 数据包 …… (293)
5.6.8 构造多个数据包 …… (300)
5.7 小结 …… (304)
第 6 章 通用网络安全开发包 Libdnet …… (305)
6.1 Libdnet 概述 …… (305)
6.1.1 Libdnet 简介 …… (305)
6.1.2 Libdnet 的主要功能 …… (306)
6.1.3 Libdnet 的安装 …… (306)
6.2 Libdnet 的组成 …… (309)
6.3 Libdnet 数据结构 …… (310)
6.3.1 网络地址结构 …… (310)
6.3.2 协议结构 …… (311)
6.3.3 ARP 缓存的数据结构 …… (331)

6.3.4 网络路由表数据结构 …… (331)
6.3.5 网络接口的数据结构 …… (332)
6.3.6 防火墙结构 …… (332)
6.4 Libdnet 函数 …… (333)
6.4.1 网络地址操作函数 …… (333)
6.4.2 以太网协议函数 …… (335)
6.4.3 IPv4 协议函数 …… (336)
6.4.4 IPv6 协议函数 …… (337)
6.4.5 数据块操作函数 …… (338)
6.4.6 ARP 缓存操作函数 …… (340)
6.4.7 网络接口函数 …… (340)
6.4.8 网络路由函数 …… (341)
6.4.9 随机数函数 …… (342)
6.4.10 防火墙函数 …… (343)
6.4.11 IP 隧道函数 …… (344)
6.5 Libdnet 开发包的使用 …… (345)
6.5.1 简单应用 …… (345)
6.5.2 构造和发送网络数据包 …… (349)
6.5.3 ARP 缓存操作 …… (364)
6.5.4 网络接口操作 …… (367)
6.5.5 网络路由表操作 …… (375)
6.5.6 防火墙操作 …… (378)
6.6 小结 …… (383)
第 7 章 网络入侵检测开发包 Libnids …… (384)
7.1 Libnids 概述 …… (384)
7.1.1 Libnids 的使用范围 …… (384)
7.1.2 Libnids 的安装 …… (385)
7.2 Libnids 数据结构 …… (387)
7.2.1 基本常量 …… (387)
7.2.2 tuple4 …… (390)
7.2.3 half_stream …… (390)
7.2.4 tcp_stream …… (391)
7.2.5 nids_prm …… (392)
7.2.6 nids_chksum_ctl …… (395)
7.3 Libnids 函数 …… (395)
7.4 Libnids 的使用 …… (397)
7.4.1 显示 TCP 连接 …… (397)
7.4.2 显示 UDP 数据报 …… (408)
7.4.3 HTTP 协议分析 …… (413)
7.4.4 FTP 协议分析 …… (426)

7.4.5 Telnet 协议分析 …… (439)
7.4.6 POP3 协议分析 …… (450)
7.4.7 SMTP 和 ESMTP 协议分析 …… (458)
7.4.8 IP 数据包的捕获和分析 …… (468)
7.4.9 检测攻击的一个例子 …… (475)
7.5 小结 …… (481)
参考文献 …… (483)

第1章 网络安全

1.1 网络安全问题

1.1.1 网络安全重要性

随着网络技术的飞速发展和网络时代的到来，网络安全问题变得越来越严重。现在，全球每年关于网络安全问题的损失是巨大的。每年关于网络安全问题的报道层出不穷，产生的损失越来越大，为解决网络安全问题而投入的资金也越来越多。

网络安全问题严重影响了人们的生活和工作，以至整个国家的安全。它可能对国家的重大部门造成严重后果，如金融部门、政府机构、军事设施、公共基础设施等。因为现在大多数部门都实现了网络信息系统，这样就为网络安全问题提供了可能产生的土壤。

特别是现在网络技术的普及和发展，更为网络安全问题的发展提供了条件。可以肯定地说，哪里有网络哪里就有网络安全问题。现在，很多个人和家庭都连接到网络，他们的网络安全意识薄弱，是最容易受到网络安全问题困扰的群体。

互联网的影响已经逐渐渗透到我国国民经济的各个领域和人民生活的各个方面。但是，随着互联网应用的不断创新与发展，网络安全也面临着前所未有的严峻形势，网络安全领域所面临的挑战日益严峻。

所谓网络安全，是指网络系统的硬件、软件及系统中的数据信息能够受到保护，不会因为偶然或者恶意原因而遭破坏、更改、泄露，系统能够连续、可靠地运行，网络服务不被中断。但在现实中，绝对的网络安全是没有的。

由于网络安全问题是多种多样的，很多新的网络安全问题不断涌现，防不胜防。已知的网络安全问题就包括很多种，目前比较有影响的有以下几种：

（1）网络病毒感染。这可以说是最容易遇到的网络安全问题。网络病毒层出不穷，其传播的方式也不断发展，既可以通过邮件传播，也可以通过网络即时通信系统传播，还可以通过网页传播。而且，新的病毒还跟其他网络安全技术（如木马、间谍软件）相结合，产生综合病毒，极难防范。因此，病毒经常给用户带来巨大的经济损失。

（2）木马问题。这个网络安全问题变得越来越突出，很多木马程序在网络上可以轻易得到，而且其使用方法极其简单，但是其危害程度非常高。如果计算机被植入木马，就相当于整个计算机被别人所控制，任人为所欲为了。由于新的木马程序不断涌现，感染木马后又难以察觉，所以木马问题值得重视。

（3）恶意扫描。对网络主机的扫描，是攻击者最先做的事情，它能够检查出主机存在多少漏洞，以及漏洞的种类等很多有价值的信息，它是攻击者进行攻击的基础。本来网络安全扫描技术是一个非常有效的网络防御技术，但是如果被攻击者所使用，就变成了危害网络安全的技术了。

（4）网络监听。它可以监听整个网络的数据包，然后查看其数据内容。本来，网络监听可以用于网络管理和网络流量检测，但是攻击者使用它就变成了窃取网络信息的行为了。

1.1.3 网络安全策略

网络安全问题产生的原因是多种多样的，它可能是由于系统本身的问题，如目前使用最广泛的 TCP/IP 协议，由于历史的原因，TCP/IP 协议存在很多的安全漏洞；也可能是用户管理出现的问题，如由于管理不当或管理的疏忽而造成网络存在安全问题。所以，网络的管理至关重要，很多出现的网络安全问题并不是由于外在的网络攻击造成的，而是由于管理的漏洞而产生的。

在网络安全中，除了采用各种网络安全技术措施之外，加强网络的安全管理，制定有关规章制度，对于确保网络安全、可靠地运行，将起到十分有效的作用。网络的安全管理策略有很多，如确定安全管理等级和安全管理范围，制定网络系统的维护制度和应急措施等。

确定一个完整的网络安全策略是保证网络安全的一个重要手段。现在比较流行的一个网络安全模型 PPDR（Policy Protection Detection Response）就是以网络安全策略为核心的。PPDR 安全模型是一个动态模型，它包括策略（Policy）、防护（Protection）、检测（Detection）、响应（Response）四个部分。策略是这个模型的核心，反映了整个网络所要具备的网络安全目标，网络安全的其他几个方面要围绕着策略而进行。PPDR 是一个基于时间的模型，首先是防护，对整个网络进行保护，但它是被动的；然后就是对网络进行不断的检测，检测在这个模型中占据重要的地位；如果在检测的过程中发现有问题存在，就要及时作出响应，这时就要执行响应模块。

1.2 网络安全技术

随着对网络安全的不断研究，涌现出了很多网络安全技术，如网络安全扫描技术、网络数据包生成技术、网络防火墙技术、入侵检测技术、协议分析技术、蜜罐技术、加密技术以及认证技术等。

网络安全包括的内容极其广泛，它跟其他领域的知识联系得非常紧密。由于网络上传输的数据是信息，所以信息安全技术也与网络安全密切相关。信息安全与网络安全密不可分，有人把网络安全归纳到信息安全之中，也说明了网络安全和信息安全之间有密切联系。其中，最能体现这种联系的就是密码技术。密码技术是信息安全的核心技术，而加密技术是密码技术的一个重要内容。若在网络中传播的信息不想让别人知道，就需要进行加密。信息交换加密技术主要分为两类：对称加密和非对称加密。在对称加密技术中，对信息的加密和解密都使用相同的密钥。这种加密方法可简化加密处理过程，信息交换双方都不必彼此研究和交换专用的加密算法。如果在交换阶段私有密钥未曾泄露，那么机密性和报文完整性就可以得到保证。在非对称加密体系中，密钥被分解为一对。这对密钥中任何一个都可以作为公开密钥通过非保密方式向他人公开，而另一个作为私有密钥加以保存。公开密钥用于加密，私有密钥用于解密，私有密钥只能由生成密钥的交换方掌握，公开密钥可广泛公布，但它只对应于生成密钥的交换方。非对称加密方式可以使通信双方无须事先交换密钥就可以建立安全通信，广泛应用于身份认证、数字签名等信息交换领域。非对称加密体系一般建立在某些已知的数学难题之上，是计算机复杂性理论发展的必然结果，其中最具代表性的是 RSA 公钥密码体制。

密码技术是信息安全的基石，有了它就可以发展很多其他安全系统，包括计算机网络系统。与密码技术密切相关的网络安全技术就是安全协议，如 Kerberos 协议、IPSec 协议。它

们是安全协议，只是一种协议，但跟密码技术密切联系。由密码技术而发展起来的 PKI 技术更是一个重点内容，PKI（Public Key Infrastructure）是用公钥原理和技术实施和提供安全服务的具有普适性的安全基础设施。它主要使用了两种技术，即数字签名技术和消息认证码技术。还有为了保护 Web 通信的安全，于是产生了 SSL（Secure Socket Layer）/TLS（Transport Layer Security）安全协议。它们的主要目的是为两个通信个体之间提供保密性和完整性。其中比较著名的是 OpenSSL，其他安全协议还有 S/MIME, SET, SSH 和 HTTPS。

网络安全除了与信息安全技术密切相关外，还与其他方面的内容有直接联系，如计算机网络，特别是网络协议方面。由于计算机网络的核心是网络协议，所以讨论协议与网络安全的关系是一个重要的两络安全内容。注意这里讨论的协议与网络安全不是指安全协议（Ipsec 协议、Kerberos 协议等），而是指由于网络协议而产生的一些网络安全问题。例如由于 TCP/IP 协议的一些缺陷而产生的网络安全问题等。

一些网络攻击方式主要是针对计算机网络协议而发生的攻击。由于流行的 TCP/IP 协议存在固有的缺陷，所以会产生各种各样的网络安全问题。为了解决这些网络安全问题，于是就提出了很多与网络协议密切相关的网络安全技术，如上述的防火墙技术、入侵检测技术、网络安全扫描技术、协议分析技术和数据包生成技术等。这些技术的出现，都是为了解决与协议相关的网络安全问题的。

在此着重介绍一些与网络协议密切相关的网络安全技术。

1.2.1 网络数据包捕获技术

很多网络安全系统最首要的任务就是捕获网络上的数据信息，而网络数据包捕获技术就解决了这个问题。网络数据包捕获技术是从网络上捕获所有的或者特定的网络数据包信息，供其他网络安全系统所用。它的功能非常简单，就是捕获网络上的数据包信息。

不同的网络有不同的捕获技术，不同的操作系统其捕获的机理也稍有不同，所以在选择捕获技术的时候要慎重考虑。由于现在用得最多的网络形式是以太网，所以下面讨论最多的是以太网环境下的数据包捕获。而以太网采用了 CSMA/CD 技术，它使用了广播机制，所有与网络连接的机器都可以看到网络上传播的数据。在以太网环境下捕获数据包是非常容易的。操作系统提供的捕获机制主要有以下三种。

1. SOCK_PACKET 类型套接口

Linux 中套接字类型中有一种套接字类型 SOCK_PACKET。SOCK_PACKET 类型的 socket 可以接收网络上所有数据包。它的实现由操作系统提供的编程接口来进行。

2. 数据链路提供者接口（DLPI）

数据链路提供者接口（DLPI，Data Link Provider Interface）定义了数据链路层向网络层提供的服务，是数据链路服务的提供者和使用者间的一种标准接口，在实现上基于 UNIX 的流（Streams）机制。数据链路服务的使用者既可以是用户的应用程序，也可以是访问数据链路服务的高层协议，如 TCP/IP 等。

3. 伯克利数据包过滤器（BPF）

伯克利数据包过滤器（BPF，Berkeley Packet Filter ）是一个高效的数据包捕获机制。它工作在操作系统的内核层。BPF 主要由两部分组成：网络转发部分和数据包过滤部分。网

络转发部分是从链路层中捕获数据包并把它们转发给数据包过滤部分，数据包过滤部分是从接收到的数据包中接收过滤规则决定的网络数据包，其他数据包就被抛弃。

这两个部分都是在操作系统内核层实现的，它提供给应用层的数据包是过滤后的数据包，所以捕获数据包和过滤数据包都是在内核中完成的，效率很高。而且，它使用了数据缓存机制，使捕获数据包缓存在内核中，达到一定量的时候就传递给应用程序，这样也提高了处理的效率。基于 BSD 的系统使用 BPF，基于 SVR4 的系统一般使用 DLPI。从效率上来讲，BPF 比 DLPI 性能好很多，而 SOCK_PACKET 更弱。

在实际应用中，实现网络数据包捕获技术的代表是 Libpcap。Libpcap 是一个专业的跨平台的网络数据包捕获开发包。使用 Libpcap 可以很轻松地实现网络数据包的捕获功能，它的捕获机制就是 BPF 捕获机制。

另外，在 Windows 平台有与 Libpcap 兼容的 Winpcap 开发包，它是专门针对 Windows 平台而开发的，是 Windows 平台下的专业网络数据包捕获开发包，它也屏蔽了不同 Windows 操作系统的区别。

1.2.2 网络协议分析技术

由于网络的核心是网络协议，所以网络协议分析技术在网络安全领域也是一项重要的技术。网络协议分析是指对网络上的数据进行相应的协议分析。网络上的协议是多种多样的，所以产生的数据也是不同的，但是一个网络数据归根结底是基于协议产生的，也就是说任意一个网络数据都使用了一定的协议。究竟是什么协议呢？那就要对数据包进行协议分析，得出数据包的整个协议内容。

现在流行的协议是 TCP/IP 协议栈。它里面最核心的协议有以太网协议、ARP/RARP 协议、IP 协议、UDP 协议、TCP 协议、ICMP 协议等。当然还有很多应用层协议，如 HTTP 协议、FTP 协议、Telenet 协议、DNS 协议、POP3 协议等。新的协议层出不穷，各种各样的网络系统使用各种各样的协议，然后它们之间才能够通信。协议是公开的，不然就不可能进行通信，所以对协议的分析是可行的。

协议分析的过程主要包括三部分内容：捕获数据包、过滤数据包和具体协议分析。

1. 捕获数据包

对网络数据包的协议分析，第一步就是要捕获网络上的数据包。可以使用专业的捕获数据包的开发包，如比较著名的开发包 Libpcap 和 Winpcap，在本书的后面章节将对它们进行详细讨论。

当然，也可以不使用专业的捕获数据包的开发包，而利用操作系统提供的功能来实现，但是那样是比较烦琐的，容易出错。在不同的操作系统下面，捕获数据包的实现不尽相同。

2. 过滤数据包

由于网络上的数据信息量是庞大的，不可能对每个数据进行协议分析，并且在实际应用中有时只想分析某种具体的协议，其他协议都不用考虑。这样，就需要对捕获到的数据包进行过滤。

过滤的方式有两种，一种是在内核层就过滤掉，另一种是在应用层过滤。显然，第一种的效率要高一些，因为从内核层到应用层之间的转换是费时费力的，这样对性能有很大的影响。

如果使用开发包 Libpcap，它里面提供了 BPF 过滤机制，它是在内核层实现过滤的，效率很高，而且 BPF 过滤机制进行了优化处理，效果明显改善。所以，使用 Libpcap 不仅实现了数据包的捕获功能，也可以实现数据包的过滤功能。

3. 具体协议分析

捕获到特定的网络数据包之后，就可以分析网络协议了。根据 TCP/IP 协议层次的概念，对网络数据包的协议分析是从链路层开始的。首先分析数据包的链路层协议，如以太网协议等。其次根据链路层协议的分析结果分析网络层协议，判断网络层的协议是什么，如 IP 协议等。然后再根据网络层协议分析的结果分析传输层协议，如 TCP 协议、UDP 协议等。最后根据传输层协议分析应用层协议，如 HTTP 协议、FTP 协议等。这样，一层一层地分析下去，就可以把整个数据包的协议都分析出来。

使用网络协议分析技术可以设计专门的网络协议分析系统和各种网络嗅探器，如著名的网络嗅探器 Tcpdump 和 Windump，著名的网络协议分析系统 Ethereal 等。

网络协议分析技术还可以应用到其他各种系统中。例如，它可以应用于网络流量统计系统、网络监视系统、网络入侵检测系统、网络安全扫描系统等。可以说，网络协议分析技术是各种网络安全系统的基础，掌握网络协议分析技术对于研究和设计其他网络安全系统是很有裨益的。

1.2.3 网络数据包生成技术

数据包生成技术是指人工构造网络数据包，然后把此网络数据包发送到网络上，让它们像正常的网络数据信息一样传输的技术。网络数据包捕获技术实现的是被动地从网络上捕获数据包，而数据包生成技术是主动地构造网络数据包，并把它们放到网络上。网络数据包构造技术的功能包括构造各种各样的网络协议数据包，如 TCP/IP 协议栈中的各种协议，然后把构造的数据包发送到网络上。由于网络协议是多种多样的，所以构造的网络数据包也应该是多种多样的。

利用网络数据包生成技术，可以构造各种各样的网络安全系统，如网络安全扫描系统、网络安全测试系统。

网络安全扫描系统扫描网络或主机的漏洞，可以利用数据包生成技术来完成构造数据包的功能，然后发送给远程主机来探测远程主机，根据返回的信息来检查远程主机的漏洞。

使用网络安全测试系统可检测其他安全系统（如防火墙系统、入侵检测系统）的性能。构造各种各样的网络数据包来检测防火墙的性能，是测试网络防火墙的一个重要手段。同样的道理，对入侵检测系统的测试，也可以构造不同的网络数据包进行测试。所构造的网络数据包不仅可以是正常的网络数据包（符合协议规定），也可以构造异常的网络数据包（不符合协议规定）。通过构造异常的网络数据包来检测入侵检测系统的性能，是非常有效的一个检测方法。

利用网络数据包生成技术可以产生各种各样的网络攻击手段。例如，TCP SYN 拒绝服务攻击（SynFlood）、死亡之 ping（ping of death），攻击泪滴（teardrop）等，都可以使用数据包生成技术来实现。可以说，数据包生成技术是攻击者用得最多的一种技术。

在实际应用中，数据包生成技术的代表是 Libnet。Libnet 是一个专业的网络数据包生成开发包。利用 Libnet 可以构造任意的网络协议数据包，然后发送到网络上。Libnet 现在支持

的协议种类很多，并且开发者只要根据 Libnet 的规范还可以对其进行其他协议的扩展。

1.2.4 网络安全扫描技术

网络安全扫描技术是通过 Internet 远程检测网络或主机的漏洞的网络安全技术，它是网络安全领域的重要技术之一。通过对网络的扫描，网络管理员可以了解网络的安全配置和运行的应用服务，及时发现安全漏洞，客观评估网络风险等级。网络管理员可以根据扫描的结果更正网络安全漏洞和系统中的错误配置，在黑客攻击前进行防范。如果说防火墙和网络监控系统是被动的防御手段，那么安全扫描就是一种主动的防范措施，可以有效避免黑客攻击行为，做到防患于未然。

网络安全扫描系统是通过网络安全扫描技术实现的一个安全工具，它在网络安全领域占有重要的地位。利用它可以做很多事情。如果是管理员，利用网络安全扫描系统可以查看系统存在的漏洞，及时进行防御和修补。如果是攻击者，利用网络安全扫描系统就可以发现被攻击者的漏洞，然后对其进行攻击。扫描技术的重要性在于它把烦琐的安全检测，通过程序来自动完成，这不仅减轻了网络管理员的工作，而且缩短了检测时间。同时，也可以认为扫描系统是一种网络安全评估软件，利用扫描器，可以快速、深入地对远程主机或网络进行安全评估。现在网络安全扫描系统的发展趋势也是向着网络安全评估专家系统进行转变的。

网络安全扫描技术很多，其中主要有以下几种。

1. 端口扫描

端口扫描是最简单的一种扫描技术，通过端口扫描就可以发现系统中哪些端口是开放的，哪些端口是关闭的。这是扫描的第一步，它表示了哪些服务是开放的。例如，如果端口 80 是开放的，一般说明 Web 服务器是打开的。然后，可以进一步对特定的端口进行其他方面的扫描，如漏洞扫描。

2. 漏洞扫描

漏洞扫描是扫描技术的核心，因为扫描的目的就是要发现漏洞。如果知道了哪些端口是开放的，就知道对应的服务器是打开的，然后就可以针对特定的服务进行漏洞扫描。不同的服务器，其漏洞的形式是不一样的，所以就可以使用专门的漏洞扫描系统，如针对 Web 服务器的 CGI 漏洞扫描。

3. 特殊扫描

特殊扫描指的是对一些特殊对象的扫描。例如，对操作系统的指纹识别，利用扫描技术来识别出远程主机使用的是什么操作系统，它的版本号是多少等信息。还有数据库扫描，它针对数据库的特点，找出与数据库相关的一些漏洞。

实际应用中的网络安全扫描系统的佼佼者是 Nmap 软件。Nmap 是一个非常优秀而著名的网络安全扫描工具，使用它可以实现很多形式的扫描技术，实现很多功能。它支持多种协议的扫描，如 UDP，TCP connect, TCP SYN (half open), ftp proxy(bounce attack), Reverse-ident, ICMP(ping sweep), FIN, ACK sweep, Xmas Tree, SYN sweep 和 Null 扫描。Nmap 还提供一些实用功能，如通过 TCP/IP 来甄别操作系统类型、秘密扫描、动态延迟和重发、平行扫描、通过并行的 PING 侦测下属的主机、欺骗扫描、端口过滤探测、直接 RPC 扫描、分布扫描、灵活的目标选择以及端口的描述。可以说，Nmap 是网络扫描系统的集大成者。

网络安全扫描技术可以通过网络协议分析技术和网络数据包生成技术实现。

1.2.5 网络防火墙技术

网络防火墙技术可以说是发展得比较成熟的网络安全技术。防火墙是一个系统或一组系统，它在企业内网与因特网间执行一定的安全策略。防火墙的功能主要是过滤掉不安全服务和非法用户，控制对特殊站点的访问。

防火墙技术主要有以下几种。

1. 包过滤

包过滤防火墙将对每一个接收到的包作出允许 / 拒绝的决定。具体地说，它对每一个数据包的包头，按照包过滤规则进行判定，与规则相匹配的包依据路由表信息继续转发，不匹配的包则丢弃。其技术依据是网络中的分包传输技术，数据被分割成一定大小的数据包，每一个数据包中都会包含一些特定信息，如数据的源地址、目标地址、TCP / UDP 源端口和目标端口等。防火墙通过读取数据包中的地址信息来判断这些数据是否来自可信任的安全站点，一旦发现来自危险站点的数据包，防火墙便会将这些数据丢掉。

2. 网络地址转换

网络地址转换是把 IP 地址转换成临时的、外部的、注册的 IP 地址标准。它允许具有私有 IP 地址的内部网络访问因特网。它还意味着用户不需要为其网络中每一台机器取得注册的 IP 地址。当受保护网连到 Internet 上时，受保护网用户若要访问 Internet，必须使用一个合法的 IP 地址。但由于合法 Internet IP 地址有限，而受保护网络可以使用非正式 IP 地址，为此网络地址转换器在防火墙上装一个合法 IP 地址集。当内部某一用户要访问 Internet 时，防火墙动态地从地址集中选一个未分配的地址分配给该用户，该用户即可使用这个合法地址进行通信。同时，对于内部的某些服务器（如 Web 服务器），网络地址转换器允许为其分配一个固定的合法地址。外部网络的用户就可通过防火墙来访问内部的服务器。这种技术既缓解了少量的 IP 地址和大量的主机之间的矛盾，又对外隐藏了内部主机的 IP 地址，提高了安全性。

3. 代理型防火墙

代理型防火墙也可以被称为代理服务器。代理服务器位于客户机与服务器之间，完全阻挡了二者间的数据交流。从客户机来看，代理服务器相当于一台真正的服务器；而从服务器来看，代理服务器又是一台真正的客户机。当客户机需要使用服务器上的数据时，首先将数据请求发给代理服务器。代理服务器再根据这一请求向服务器索取数据，然后由代理服务器将数据传输给客户机。由于外部系统与内部服务器之间没有直接的数据通道，外部的恶意侵害也就很难伤害到企业内部网络系统。代理防火墙通常支持的一些常见的应用协议有 HTTP，HTTPS/SSL，SMTP，POP3 等。

4. 状态检测技术

传统的包过滤防火墙只是通过检测 IP 包头的相关信息来决定数据流的通过或拒绝，而状态检测技术采用的是一种基于连接的状态检测机制，将属于同一连接的所有包作为一个整体的数据流看待，构成连接状态表，通过规则表与状态表的共同配合，对表中的各个连接状态因素加以识别。这里动态连接状态表中的记录可以是以前的通信信息，也可以是其他相关

应用程序的信息。因此，与传统包过滤防火墙的静态过滤规则表相比，它具有更好的灵活性和安全性。

5. 个人防火墙

个人防火墙是一种能够保护个人计算机系统安全的软件，它可以直接在用户的计算机上运行，使用与状态检测防火墙相同的方式，保护一台计算机免受攻击。通常，这些防火墙安装在计算机网络接口的较低级别上，使得它们可以监视传入、传出网卡的所有网络通信。对遇到的每一种新的网络通信，个人防火墙都会提示用户一次，询问如何处理那种通信。然后个人防火墙便记住响应方式，并应用于以后遇到的相同的网络通信。

1.2.6 网络入侵检测技术

入侵检测是指在特定的网络环境中发现和识别未经授权的或恶意的攻击和入侵，并对此作出反应的过程。而入侵检测系统（IDS）是一套运用入侵检测技术对计算机或网络资源进行实时检测的系统工具。IDS 一方面检测未经授权的对象对系统的入侵，另一方面监视授权对象对系统资源的非法操作。入侵检测作为一种积极主动的安全防护技术，提供了对内部攻击、外部攻击和误操作的实时保护，在网络系统受到危害之前对入侵进行拦截和响应。

根据检测的对象来分，可分为基于主机的入侵检测系统（HIDS, Host Intrusion Detection System）和基于网络的入侵检测系统（NIDS, Network Intrusion Detection System）。入侵检测技术主要有两种，分别为异常检测（Anomaly Detection）和误用检测（Misuse Detection）。异常检测也被称为基于行为的检测，基于行为的检测指根据使用者的行为或资源使用状况来判断是否入侵。使用异常检测的方法主要有统计分析技术和神经网络技术。误用检测也称为基于知识的检测，它指运用已知攻击方法，根据已定义好的入侵模式，通过判断这些入侵模式是否出现来检测。使用误用检测的方法主要有模式匹配、专家系统、模型推理、统计检测分析和状态转换分析等。

Snort 是开放源码网络入侵检测系统，它是一个轻量级的入侵检测系统，但是它的功能非常丰富，性能非常优越。它能够在 IP 网络上进行执行实时的网络流量分析和数据包记录，它能够进行数据包的协议分析，内容检查和匹配，能够用来检测不同的攻击和探测，如缓冲区溢出、秘密端口扫描、CGI 攻击、SMB 探测、操作系统识别尝试等。Snort 的开发是基于开发包 Libpcap 和开发包 Libnids 的。前者是一个网络数据包捕获开发包（上面提到过），而后者是一个专业的网络入侵检测开发包。

Libnids 是一个提供了 IP 重组和 TCP 数据流重组的网络入侵检测系统开发库，它是一个专业的网络入侵检测开发包。利用 Libnids 可以使网络入侵检测的设计变得简单，它提供了比较基础的功能，其中重要的就是 IP 重组和 TCP 流的跟踪。利用 Libnids 可以实现很多常见的网络入侵行为的检测功能。Libnids 还提供了扫描检测功能，使用它可以检测到扫描行为。

1.3 网络安全开发

1.3.1 网络安全开发的主要内容

网络安全开发体现在各个方面。例如，在网络安全研究方面，在研究的过程中要用实际

的模型来进行测试验证，这就需要进行网络安全开发，以设计和实现基本的模型来验证该研究的正确性和性能分析。

当然，最直接的网络安全开发主要体现在网络安全应用程序的设计方面。由于网络安全技术的多样性，网络安全的应用程序也包括很多内容。每一个网络安全应用程序只解决某一部分网络安全问题，随着新的网络安全问题的不断涌现，也会产生新的网络安全应用程序。

已知的网络安全应用程序非常多，如网络防火墙系统、网络入侵检测系统、防病毒系统、网络安全扫描系统、网络加密系统等。新的网络安全应用程序不断出现，如比较新的网络安全程序、蜜罐系统。当然，还有其他各种各样的应用程序，这里不一一列举。

如上所述，网络安全包括的内容非常广泛，但最主要的体现为与网络协议相关的安全问题上。由于现在使用的 TCP/IP 协议族具有固有的缺陷，所以产生了很多的与协议相关的安全问题。很多网络攻击都是针对网络协议而产生的，如 IP 欺骗、IP 碎片攻击、拒绝服务攻击、TCP 会话劫持攻击等。为了解决这些与网络协议相关的网络安全问题，就产生了很多的网络安全技术，如上面所讨论的防火墙技术、协议分析技术、入侵检测技术、网络安全扫描技术等。

所以，与网络协议相关的网络安全问题是一个重要方面。本书着重讨论与网络协议密切相关的网络安全开发包技术，使用这些网络安全开发包可以开发各种各样的网络安全应用程序。与网络协议相关的网络安全开发主要体现在以下几个方面。

1. 网络防火墙开发

网络防火墙技术是网络安全技术的核心部分，它是应用最多的一种网络安全技术。关于网络防火墙的开发是非常重要的，它涉及的内容非常多，但是其根本的原理离不开对网络数据信息的控制。而网络数据信息又是网络协议所产生的，所以对网络防火墙的开发必须了解网络协议的运行机理。

2. 网络入侵检测开发

网络入侵检测技术是比较新的一个网络安全技术，它把被动的防御理念变成主动的防御。基于网络的入侵检测系统研究的对象是计算机网络数据信息，也是跟网络协议密切相关的。入侵检测开发与网络攻击原理也分不开，不同的网络攻击方式需要不同的入侵检测方式。

3. 网络安全扫描开发

网络安全扫描技术是一种主动的网络安全检测技术，它对网络对象进行主动检测，查看其是否有不安全的因素。把入侵检测技术和网络安全扫描技术结合起来，更好地提供了主动的网络安全防御体系。网络安全扫描的开发也离不开网络协议，它是根据网络协议的特征进行扫描的，离开了网络协议就实现不了扫描。

4. 网络协议分析开发

网络协议分析技术是其他网络安全技术的基础，它研究的对象是网络协议，是对网络协议进行深入探讨的网络安全技术，而很多其他安全技术都要涉及到网络协议，所以网络协议分析的开发至关重要。

4. 网络安全原理

由于网络安全包括的内容非常广泛，要理解和掌握网络安全的基本原理，不可能把所有的领域都很好地掌握。但是有些网络安全方面的基本原理和方法是必须掌握的，如基本密码技术、访问控制技术、防火墙基本原理、入侵检测基本原理等。

5. C 语言或其他语言

C 语言是比较流行的网络安全开发语言，当然还有其他很多的开发语言，但是用得最多的还是 C 语言。本书讨论的网络安全开发包都有基于 C 语言的编程接口，它们的默认开发语言是 C 语言。由于 C 语言的灵活性、高效性和可移植性，所以使用它来设计网络安全程序是非常好的选择。当然也有其他很多语言可以使用，各有优点，如 Perl 语言、Python 语言等。有些网络安全开发包不仅支持 C 语言，也支持这些语言。

与网络安全研究和开发相关的知识不局限于上述几个方面，还包括其他很多方面的内容。网络安全领域与其他很多学科都有关联，它是一个比较典型的交叉领域。

第2章　网络安全开发包

2.1　网络安全开发包技术

2.1.1　网络安全开发包的作用

网络安全开发包是指用于网络安全研究和开发的一些专业开发函数库。它的主要作用是提供用于网络安全研究和开发的基本功能的实现，为研究者和开发者进一步研究和开发网络安全提供编程接口，为网络安全功能的实现提供方便。

一个网络安全开发包是针对特定的网络安全技术而开发的，它可能实现了某一个网络安全技术，也可能实现了某一类网络安全技术。网络安全开发包都是经过很多网络安全研究和开发者的长期研究而形成的，通过不断测试和使用而逐渐成熟起来，并在实际的应用中得到了深入推广。

网络安全开发包的最大作用是用于设计网络安全系统，其直接功能是实现了某些网络安全技术，如网络数据包捕获技术、网络数据包生成技术等。利用它们可以很快地实现某些网络安全系统的基本功能。例如，使用网络数据包捕获技术可以实现网络监视系统的最基本的功能，而使用网络数据包生成技术可以构造网络安全扫描系统的基本功能。网络安全开发包在实际的系统中得到了广泛的应用，一些比较著名的网络安全系统都使用了一些网络安全开发包。

某些网络安全开发包基本上已实现了某个特定的网络安全系统的基本框架和基本功能，然后开发者可以在这个已经构造的基本框架下面进行进一步的深入开发。这样，就为开发者节省了时间和精力，为进一步开发功能更强大的系统提供了方便。

某些网络安全开发包实现的是一些比较重要的但又比较基础的功能，这些功能在大部分的网络安全系统中都要碰到，如网络地址的转换问题、网络接口的配置问题等。很多网络安全系统都会遇到网络地址的转换问题。例如，IPv4 地址提供给人们的通常形态是用点隔开的十进制格式，但是在网络传输中是以二进制格式进行存储的。十进制格式是由二进制翻译过去的。用十进制表示，是为了使人们便于使用和掌握，而在计算机中使用的是二进制格式，这样就必须把二进制转化为十进制。二进制 IP 地址共有 32 位，如 10000011011010110000001100011000，每八位组用一个十进制数表示，并以点分隔，即为 131.107.3.24。有时还要求把十进制格式转换为二进制格式。当然，网络地址形式还有以太网地址、IPv6 地址等，它们有也类似的转换问题。这些地址的转换操作在网络安全系统中使用的次数非常频繁，如果使用一个网络开发包来解决这些问题，就可以直接调用网络安全开发包函数就可以了，不需要另外实现这些操作。有些网络安全开发包就很好地提供了这些功能。

网络安全开发包不仅仅用于设计和实现网络安全系统，它对于网络安全技术的研究也很有帮助。现在网络安全技术不断创新，新的网络安全技术研究也日益突出，如何检验网络技术研究的正确性和性能，通常需要进行实际的研究测试。而网络安全开发包就提供了一个很好的开发平台，为新的网络技术研究提供了环境和基础。

2.1.2 网络安全开发包的种类

网络安全开发包的种类非常多，其实现的功能也千差万别。新的网络安全开发包不断涌现，很多不成熟的网络安全开发包也在不断壮大、成熟。一些在实际的使用中用得比较多的网络安全开发包如下：

（1）网络数据包捕获开发包 Libpcap；

（2）Windows 平台专业数据包捕获开发包 WinPcap；

（3）网络数据包构造和发送开发包 Libnet；

（4）网络入侵检测开发包 Libnids；

（5）通用网络安全开发包 Libdnet。

它们都是经历了很长一段时间才不断发展起来的。很多网络安全开发包现在几乎已经成了一个标准的编程接口。后面的章节要对这些网络安全开发包进行详细的介绍。

除了上述的几种比较著名的网络安全开发包之外，还有很多其他网络安全开发包，如 Libicmp 和 OpenSSL。Libicmp 是专门的 ICMP 协议数据包处理开发包，使用它可以简化 ICMP 协议的操作，如发送 ICMP 数据包和接收 ICMP 数据包等。OpenSSL 是一个著名的网络信息加密传输的开发包，它提供了安全套接字层（SSL，Secure Sockets Layer）和传输层安全（TLS，Transport Layer Security），并提供了加密算法、数字签名算法及证书算法等功能，使用它可以很好地保证数据的完整性、保密性和正确性。

2.1.3 如何使用开发包

网络安全开发包归根结底是一些函数库，它们提供了编程接口。使用网络安全开发包就是要掌握怎样使用这些函数，调用这些编程接口。但在调用这些函数之前，必须对它们的调用方式有个清楚的认识，特别是一些比较复杂的网络安全开发包，它们本身使用的原理就比较复杂，所以还必须了解网络安全开发包的基本原理。

如何使用网络安全开发包，应从以下几个方面进行考虑。

1. 网络安全开发包的基本原理

如果想正确地使用网络安全开发包，首先必须对网络安全开发包有个基本的了解，包括它的基本原理。例如，要掌握网络数据包捕获开发包 Libpcap，就需要了解它是怎样实现网络数据包捕获技术的，怎样实现网络数据包过滤（BPF）功能。Libpcap 中使用了 BPF 机制，那么弄清楚 BPF 机制是一个什么样的机制，对于使用 Libpcap 开发包是非常有帮助的。

2. 网络安全开发包的数据结构

每种网络安全开发包都使用了很多数据结构，这些数据结构在开发包中都要用到，它们是信息的载体。特别是一些典型的数据结构，它们基本上描述了开发包的一些核心内容，掌握它们对于理解和掌握网络安全开发包是至关重要的。有些网络安全开发包使用的数据结构非常复杂，一个数据结构里面还有另一个数据结构，一环套一环。特别是数据结构中的多重指针问题，更要引起足够的重视。例如，在网络数据包构造和发送开发包 Libnet 中就有几个典型的数据结构，如协议块数据结构 libnet_pblock_t、Libnet 句柄数据结构 libnet_t 以及 Libnet 句柄队列 libnet_cqd_t，它们三个之间互相联系又独立使用。了解它们的相互联系对于理解 Libnet 开发包的运行机理很有帮助。

3. 网络安全开发包的输出函数

网络安全开发包的输出函数是提供给用户的编程接口，它是开发者直接打交道的对象，也是开发者最终要掌握的对象。使用网络安全开发包归根结底是调用网络安全开发包提供的输出函数，所以掌握输出函数是最重要的，也是了解网络安全开发包的目的所在。要掌握网络安全开发包的输出函数，必须从三个方面进行了解，即函数的返回值、函数的参数描述和函数的功能。

函数的返回值有时候说明了调用此函数的返回状态，它包括在什么情况下函数调用成功，在什么情况下函数调用失败；如果成功，函数的返回值是什么；如果失败，函数的返回值又是什么。有时候，函数的返回值是一个有价值的信息，而不是函数调用成功的状态。这样，了解了函数的返回信息，就可以把它存储起来。但有时候，函数没有任何返回值。

函数的参数描述说明了函数的参数类型和所表示的意义。函数参数是函数的一个重要内容，它是在函数调用时要特别注意的地方。有一类参数需要特别注意，就是函数指针参数，它可以表示一个回调函数，而回调函数的机理是很多网络安全开发包要使用到的。例如，在网络数据包捕获开发包 Libpcap 中，当捕获到一个数据包之后，就调用回调函数对数据包进行处理。

函数的功能是函数的重点内容，它说明了此函数要实现什么功能，它的作用是什么。网络安全开发包提供的函数很多，它们都是为了实现某一个功能而提出的。掌握函数的功能，是使用函数的前提条件。例如，在网络数据包构造和发送开发包 Libnet 中，有很多函数，不同网络协议数据的构造都用不同的函数来实现。知道了函数的功能，就可以调用相应的函数来构造不同协议的网络数据。

4. 网络安全开发包的使用方法

在掌握了网络安全开发包的基本原理、数据结构和输出函数之后，还应掌握具体使用网络安全开发包的方法。每个网络安全开发包都有自己的使用方法，使用方法也就是网络安全开发包的函数调用过程。哪些函数应该在前面调用和哪些函数应该在后面调用是有顺序的，不能随便颠倒。例如，在网络数据包捕获开发包 Libpcap 中，一般的调用顺序是获得网络接口，然后获得网络接口的网络地址和掩码地址，然后打开网络接口，编译 BPF 过滤规则，设置过滤规则，最后捕获网络数据包。再例如，在网络数据包构造和发送开发包 Libnet 中，一般的调用顺序是设置 Libnet 句柄，然后构造各种网络协议数据块，从网络协议的上层到下层依次构造，最后发送网络数据包。

只有掌握了网络安全开发包的使用方法，才能正确地使用网络安全开发包。在本书中，对每个网络安全开发包的使用方法都是通过举例进行讲解的。这样，读者可以比较迅速地掌握网络安全开发包的使用过程。

2.2 常用的网络安全开发包简介

2.2.1 Libpcap

网络数据包捕获开发包 Libpcap 是一个著名的、专门用来捕获网络数据的编程接口。它在很多网络安全领域得到了广泛的应用，很多著名的网络安全系统都是基于 Libpcap 而开发的，如著名的网络数据包捕获和分析工具 Tcpdump，著名的网络入侵检测系统 Snort，以及

著名的网络协议分析工具 Ethereal 等。

Libpcap 几乎成了网络数据包捕获的标准接口。它效率很高，使用又极其方便，而且它是一个跨平台的网络编程接口。它使用了 BPF 数据包捕获机制，具有快速的网络数据包过滤功能。

由于网络数据包捕获功能是很多网络安全系统都要实现的功能，所以 Libpcap 应用的范围非常广泛。例如，它可以应用于网络嗅探器、网络协议分析、网络入侵检测、网络安全扫描、网络管理、网络统计等网络安全系统。

使用 Libpcap 而开发的网络安全软件非常多，其中几个非常著名的软件在实际的使用中用得非常频繁，包括 Tcpdump，Snort，Ethereal，Nmap，Dsniff 等。

Tcpdump 是一个著名的网络嗅探工具，它是基于 Libpcap 而开发的。使用 Tcpdump 可以对网络数据包进行强有力的分析。使用 Tcpdump 可以捕获网络上的所有数据，也可以根据过滤规则捕获特定的数据。其采用的过滤规则是基于内核的 BPF 过滤规则，具有很高的效率。

Snort 是一款轻量级的网络入侵检测系统，能够在 IP 网络上进行实时的流量分析和数据包记录。它不仅能进行协议分析、内容检索、内容匹配，而且能用于侦测缓冲溢出、隐秘端口扫描、CGI 攻击、SMB 探测、操作系统指纹识别等大量的攻击或非法探测。Snort 使用灵活的规则去描述哪些流量应该被收集或被忽略，并且提供一个模块化的探测引擎。

Ethereal 是一款免费的网络协议分析程序，支持 UNIX，Windows 环境。借助这个程序，既可以直接从网络上抓取数据进行分析，也可以对由其他嗅探器抓取后保存在硬盘上的数据进行分析。利用它能交互式地浏览抓取到的数据包，查看每一个数据包的摘要和详细信息。Ethereal 有多种强大的功能，如支持几乎所有的协议、丰富的过滤语言、易于查看 TCP 会话经重构后的数据流等。

Nmap 是著名的网络安全扫描软件，它也是基于 Libpcap 来捕获目标主机返回的数据包的。Nmap 的基本功能如下：

（1）探测一组主机是否在线；

（2）扫描主机端口，嗅探所提供的网络服务；

（3）推断主机所用的操作系统。

Dsniff 是一个著名的网络数据嗅探软件包。它里面包括的工具很多，可以实现很多强大的功能。其中 dsniff, filesnarf, mailsnarf, msgsnarf, rlsnarf 和 webspy 可以用于嗅探网络上敏感的数据。例如，它可以嗅探简单的密码，可以处理很多协议的认证信息，可以重组 TCP 数据连接。其中 arpspoof, dnsspoof 和 macof 能很容易地嗅探到攻击者通常难以获得的网络信息，sshmitm 和 webmitm 则能用于实现重写 SSH 和 HTTPS 会话，达到中间人攻击。Dsniff 使用了 Libpcap 开发包，还使用了 Libnids 和 Libnet 开发包。

除上述软件之外，还有很多网络安全程序使用了 Libpcap 开发包，如 Nmap 网络安全扫描软件、fragroute 攻击软件、Ettercap 中间人攻击软件、Bro 入侵检测系统、RING 指纹探测软件、Ngrep 网络数据包捕获分析软件、rpcap 远程数据包捕获软件、IPDump 数据包分析工具、Tcpstat 数据包分析工具等。

2.2.2 WinPcap

WinPcap（Windows Packet Capture）是 Windows 平台下一个免费、公共的网络访问系

统，是其他一些安全工具的应用基础。它是 Libpcap 在 Windows 平台下的版本，对 Windows 进行了优化处理和扩展。WinPcap 基于 Libpcap，但不等同于 Libpcap，而是在 Libpcap 的基础上进行了一些功能的扩充，对各种 Windows 系统进行了平台隐蔽和优化处理。

开发包 WinPcap 是 Windows 平台下的专业网络数据包捕获开发包。WinPcap 开发包包括三部分，即数据包捕获和过滤模块 NPF、较低层的网络编程接口 packet.dll 和较高层的编程接口 wpcap.dll。使用 wpcap.dll 编程接口与使用 Libpcap 的过程完全一样，操作方便，也是使用 WinPcap 的最常见的开发方式。

WinPcap 不仅具备了 Libpcap 的捕获数据包的功能，还具有一些其他功能，总结如下：

（1）数据包捕获。跟 Libpcap 一样，WinPcap 主要的功能还是完成了对网络数据包的捕获。它可以捕获在共享网络上传输的各种网络数据包。

（2）数据包过滤。WinPcap 也具备数据包的过滤功能，在接收网络数据包和发送给应用程序之前在内核层对数据包进行过滤。其过滤规则与 BPF 过滤规则兼容。

（3）数据包发送。使用 WinPcap 可以实现数据包的发送功能，它是 Libpcap 在 Windows 下的扩展，可以发送原始的网络数据包，网络数据包的内容可以由开发者自己确定。

（4）流量统计。在 WinPcap 中还实现了流量统计功能，这也是在内核层实现的。

（5）数据包存储。使用 WinPcap 可以在内核中将捕获到的数据包直接存储到磁盘中。在 Libpcap 中也可以对捕获到的数据包进行存储，但是它是在应用层操作的，而不是在内核层。

从 WinPcap 所具备的功能来看，Libpcap 所应用的范围，WinPcap 都可以应用，并且还具备 LibPcap 所不具备的一些功能。

Windump 软件就是基于 WinPcap 而开发的，它其实是 Tcpdump 在 Windows 下的移植版本。其基本功能与 Tcpdump 是一样的，使用方法也一样。在使用 Windump 之前，必须首先安装 WinPcap。

使用 WinPcap 开发的软件还有 Analyzer 工具。Analyzer 是一个 Windows 平台下的网络数据包捕获和分析系统，它可以运行在多种 Windows 平台下。其下载网址为 http://analyzer.polito.it/。

2.2.3 数据包构造和发送开发包 Libnet

Libnet 是一个著名的专业网络数据包构造和发送开发包，它实现了对网络协议的数据包构造并把构造的网络数据包发送到网络的功能。利用 Libnet 可以非常方便地构造各种网络协议的数据，其构造过程都非常相似，这样只需掌握一种协议的构造方法，其他协议的构造过程都是类似的。利用 Libnet 可以方便地进行数据包的发送，把各种构造好的协议数据包发送到网络上，并且还可以发送多个不同的网络数据包。

Libnet 是一个高层次的编程接口，它实现了很多底层的功能，屏蔽了不同操作系统的不同之处。它不仅可以构造任意的网络数据包，还可以对构造好的网络数据包进行修改，其修改的方式与构造的方式是一样的，只需设置修改标记。Libnet 提供了构造不同格式数据包的功能，根据协议数据的格式进行构造，不同的协议有不同的协议特征和不同的参数，Libnet 都提供了灵活的参数传递方式。

由于在某种情况下，有时要构造多个网络数据包。例如，在网络安全扫捕系统中，要构造不同端口的数据包，Libnet 就提供了这样的功能。利用 Libnet 可以构造多个网络数据包

（Libnet 利用了链表数据结构来管理不同的数据包），然后利用 Libnet 进行逐个发送。利用 Libnet 构造多个数据包是极其方便的。

Libnet 提供了两种类型的构造数据包的方式，一是基于原始套接字 IP 层，一是基于链路层。Libnet 还提供了多个高级的构造数据包的方法，其目的是为了简化构造数据包的过程。因为在构造一个网络数据包的时候，它可能包括几个协议数据块，不同的协议数据块使用不同的协议，而有些协议格式是非常复杂的，并且参数设置也非常烦琐。为了简化这些操作，Libnet 去掉了一些参数的设置，而使用默认的参数设置方式来构造协议块。这样，使用 Libnet 提供的高级构造数据包的方法，就可简化构造数据包的过程，使构造数据包的效率大大提高。

Libnet 还提供了自动计算校验和的功能。在构造协议数据时，经常要进行校验和的计算，而这些计算都是比较烦琐的，而且也都是固定的。利用 Libnet 可以设置是否计算校验和，只需利用参数进行设置。

Libnet 支持多种平台操作，如 Linux，DOS，Windows 等。Libnet 屏蔽了不同操作系统的不同点，使不同平台下的操作过程都是一样的，这样提供给开发者的都是统一的开发接口。Libnet 提供了丰富的数据结构类型和输出函数。

Libnet 支持的协议非常多，几乎包括所有常用的网络协议。这样，利用 Libnet 可以构造各种各样的网络数据包和任意数据的网络数据包。利用 Libnet 构造数据包的过程是根据协议层次进行构造的，首先构造最高层的协议，然后再下一层，直到最低层协议。每个网络数据包的构造过程都是如此，所以掌握 Libnet 的使用方法是非常容易的。

Libnet 还支持二次开发，利用 Libnet 不仅可以构造 Libnet 支持的网络协议数据包，还可以构造不支持的网络协议数据包，这个时候就可以对 Libnet 进行二次开发，使它支持新的协议数据包。可以把构造新的协议数据包的功能加入 Libnet 中，扩充 Libnet 的功能。Libnet 提供了二次开发的接口，也就是编程规范，根据其规定的方法可以进行任意扩充。所以，使用 Libnet 可以构造丰富的协议数据包，而且可以进行二次开发，使它支持更多的协议类型。

基于 Libnet 开发的软件非常多，如 Ettercap，Firewalk，Snort，Tcpreplay 和 RING 等。其中，Ettercap 是一个交换网络环境下的网络监视、捕获和记录工具，Firewalk 是一个高级的 traceroute 工具，Snort 是一个轻便的网络入侵检测系统，Tcpreplay 能够重现网络数据流，RING 是指纹探测软件。

Libnet 的应用范围非常广泛，它可以应用在网络攻击程序、网络安全扫描系统、网络测试系统、网络入侵检测系统等网络安全系统中，只要系统需要构造数据包的功能，就可以利用 Libnet 来实现。

2.2.4 通用网络安全开发包 Libdnet

Libdnet 是一个通用的网络安全开发包和高级编程接口，它提供了很多非常有意义的功能。其中主要的功能有网络地址转接操作，ARP 缓存和路由表的查询和管理，网络防火墙管理，网络接口操作，IP 隧道操作，原始 IP 包和以太网帧的构造和发送等。

Libdnet 为开发者节省了时间和精力，它提供的功能在很多网络安全系统中都可以用到。例如，对于网络地址的转换操作，Libdnet 提供了对 IPv4 地址、以太网地址和 IPv6 地址的各种格式的相互转换。地址转换是所有网络安全程序要碰到的问题，Libdnet 提供了方便的转换函数。

Libdnet 可以对 ARP 缓存进行操作，如添加 ARP 缓存记录、删除 ARP 缓存记录、修改 ARP 缓存记录以及查询 ARP 缓存记录。ARP 协议是一个重要局域网协议，对 ARP 缓存的操作是各种操作系统提供的功能，Libdnet 提供了一个高级编程接口，可以对 ARP 缓存进行方便的操作。

Libdnet 对防火墙操作的功能对于开发者是非常方便的。有时开发者需要与防火墙打交道，如果防火墙没有提供接口就非常困难，即使有些防火墙提供了编程接口，但是不同的防火墙提供的接口是不同的，所以使用起来也非常麻烦。Libdnet 提供了操作防火墙的高级接口，其支持的防火墙主要有 ipfw，ipchains，pf 等。它提供的接口对于这些不同的防火墙都是一样的，也就是说可以用同样的方法来操作不同的防火墙系统。

Libdnet 对路由表进行操作的功能，包括查询路由表、插入路由表记录、修改路由表记录以及删除路由表记录等。

利用 Libdnet 对网络接口的操作功能，可以查看本机的网络接口，并且对其参数进行修改和设置操作，使用户对本机的网络接口的管理更加方便。因为在一般的网络程序中，都要对网络接口进行干预，如网络协议分析程序、网络防火墙等，所以 Libdnet 对网络接口的管理进行了封装。

Libdnet 还提供了构造和发送数据包的功能，可以利用 Libdnet 轻松地构造基于原始 IP 数据包和以太网帧两种类型的数据包。这样，就可以轻松地构造基于 IP 协议的任何其他协议，如 TCP，UDP，ICMP 等。Libdnet 比 Libnet 更加快捷，使用的步骤更少，但灵活性较差。在 Libdnet 中还实现了 IP 隧道功能。

Libdnet 的上述功能，提供给开发者的编程接口模式是一致的，都包括功能操作的打开、进行和关闭几部分。

使用 Libdnet 进行开发的软件非常多，如 arpscan，dhcp-agent，Firewalk，Fragroute，Ganglia，Honeyd，RING 等。其中，arpscan 是一个 ARP 监视器，dhcp-agent 是 UNIX DHCP 客户端，Firewalk 可以远程控制防火墙策略，Fragroute 是可以向目标截取、修改、重写数据包的系统，Ganglia 用于大规模计算机集群的实时监控和执行环境，Honeyd 是虚拟网络蜜罐系统，RING 是指纹探测软件。

由于 Libdnet 是一个通用网络安全开发包，包括的功能非常多，所以 Libdnet 的应用范围也是非常广泛的。例如，它可以应用于网络分析系统、网络防火墙系统、入侵检测系统、网络管理系统、网络测试系统、网络安全扫描系统等。

2.2.5 网络入侵检测开发包 Libnids

网络入侵检测开发包 Libnids（Library Network Intrusion Detection System）是一个用于网络入侵检测系统设计的专业开发包，提供了一个网络入侵检测系统的基本框架和基本功能。它可以进行 IP 碎片重组和 TCP 流重组，还提供了检测网络扫描的功能。利用 Libnids 可以快速地实现网络入侵检测的基本功能。

Libnids 是基于 Libpcap 和 Libnet 而开发的，所以具有它们的优点，以及跨平台、可移植、稳定、方便等特点。利用 Libnids 可以进行数据包的捕获，并对捕获到的数据包进行协议分析，当然也可以对数据包进行过滤，只捕获感兴趣的数据包。利用 Libnids 可以进行协议分析，如分析 IP、TCP、UDP 等协议。

Libnids 提供的 TCP 数据流重组功能是非常强大的，它可以对基于任何 TCP 协议的应用

层协议进行 TCP 数据流重组，显示它们的连接过程，并对它们传输的数据进行分析。利用 Libnids 提供的 TCP 数据流重组功能，可以实现对敏感数据的分析，如对用户、密码、账号进行嗅探。

利用 Libnids 提供的 TCP 数据流重组功能，还可以对网络传输的内容进行还原，重现网络数据。例如，可以利用 Libnids 的 TCP 数据流功能来分析 HTTP 协议，同时，对所传输的内容（如网页）进行分析。网络实际传输的是 HTML 文件内容，对其进行分析，就可以对网页进行重现。例如，可以对 POP3 和 SMTP 协议传输的邮件内容进行重现。

Libnids 是仿照 Linux 操作系统内核的网络协议栈来实现的，所以它提供的功能非常稳定、可靠，通过了很多测试过程。IP 碎片重组是 Libnids 提供的另外一个重要功能，利用它可以捕获所有的 IP 碎片数据包（包括异常数据包），当然也可以捕获正常的 IP 数据包。Libnids 不仅可以捕获异常 IP 数据包，也可以检测异常 TCP 数据包和异常 UDP 数据包。Libnids 还提供了端口扫描检测功能，能够检测 TCP 端口扫描攻击，并检测出 TCP 端口扫描类型。

Libndis 提供了一个开发框架，可以在此框架下面进行更深入的开发。它提供的这些基本功能，为开发者进行进一步的开发打下基础，便于开发者开发更多有针对性的功能。Libndis 的目的主要是用于入侵检测开发，所以利用 Libnids 可以快速构造入侵检测系统，并且在此基础上添加更多的入侵检测功能。

使用 Libnids 也可以用于其他网络安全系统，如网络分析系统、网络流量检测系统、网络管理系统等。

第 3 章　网络数据包捕获开发包 Libpcap

3.1　Libpcap 概述

3.1.1　Libpcap 简介

数据包捕获开发包 Libpcap（the Packet Capture Library）是一个平台独立的网络数据包捕获开发包。它是一个高层的编程接口，隐藏了操作系统的细节，可以捕获网络上的所有数据包，包括到达其他主机的数据包。Libpcap 中使用了 BPF 过滤机制，这部分是基于内核的过滤模块，它使 Libpcap 具有捕获特定数据包的功能，可以过滤掉网络上不需要的数据包，而只捕获用户感兴趣的数据包。使用 Libpcap 可以把从网络上捕获到的数据包存储到一个文件中，还可以把数据包信息从文件中读出，读出的结果与从网络上捕获数据包的结果是一样的。

Libpcap 是由 Berkeley 大学的 Van Jacobson，Craig Leres 和 Steven McCanne 编写的。该函数库支持 Linux, Solaris 和 BSD 系统平台。Libpcap 的应用非常广泛，几乎只要涉及到网络数据包的捕获功能，都可以使用 Libpcap 开发包。Libpcap 的主页是 www.tcpdump.org。

Libpcap 的开发语言是 C 语言，但也有其他语言支持的 Libpcap 开发包。例如，支持 Python 语言的开发包 pycap, py-libpcap/py-libpcap-win32，还有支持 Java 语言的开发包 Jpcap，支持 Perl 语言的 Net::Pcap, Net::Pcap for Win32，支持 Visual Basic 的 PacketVB，等等。

使用 Libpcap 开发的程序有很多，在本书 2.2.1 节已简单介绍过，在此不再重复。

目前，Libpcap 的最新稳定版本为 0.8.3，最新测试版本为 0.9.0。Libpcap 开发包的开发非常活跃，版本在不断更新，读者应该经常关注，检查一下是否有新的版本发布。Libpcap 的官方网址为：www.tcpdump.org，可以在此网站上下载。

本章以 Libpcap 最新稳定版本 0.8.3 进行介绍。

3.1.2　Libpcap 的作用

Libpcap 的最主要的作用就是捕获网络数据包。它把捕获网络数据包的功能进行了封装，使其使用起来非常方便。在捕获数据包的过程中，可以对捕获模式进行限制。例如，可以设置捕获的过滤规则，只捕获感兴趣的网络数据包。还可以对捕获的对象进行限制，而且限制的精度很高，灵活性非常好。

利用 Libpcap 捕获数据包的功能，可以实现其他很多的功能。例如，对网络数据包进行分析，读出所有网络数据包的详细信息。有名的软件 tcpdump 就是完成此项功能的。当然，还可以做很多其他的事情，如网络流量分析、网络入侵检测、网络安全监控等。这些都是在捕获数据包的基础上进行操作的，所以都可以使用 Libpcap 来实现。例如，著名的网络入侵检测系统 Snort 就是使用 Libpcap 来实现的。

总结起来，Libpcap 的作用主要有以下三个方面。

1. 捕获各种网络数据包

开发包 Libpcap 的最大功能就是捕获网络数据包，它是一个专业的网络数据包捕获函数库。使用 Libpcap 可以方便地对网络上的数据包进行捕获，且操作过程简单，效率也非常高。使用 Libpcap 捕获网络上的数据包之后，就可以对捕获到的数据包进行各种各样的分析。例如，对它进行协议分析，或者进一步进行网络流量统计，以及在协议分析的基础上进行入侵检测分析等。

2. 过滤网络数据包

在 Libpcap 使用的 BPF 捕获机制中，有一个网络数据包过滤模块，使用它可以过滤各种各样的网络数据包。它使用了一个非常简单的过滤规则，该规则与 Tcpdump 的过滤规则是一样的。由于过滤模块是在内核层次中实现的，所以效率非常高，而且 BPF 过滤规则还进行了优化处理。

3. 分析网络数据包

使用 Libpcap 捕获数据包后，可以对数据包进行各种各样的分析操作。另外，在 Libpcap 中提供了很多辅助功能，帮助开发者实现对网络数据包的捕获。虽然使用 Libpcap 捕获的数据包是一个字节流信息，但是也提供了此信息的基本信息，如捕获的时间、数据包的长度等。开发者可以在此基础上进一步分析数据包的内容，这也是开发者所要做的主要事情。

4. 存储网络数据包

在从网络上捕获数据包时，可以不马上对其进行分析，而只是存储起来，留到以后进行分析。开发包 Libpcap 提供了存储网络数据包的功能，它把从网络上捕获到的网络数据包存储到文件之中，然后使用 Libpcap 从文件中读取数据包的内容，以使进行进一步分析。

3.1.3 Libpcap 的安装

把下载的 Libpcap 开发包拷入一个文件夹下，然后就可以安装 Libpcap 开发包。很多 Linux 版本下就有 Libpcap 开发包，在安装 Linux 操作系统时，已经安装了 Libpcap 开发包，但是它可能不是最新的版本，所以最好还是重新安装一遍。

安装 Libpcap 的过程大概有四个步骤，这也是很多软件在 Linux 下的一般安装步骤。这些步骤如下。

1. 解压缩 Libpcap 开发包

使用命令 tar 进行解压缩，tar 命令是 Linux 下的一个压缩和解压缩命令。具体解压缩命令如下：

```
tar -xzvf libpcap-0.8.3.tar.gz
```

其操作过程如图 3-1 所示。

2. 配置

在步骤 1 中，解压缩之后会生成一个目录，如“libpcap-0.8.3”。使用命令“cd”进入此目录，然后对 Libpcap 进行配置，配置命令如下：

```
./configure
```

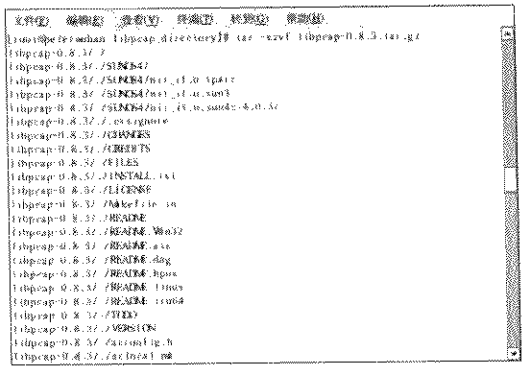

图 3-1 Libpcap 开发包的解压缩

其操作过程如图 3-2 所示。

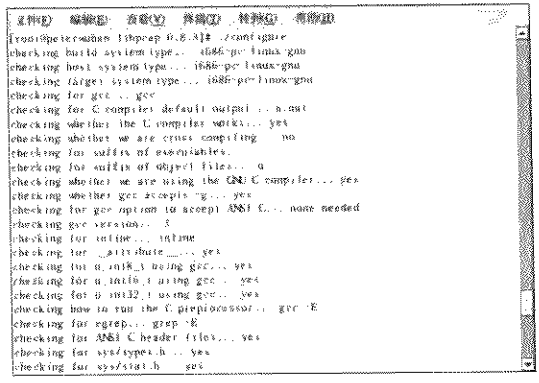

图 3-2 对 Libpcap 进行配置

3. 编译

配置成功之后，对 Libpcap 开发包进行编译，这一步消耗的时间要长一点。具体的编译操作是执行 make 命令：

```
make
```

其操作过程如图 3-3 所示。

4. 安装

如果在编译的过程中没有出现错误，就可以执行最后一步—— 安装过程。安装时执行 make install 命令：

```
make install
```

具体安装过程如图 3-4 所示。

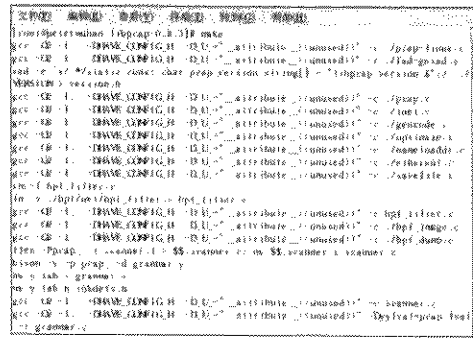

图 3-3　对 Libpcap 进行编译

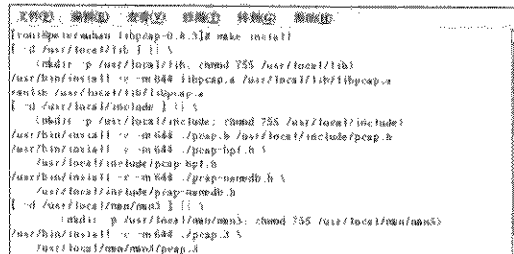

图 3-4　对 Libpcap 进行安装

安装成功后，就可以在程序中使用 Libpcap 开发包。在具体使用时，一般使用编译参数"-l"表示使用的是什么库。这里使用的是 Libpcap 开发包的库，它的库名为 pcap，所以在使用 Libpcap 开发程序时，使用的命令应该是"-lpcap"。举一个例子，假设源文件名字为 test.c，编译后的文件为 test，那么编译命令如下：

```
gcc -o test test.c -lpcap
```

其中"-o"表示生成二进制文件 test。

3.2　Libpcap 的组成

3.2.1　BPF 捕获机制

伯克利数据包过滤器（BPF，Berkeley Packet Filter）是 Libpcap 使用的捕获机制，它是一个高效的数据包捕获机制。

BPF 的架构如图 3-5 所示。

BPF 主要由两部分组成，一是网络转发部分，二是数据包过滤部分。网络转发部分从链路层中捕获数据包，并把它们转发给数据包过滤部分；数据包过滤部分从接收到的数据包中接收过滤规则决定的网络数据包，其他的数据包就被抛弃。

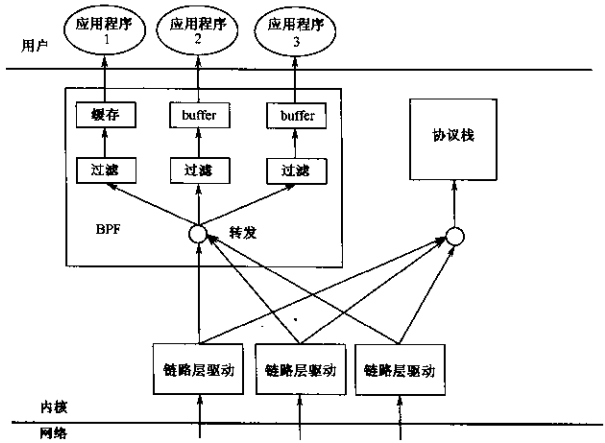

图 3-5　BPF 的架构

BFP 的这两部分都是在操作系统内核层实现的，它提供给应用层的数据包是过滤后的数据包，所以捕获数据包和过滤数据包都是在内核中完成的，效率是非常高的。另外，BPF 使用了缓存机制，使捕获数据包缓存在内核中达到一定量时就传递给应用程序，这样也提高了处理的效率。

3.2.2　过滤规则

在 BPF 过滤机制中，其过滤规则是比较重要的，掌握它对捕获网络数据包很有好处。

BPF 过滤规则在很多软件中用到，最著名的程序是 Tcpdump。当然还有很多其他应用程序也使用 BPF 过滤规则，如 Ethereal, Analyzer, Snort 等。只要是基于 Libpcap 开发包开发的应用程序，都使用 BPF 过滤规则。

BPF 过滤规则是一个字符串，其定义形式有一定的规律性。它主要由两类数据组成，一是标识，二是标识的修饰词。修饰词有三种，分别为类型、方向和协议。

类型修饰词表示标识的种类。有 3 种类型修饰词：host，net 和 port。host 代表主机类型，net 代表网络类型，port 代表端口类型。

方向修饰词表示数据的传输方向。有 4 种方向修饰词：src，dst，src or dst 和 src and dst。src 表示源地址，即此数据包是从哪个主机发送的；dst 表示目的地址，即此数据包到达哪个主机；src or dst 表示无方向；src and dst 表示源地址和目的地址都必须满足。

协议修饰词表示数据包使用的协议类型，主要有 ether，fddi，ip，arp，rarp，decnet，lat，sca，moprc，mopdl，tcp 和 udp。它们分别代表不同的协议。

修饰词和标识就构成了原语。过滤规则就是由一个或多个原语组成的。它们之间可以使用连接词来进行连接，连接词主要有 and，or 和 not。and 表示“并且”的意思，表示两者都必须满足。例如，“src 192.168.0.2 and port 30”表示捕获源地址为 192.168.0.2 并且端口号为 30 的网络数据包。or 表示“或者”的意思，表示只要满足其一即可。例如，“src 192.168.0.2 or dst 192.168.0.3”表示捕获源地址为 192.168.0.2 或者目的地址为 192.168.0.3 的数据包。not

表示“非”的意思，表示否定。例如，“not dst 192.168.0.3”表示捕获所有目的地址不是192.168.0.3的数据包。

下面对一些比较常用的原语进行具体介绍，更详细的内容请查看Tcpdump帮助文档。

- src host hostname —— 表示所捕获源地址符合 hostname 的网络数据包。此时的hostname 可以是主机名字也可以是网络地址。例如，“src host 192.168.0.2”表示捕获源地址为192.168.0.2的数据包。
- dst host hostname —— 表示所捕获目的地址符合 hostname 的网络数据包。此时的hostname 可以是主机名字也可以是网络地址，例如“dst host 192.168.0.4”，表示捕获目的地址为192.168.0.4的数据包。
- host hostname —— 表示捕获的目的地址或者源地址是 hostname 的网络数据包。此时的 hostname 可以是主机名字也可以是网络地址。例如，“host 192.168.0.2”表示捕获的目的地址或者主机地址是192.168.0.2的网络数据包。
- ip src host hostname —— 表示捕获的源地址是 hostname 的 IP 协议数据包。例如，“ip src host 192.168.0.2”表示捕获的源地址是192.168.0.2的IP协议数据包。
- Ip dst host hostname —— 表示捕获的目的地址是 hostname 的 IP 协议数据包。例如，“ip dst host 192.168.0.3”表示所捕获目的地址是192.168.0.3的IP协议数据包。
- ether dst macaddress —— 表示所捕获目的以太网地址是 macaddress 的网络数据包。例如，“ether dst 11:11:11:11:11:11”表示所捕获目的以太网地址是 11:11:11:11:11:11的网络数据包。
- ether src macaddress —— 表示所捕获源以太网地址是 macaddress 的网络数据包。例如，“ether src 22:22:22:22:22:22”表示所捕获源以太网地址是22:22:22:22:22:22的网络数据包。
- ether host macaddress —— 表示所捕获源以太网地址和目的以太网地址都是macaddress 的网络数据包。例如，“ether host 11:11:11:11:11:11”，表示所捕获源以太网地址或者目的以太网地址都是11:11:11:11:11:11的网络数据包。
- dst net netname —— 表示所捕获目的网络地址是 netname 的网络数据包。此时的netname 可以是名字，也可以是网络地址。
- src net netname —— 表示所捕获源网络地址是 netname 的网络数据包。同样，此时的 netname 可以是名字，也可以是网络地址。
- net netname —— 表示所捕获源网络地址和目的网络地址是 netname 的网络数据包。同样，此时的 netname 可以是名字，也可以是网络地址。
- dst port portnumber —— 表示所捕获目的端口号是 portnumber 的网络数据包。此时的 portnumber 可以是数字，也可以是协议名字。例如，“dst port 80”表示捕获所有目的端口号是80的网络数据包。
- src port portnumber —— 表示所捕获源端口号是 portnumber 的网络数据包。同样，此时的 portnumber 可以是数字，也可以是协议名字。例如，“src port 4213”表示所捕获源端口号是4213的网络数据包。
- port portnumber —— 表示所捕获源端口号或目的端口号是 portnumber 的网络数据包。同样，此时的 portnumber 可以是数字，也可以是协议名字。例如，“port 1268”，表示所捕获源端口号或目的端口号是1268的网络数据包。

- tcp src port portnumber —— 表示所捕获源端口号是 portnumber 的 TCP 协议数据包。例如，“tcp src port 1234”表示所捕获的源端口号是 1234 的 TCP 协议数据包。
- tcp dst port portnumber —— 表示所捕获目的端口号是 portnumber 的 TCP 协议数据包。例如，“tcp dst port 4321”表示所捕获目的端口号是 4321 的 TCP 协议数据包。
- udp src port portnumber —— 表示所捕获源端口号是 portnumber 的 UDP 协议数据包。例如，“udp src port 1111”表示所捕获源端口号是 1111 的 UDP 协议数据包。
- udp dst port portnumber —— 表示所捕获目的端口号是 portnumber 的 UDP 协议数据包。例如，“udp dst port 2222”表示所捕获目的端口号是 2222 的 UDP 协议数据包。

3.2.3 网卡设置

在 Libpcap 中对网卡的设置有两种状态，一是混杂模式，二是非混杂模式。

如果设置成混杂模式，那么 Libpcap 可以捕获局域网中的所有数据包，包括局域网中的每台机器的进出网络数据包，当然也包括本机的进出网络数据包。

如果设置成非混杂模式，那么 Libpcap 只捕获本机的进出网络数据包。

对网卡的设置非常简单，主要由函数 pcap_open_live 来完成，详细信息请参阅 3.4.1 节。

3.2.4 文件

Libpcap 的文件主要由一系列的源文件和头文件构成。其源文件有：

- ethernet.c；
- bpf_dump.c；
- bpf_image.c；
- fad-getad.c；
- fad-gifc.c；
- fad-glifc.c；
- fad-null.c；
- fad-win32.c；
- gencode.c；
- inet.c；
- nametoaddr.c；
- optimize.c；
- pcap.c；
- pcap-bpf.c；
- pcap-dag.c；
- pcap-dlpi.c；
- pcap-enet.c；
- pcap-liuux.c；
- pcap-linux.c；
- pcap-nit.c；
- pcap-null.c；

```
#ifdef __cplusplus
extern "C" {
#endif
#define PCAP_VERSION_MAJOR 2
#define PCAP_VERSION_MINOR 4
#define PCAP_ERRBUF_SIZE 256
#if BPF_RELEASE - 0 < 199406
typedef  int bpf_int32;
typedef  u_int bpf_u_int32;
#endif
typedef struct pcap pcap_t;                          /*pcap_t 的定义*/
typedef struct pcap_dumper pcap_dumper_t;            /*pcap_dumper_t 的定义*/
typedef struct pcap_if pcap_if_t;                    /*pcap_if_t 的定义*/
typedef struct pcap_addr pcap_addr_t;                /*pcap_addr_t 的定义*/
struct pcap_file_header {                            /*pcap 文件的数据结构定义*/
    bpf_u_int32 magic;
    u_short version_major;                           /*主版本号*/
    u_short version_minor;                           /*次版本号*/
    bpf_int32 thiszone;
    bpf_u_int32 sigfigs;
    bpf_u_int32 snaplen;
    bpf_u_int32 linktype;
};
struct pcap_pkthdr {                                 /*协议头的数据结构定义*/
    struct timeval ts;
    bpf_u_int32 caplen;
    bpf_u_int32 len;
};
struct pcap_stat {                                   /*pcap 统计的数据结构定义*/
    u_int ps_recv;
    u_int ps_drop;
    u_int ps_ifdrop;
#ifdef WIN32
    u int bs_capt;
#endif
};
struct pcap_if {                                     /*网络接口的数据结构的定义*/
    struct pcap_if *next;
    char *name;
    char *description;
    struct pcap_addr *addresses;
    bpf_u_int32 flags;
};
```

```
#define PCAP_IF_LOOPBACK    0x00000001
struct pcap_addr {                                  /*地址的定义*/
    struct pcap_addr *next;
    struct sockaddr *addr;
    struct sockaddr *netmask;
    struct sockaddr *broadaddr;
    struct sockaddr *dstaddr;
};
typedef void (*pcap_handler)(u_char *, const struct pcap_pkthdr *, const u_char *);
/*回调函数的类型定义*/
/*以下是Libpcap的输出函数的声明*/
char  *pcap_lookupdev(char *);
int  pcap_lookupnet(const char *, bpf_u_int32 *, bpf_u_int32 *, char *);
pcap_t  *pcap_open_live(const char *, int, int, int, char *);
pcap_t  *pcap_open_dead(int, int);
pcap_t  *pcap_open_offline(const char *, char *);
void  pcap_close(pcap_t *);
int  pcap_loop(pcap_t *, int, pcap_handler, u_char *);
int  pcap_dispatch(pcap_t *, int, pcap_handler, u_char *);
const u_char*
    pcap_next(pcap_t *, struct pcap_pkthdr *);
int  pcap_next_ex(pcap_t *, struct pcap_pkthdr **, const u_char **);
void  pcap_breakloop(pcap_t *);
int  pcap_stats(pcap_t *, struct pcap_stat *);
int  pcap_setfilter(pcap_t *, struct bpf_program *);
int  pcap_getnonblock(pcap_t *, char *);
int  pcap_setnonblock(pcap_t *, int, char *);
void  pcap_perror(pcap_t *, char *);
char  *pcap_strerror(int);
char  *pcap_geterr(pcap_t *);
int  pcap_compile(pcap_t *, struct bpf_program *, char *, int, bpf_u_int32);
int  pcap_compile_nopcap(int, int, struct bpf_program *, char *, int, bpf_u_int32);
void  pcap_freecode(struct bpf_program *);
int  pcap_datalink(pcap_t *);
int  pcap_list_datalinks(pcap_t *, int **);
int  pcap_set_datalink(pcap_t *, int);
int  pcap_datalink_name_to_val(const char *);
const char *pcap_datalink_val_to_name(int);
const char *pcap_datalink_val_to_description(int);
int  pcap_snapshot(pcap_t *);
int  pcap_is_swapped(pcap_t *);
int  pcap_major_version(pcap_t *);
int  pcap_minor_version(pcap_t *);
```

```
FILE *pcap_file(pcap_t *);
Int  pcap_fileno(pcap_t *);
pcap_dumper_t *pcap_dump_open(pcap_t *, const char *);
int  pcap_dump_flush(pcap_dumper_t *);
void  pcap_dump_close(pcap_dumper_t *);
void  pcap_dump(u_char *, const struct pcap_pkthdr *, const u_char *);
FILE  *pcap_dump_file(pcap_dumper_t *);
Int  pcap_findalldevs(pcap_if_t **, char *);
Void  pcap_freealldevs(pcap_if_t *);
const char *pcap_lib_version(void);
u_int  bpf_filter(struct bpf_insn *, u_char *, u_int, u_int);
int  bpf_validate(struct bpf_insn *f, int len);
char  *bpf_image(struct bpf_insn *, int);
void  bpf_dump(struct bpf_program *, int);
#ifdef WIN32
int pcap_setbuff(pcap_t *p, int dim);
int pcap_setmode(pcap_t *p, int mode);
int pcap_sendpacket(pcap_t *p, u_char *buf, int size);
int pcap_setmintocopy(pcap_t *p, int size);
#ifdef WPCAP
#include <Win32-Extensions.h>
#endif
#define MODE_CAPT 0
#define MODE_STAT 1
#define MODE_MON 2
#else
int  pcap_get_selectable_td(pcap_t *);
#endif
#ifdef __cplusplus
}
#endif
#endif
```

3.3 Libpcap 数据结构

3.3.1 pcap

pcap 数据结构是 Libpcap 的核心数据结构，其定义如下：

```
struct pcap {
#ifdef WIN32
    ADAPTER *adapter;            /*网络接口*/
    LPPACKET Packet;
    int timeout;                 /*时间*/
    int nonblock;                /*非阻塞模式参数*/
```

```
#else
    int fd;
    int selectable_fd;
#endif /* WIN32 */
    int snapshot;
    int linktype;                /*链路层类型*/
    int tzoff;                   /*时间域 */
    int offset;                  /* 偏移*/
    int break_loop;              /* 循环结束标志*/
    struct pcap_sf sf;
    struct pcap_md md;
    int bufsize;                 /*缓冲区大小*/
    u_char *buffer;              /*缓冲区*/
    u_char *bp;
    int cc;
    u_char *pkt;
    int  (*read_op)(pcap_t *, int cnt, pcap_handler, u_char *);
    int  (*setfilter_op)(pcap_t *, struct bpf_program *);
    int  (*set_datalink_op)(pcap_t *, int);
    int  (*getnonblock_op)(pcap_t *, char *);
    int  (*setnonblock_op)(pcap_t *, int, char *);
    int  (*stats_op)(pcap_t *, struct pcap_stat *);
    void (*close_op)(pcap_t *);
    struct bpf_program fcode;
    char errbuf[PCAP_ERRBUF_SIZE + 1];          /*错误信息*/
    int dlt_count;                              /*链路层类型个数*/
    int *dlt_list;                              /*链路层类型列表*/
    struct pcap_pkthdr pcap_header;
};
typedef struct pcap pcap_t;
```

此数据结构表示的是 Libpcap 句柄的数据结构，是 Libpcap 的内部数据结构，在 Libpcap 内部实现的过程中要用到。程序员不会与它直接打交道，而一般是通过函数来操作的。

3.3.2 pcap_file_header

pcap_file_header 数据结构定义如下：

```
struct pcap_file_header {
        bpf_u_int32 magic;
        u_short version_major;
        u_short version_minor;
        bpf_int32 thiszone;
        bpf_u_int32 sigfigs;
        bpf_u_int32 snaplen;
        bpf_u_int32 linktype;
```

```
};
```

此数据结构用来描述一个 Libpcap 存储文件类型。不同的 Libpcap 存储形式使用不同的存储文件类型，一个存储文件类型用惟一值来表示：

- 成员 magic 是一个 32 位的整数值，代表一个存储文件类型，不同的存储文件类型有不同的 magic 数值；
- 成员 version_major 和 version_minor 用来表示版本号；
- 成员 thiszone 用来表示格林尼治标准时间，用来修正当地区域的时间；
- 成员 sigfigs 用来表示时间戳；
- 成员 snaplen 用来表示数据包的最大捕获长度；
- 成员 linktype 用来表示数据链路层的类型。

3.3.3 pcap_pkthdr

pcap_pkthdr 的数据结构的定义如下：

```
struct pcap_pkthdr {
        struct timeval ts;
        bpf_u_int32 caplen;
        bpf_u_int32 len;
};
```

此数据结构用来描述每个捕获到的数据包的一些基本信息。每个数据包都有此数据结构。

- 成员 timevalts 表示数据包的时间戳；
- 成员 caplen 表示捕获的长度；
- 成员 len 表示数据包的长度。

3.3.4 pcap_stat

pcap_stat 数据结构的定义如下：

```
struct pcap_stat {
        u_int ps_recv;
        u_int ps_drop;
        u_int ps_ifdrop;
    #ifdef WIN32
        u_int bs_capt;
    #endif
};
```

此数据结构主要是描述的 Libpcap 的状态信息。

- 成员 ps_recv 表示的是捕获到的数据包的个数；
- 成员 ps_drop 表示的是丢失的数据包的个数；
- 成员 ps_ifdrop 未用；
- 成员 bs_capt 表示的是数据包到达应用层的个数。

3.3.5 pcap_if

pcap_if 数据结构的定义如下：

```
typedef struct pcap_if pcap_if_t;
struct pcap_if {
        struct pcap_if *next;
        char *name;
        char *description;
        struct pcap_addr *addresses;
        bpf_u_int32 flags;
};
```

此数据结构描述的是一个网络接口，它实际上是网络接口链表中的一个结点。一个结点就描述了一个网络接口，当网络接口比较多的时候，就把它们组成一个链表，用此链表来存储所有的网络接口。

- 成员 next 表示指向下一个网络接口结点；
- 成员 name 表示的是网络接口的名字句柄，主要用在函数 pcap_open_live()中，其定义请参阅 3.4 节；
- 成员 description 表示的是网络接口的文字描述；
- 成员 addresses 表示的是此网络接口的地址；
- 成员 flags 表示的是网络接口标记。

3.3.6 pcap_addr

pcap_addr 数据结构的定义如下：

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

此数据结构描述的是网络接口的地址，其中：

- 成员 next 表示指向下一个地址结点；
- 成员 addr 表示网络接口地址，其类型为 struct sockaddr；
- 成员 netmask 表示的是地址掩码；
- 成员 broadaddr 表示广播地址；
- 成员 dstaddr 表示的是点对点目的地址。

3.4 Libpcap 函数

3.4.1 网络接口函数

（1）int pcap_findalldevs(pcap_if_t **alldevsp, char *errbuf)

函数返回值：如果函数操作成功，就返回 0；如果失败，就返回－1。

参数描述：参数 alldevsp 表示网络接口链表，参数 errbuf 用来存储错误信息。

此函数的功能是查找机器的所有可用的网络接口，用一个网络接口链表返回。

（2）void pcap_freealldevs(pcap_if_t *alldevs)

函数返回值：无。

参数描述：参数 alldevs 表示一个网络接口链表。

此函数的功能是释放网络接口链表中的所有网络接口。

（3）char * pcap_lookupdev(char *errbuf);

函数返回值：返回网络接口的名字。

参数描述：参数 errbuf 表示错误信息。

此函数的功能是获得本机的网络接口。

（4）int pcap_lookupnet(register const char *device,register bpf_u_int32 *netp, register bpf_u_int32 *maskp,register char *errbuf)

函数返回值：函数操作成功就返回 0，失败就返回－1。

参数描述：参数 device 是网络接口名字，参数 netp 是网络地址，参数 maskp 表示网络掩码，参数 errbuf 表示错误信息。

此函数的功能是获取网络地址和网络掩码。

（5）pcap_t * pcap_open_live(const char *device, int snaplen, int promisc, int to_ms,char *ebuf)

函数返回值：返回一个 Libpcap 句柄。

参数描述：参数 device 表示网络接口名字；参数 snaplen 表示捕获数据包的长度；参数 promisc 表示是否设置混杂模式，如果赋值为 1，就表示设置为混杂模式；参数 to_ms 表示等待的时间；参败 ebuf 表示存储错误信息。

此函数的功能是打开一个网络接口进行数据包捕获。打开的模式由 promisc 表示，如果是 1，就表示以混杂模式打开此网络接口；否则，以非混杂模式打开网络接口。

3.4.2 规则函数

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

函数返回值：函数操作成功就返回 0，失败就返回－1。

参数描述：参数 p 表示 Libnet 句柄，参数 fp 表示 BPF 过滤规则。

此函数的功能是设置 BPF 过滤规则，由参数 fp 确定。

（2）int pcap_compile(pcap_t *p, struct bpf_program *program,char *buf, int optimize, bpf_u_int32 mask)

函数返回值：函数操作成功则返回 0，失败则返回－1。

参数描述：参数 p 表示 Libpcap 句柄，参数 program 表示 bpf 过滤规则，参数 buf 表示

过滤规则字符串，参数 optimize 表示优化，参数 mask 表示掩码。

此函数的功能是编译 BPF 过滤规则。

（3）int pcap_compile_nopcap(int snaplen_arg, int linktype_arg,struct bpf_program *program, char *buf, int optimize, bpf_u_int32 mask)

函数返回值：函数操作成功则返回 0，失败则返回－1。

参数描述：参数 snaplen 表示长度，参数 linktype_arg 表示链路类型，参数 program 表示 bpf 过滤规则，参数 buf 表示过滤规则字符串，参数 optimize 表示优化，参数 mask 表示掩码。

此函数的功能也是编译 BPF 过滤规则。

（4）void pcap_freecode(struct bpf_program *program)

函数返回值：无。

参数描述：参数 program 表示 BPF 过滤规则。

此函数的功能是释放过滤规则的内存。

3.4.3 数据包捕获函数

（1）int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user)

函数返回值：函数操作成功则返回 0，失败则返回负数。

参数描述：参数 p 表示 Libpcap 句柄；参数 cnt 表示捕获数据包的个数，如果是－1，就表示捕获个数为无限个；参数 callback 表示回调函数；参数 user 表示向回调函数中传输的参数。

此函数是循环捕获网络数据包，直到遇到错误或满足退出条件。每捕获一个数据包就调用 callback 指示的回调函数。所以，可以在回调函数中对捕获到的数据包进行操作。

（2）int pcap_dispatch(pcap_t *p, int cnt, pcap_handler callback, u_char *user)

函数返回值：函数操作成功则返回数据包个数，失败则返回负数。

参数描述：参数 p 表示 Libpcap 句柄，参数 cnt 表示捕获数据包的个数，参数 callback 表示回调函数，参数 user 表示向回调函数传递的参数。

此函数的功能是捕获网络数据包。

（3）const u_char * pcap_next(pcap_t *p, struct pcap_pkthdr *h)

函数返回值：返回一个数据包的缓冲数据。

参数描述：参数 p 表示 Libpcap 句柄，参数 h 表示数据包头。

此函数的功能是捕获一个网络数据包，然后返回此数据包。

（4）int pcap_next_ex(pcap_t *p, struct pcap_pkthdr **pkt_header, const u_char **pkt_data)

函数返回值：函数操作成功就返回 1，如果超时就返回 0，失败则返回－1 或－2。

参数描述：参数 p 表示 Libpcap 句柄，参数 pkt_header 表示数据包头，参数 pkt_data 表示数据包。

此函数的功能是捕获一个网络数据包。

（5）void pcap_breakloop(pcap_t *p)

函数返回值：无。

参数描述：参数 p 表示 Libpcap 句柄。

此函数的功能是退出循环捕获数据包状态。

参数描述：参数 p 表示 Libpcap 文件句柄。
此函数的功能是返回一个标准的文件句柄。

3.4.5 错误处理函数

（1）void pcap_perror(pcap_t *p, char *prefix)
函数返回值：无。
参数描述：参数 p 表示 Libpcap 句柄，参数 prefix 表示错误指针。
此函数的功能是获取最后一个错误信息。
（2）char * pcap_strerror(int errnum)
函数返回值：返回错误信息。
参数描述：参数 emum 表示错误标识。
此函数的功能是获取相应的错误信息。
（3）char * pcap_geterr(pcap_t *p)
函数返回值：返回错误信息。
参数描述：参数 p 表示 Libpcap 句柄。
此函数的功能是获取最后一个错误信息。

3.4.6 辅助函数

（1）int pcap_stats(pcap_t *p, struct pcap_stat *ps)
函数返回值：函数操作成功返回 0，失败返回－1。
参数描述：参数 p 表示 Libpcap 句柄，参数 ps 表示统计信息。
此函数的功能是获得统计信息。
（2）int pcap_datalink(pcap_t *p)
函数返回值：返回一个链路层类型。
参数描述：参数 p 表示 Libpcap 句柄。
此函数的功能是获取链路层状态。例如，如果类型为 DLT_EN10MB，就表示以太网。
（3）int pcap_list_datalinks(pcap_t *p, int **dlt_buffer)
函数返回值：函数操作成功就返回数据包个数，失败就返回－1。
参数描述：参数 p 表示 Libpcap 句柄，参数 dlt_buffer 表示一个链路层类型的链表。
此函数的功能是返回支持的所有链路层类型的链表。
（4）int pcap_set_datalink(pcap_t *p, int dlt)
函数返回值：函数操作成功就返回 0，失败就返回－1。
参数描述：参数 p 表示 Libpcap 句柄，参数 dlt 表示链路层类型。
此函数的功能是设置链路层类型。
（5）int pcap_datalink_name_to_val(const char *name)
函数返回值：函数操作成功则返回链路层类型值，失败则返回－1。
参数描述：参数 name 表示数据链路层的名字。
此函数的功能是把链路层类型对应的名字转换成标识。
（6）const char * pcap_datalink_val_to_name(int dlt)
函数返回值：操作成功则返回链路层标识，失败则返回－1。

参数描述：参数 dlt 表示链路层标识。

此函数的功能是把链路层标识转换成对应的链路层名字字符串。

（7）const char * pcap_datalink_val_to_description(int dlt)

函数返回值：函数操作成功就返回链路层描述，失败就返回 NULL。

参数描述：参数 dlt 表示链路层标识。

此函数的功能是把链路层类型的描述求出来。在 Libpcap 中有如下一个数组，其中设置的就是每个链路层类型的描述形式：

```
static struct dlt_choice dlt_choices[] = {
    DLT_CHOICE(DLT_NULL, "BSD loopback"),
    DLT_CHOICE(DLT_EN10MB, "Ethernet"),
    DLT_CHOICE(DLT_IEEE802, "Token ring"),
    DLT_CHOICE(DLT_ARCNET, "ARCNET"),
    DLT_CHOICE(DLT_SLIP, "SLIP"),
    DLT_CHOICE(DLT_PPP, "PPP"),
    DLT_CHOICE(DLT_FDDI, "FDDI"),
    DLT_CHOICE(DLT_ATM_RFC1483, "RFC 1483 IP-over-ATM"),
    DLT_CHOICE(DLT_RAW, "Raw IP"),
    DLT_CHOICE(DLT_SLIP_BSDOS, "BSD/OS SLIP"),
    DLT_CHOICE(DLT_PPP_BSDOS, "BSD/OS PPP"),
    DLT_CHOICE(DLT_ATM_CLIP, "Linux Classical IP-over-ATM"),
    DLT_CHOICE(DLT_PPP_SERIAL, "PPP over serial"),
    DLT_CHOICE(DLT_PPP_ETHER, "PPPoE"),
    DLT_CHOICE(DLT_C_HDLC, "Cisco HDLC"),
    DLT_CHOICE(DLT_IEEE802_11, "802.11"),
    DLT_CHOICE(DLT_FRELAY, "Frame Relay"),
    DLT_CHOICE(DLT_LOOP, "OpenBSD loopback"),
    DLT_CHOICE(DLT_ENC, "OpenBSD encapsulated IP"),
    DLT_CHOICE(DLT_LINUX_SLL, "Linux cooked"),
    DLT_CHOICE(DLT_LTALK, "Localtalk"),
    DLT_CHOICE(DLT_PFLOG, "OpenBSD pflog file"),
    DLT_CHOICE(DLT_PRISM_HEADER, "802.11 plus Prism header"),
    DLT_CHOICE(DLT_IP_OVER_FC, "RFC 2625 IP-over-Fibre Channel"),
    DLT_CHOICE(DLT_SUNATM, "Sun raw ATM"),
    DLT_CHOICE(DLT_IEEE802_11_RADIO, "802.11 plus BSD radio information header"),
    DLT_CHOICE(DLT_APPLE_IP_OVER_IEEE1394, "Apple IP-over-IEEE 1394"),
    DLT_CHOICE(DLT_ARCNET_LINUX, "Linux ARCNET"),
    DLT_CHOICE(DLT_LINUX_IRDA, "Linux IrDA"),
    DLT_CHOICE(DLT_IEEE802_11_RADIO_AVS, "802.11 plus AVS radio information header"),
    DLT_CHOICE_SENTINEL
};
```

（8）int pcap_snapshot(pcap_t *p)

函数返回值：返回长度。

参数描述：参数 p 表示 Libpcap 句柄。
此函数的功能是返回 Libpcap 句柄中的 snapshot 数值。
（9）int　pcap_is_swapped(pcap_t *p)
函数返回值：如果是错误的文件类型，就返回 1；否则，返回 0。
参数描述：参数 p 表示 Libpcap 句柄。
此函数的功能是返回文件类型是否正确的信息。
（10）int　pcap_major_version(pcap_t *p)
函数返回值：返回版本号。
参数描述：参数 p 表示 Libpcap 句柄。
此函数的功能是返回文件的主版本号。
（11）int　pcap_minor_version(pcap_t *p)
函数返回值：返回版本号。
参数描述：参数 p 表示 Libpcap 句柄。
此函数的功能是返回文件的次版本号。
（12）const char *　pcap_lib_version(void)
函数返回值：返回版本信息。
参数描述：无。
此函数的功能是返回 Libpcap 的版本信息。
（13）int　pcap_getnonblock(pcap_t *p, char *errbuf)
函数返回值：函数操作成功就返回 0，失败就返回 -1。
参数描述：参数 p 表示 Libpcap 句柄，参数 errbuf 表示错误信息。
此函数的功能是返回 Libpcap 的非阻塞状态。
（14）int　pcap_setnonblock(pcap_t * p, int nonblock, char * errbuf)
函数返回值：函数操作成功就返回 0，失败就返回 -1。
参数描述：参数 p 表示 Libpcap 句柄，参数 errbuf 表示错误信息。
此函数的功能是设置 Libpcap 是否为非阻塞模式。

3.5　Libpcap 的使用

Libpcap 开发包是本书所讨论的第一个开发包，后面还要讨论很多其他开发包。在讨论每个开发包时都要对开发包的使用方法进行详细阐述，并且都是通过举例进行分析的。

注意，在本章以及后面章节的所有例子中，为了不使程序过于冗长，而使它短小、简洁，更容易理解，程序中省略了很多错误处理操作，因为举例的目的只是为了说明开发包的使用方法，而不过多涉及其他的东西。当然，在实际的应用程序设计中，应该进行错误处理，这样使程序更健壮、更容易维护。

在所举的例子中，大部分语句都有比较详细的注释，对整个程序的解释也基本上包含在程序的注释中，所以仔细阅读每个程序的注释部分，对于理解整个例程是很有帮助的。在这里，使用的注释符号是 C 语言中的传统注释符号：

/*注释部分*/

注释部分的内容是本书在讲解例程时的一个重点内容，也是读者理解例程的重要依据。它不仅描述了大部分语句的功能，也描述了开发包的使用方法。例如，一些重要函数的使用

方法，对其进行了详细注解。

对每个例程的分析不仅包括注释部分，还有对程序的运行结果的分析，这也是讲解例程的一个重点内容。通过分析程序的运行结果，使读者更能够理解例程的执行过程。分析程序的运行结果，是对程序解释的一个手段，使读者更清楚开发包的使用过程。本书对例程结果的分析也比较详细，使读者更直观地了解例程的运行机理。

下面着重讨论 Libpcap 开发包的使用方法。

3.5.1 最简单的应用程序

使用 Libpcap 进行程序设计，其开发步骤并不复杂。下面举几个简单的例子分析 Libpcap 的使用方法。

1. 获取接口名字和掩码等信息

在 Libpcap 开发包中有对网络接口操作的函数。使用它们可以非常方便地对网络接口进行操作，并得到网络接口的名字、网络地址、掩码地址等信息。下面是一个获取接口名字和掩码等信息的例子。

```
/* 文件名字: getmask_code.c */
#include <pcap.h>                                   /*Libpcap 的头文件，必须包含*/
typedef u_int32_t  in_addr_t;
struct in_addr                                      /*地址类型*/
{
   in_addr_t s_addr;                                /*IP 地址类型*/
};
void  main()
{
   char error_content[PCAP_ERRBUF_SIZE];            /*出错信息*/
   struct in_addr net_ip_address;                   /*网络地址*/
   struct in_addr net_mask_address;                 /*掩码地址*/
   char *net_interface;                             /*接口名字*/
   char *net_ip_string;                             /*网络地址字符串形式*/
   char *net_mask_string;                           /*掩码地址字符串形式*/
   u_int32_t net_ip;                                /*网络地址*/
   u_int32_t net_mask;                              /*掩码地址*/
   net_interface = pcap_lookupdev(error_content);   /*获取网络接口*/
   pcap_lookupnet(net_interface,                    /*网络接口*/
                  &net_ip,                          /*网络地址*/
                  &net_mask,                        /*网络掩码*/
                  error_content);                   /*错误信息*/
   /*获取网络地址和掩码地址*/
   printf("Network Interface is :%s\n",net_interface);        /*输出接口名字*/
   net_ip_address.s_addr = net_ip;
   net_ip_string = inet_ntoa(net_ip_address);    /*把二进制网络地址格式转换为字符串形式*/
   printf("Network IP Address is :%s\n",net_ip_string);       /*输出网络地址*/
```

```
    net_mask_address.s_addr = net_mask;
    net_mask_string = inet_ntoa(net_mask_address);
        /*把二进制掩码地址格式转换为字符串形式*/
    printf("Network Mask Address is:%s\n",net_mask_string);  /*输出掩码地址*/
}
```

本例程的功能是获取机器的网络接口，并获取该接口的网络地址和掩码地址信息。

在 Linux 下编译此例程，编译命令如下：

```
gcc -o getmask getmask_code.c -lpcap
```

其中 gcc 是 Linux 下的编译工具；-o 表示要生成的二进制文件的名字；-lpcap 表示所需的库为 Libpcap 开发包的库，库的名字为 pcap。

编译成功后，运行此程序，其执行结果如下：

```
# ./getmask
Network Interface is :eth0
Network IP Address is :192.168.0.0
Network Mask Address is:255.255.255.0
```

运行结果如图 3-6 所示。可以看到，其获取的网络接口是 eth0（以太网接口名字），网络地址为 192.168.0.0，掩码地址为 255.255.255.0。

图 3-6　获取网络地址和掩码的运行结果

2. 捕获一个数据包

Libpcap 的核心功能是网络数据包的捕获。这里使用 Libpcap 捕获一个网络数据包，使用函数 pcap_next() 来实现。函数 pcap_next() 的功能是每次捕获一个数据包，然后返回此数据包的字节流。例程如下：

```
/*文件名字：get_a_packet_code.c*/
#include "pcap.h"
int main()
{
    char error_content[PCAP_ERRBUF_SIZE];          /*错误信息*/
    struct pcap_pkthdr protocol_header;            /*数据包头*/
    pcap_t *pcap_handle;                           /*Libpcap 句柄*/
    struct bpf_program bpf_filter;                 /*BPF 过滤规则*/
    char bpf_filter_string[] = "";                 /*过滤规则*/
    const u_char *packet_content;                  /*数据包内容*/
    bpf_u_int32 net_mask;                          /*网络掩码*/
    bpf_u_int32 net_ip;                            /*网络地址*/
    char *net_interface;                           /*网络接口*/
```

```
net_interface = pcap_lookupdev(error_content);          /*获取网络接口*/
pcap_lookupnet(net_interface, &net_ip, &net_mask, error_content);
    /*获取网络地址和掩码地址*/
pcap_handle = pcap_open_live(net_interface,             /*网络接口*/
                             BUFSIZ,                    /*数据包大小*/
                             1,                         /*混杂模式*/
                             0,                         /*等待时间*/
                             error_content);            /*错误信息*/
/*打开网络接口*/
pcap_compile(pcap_handle,                               /*Libpcap 句柄*/
             &bpf_filter,                               /*BPF 过滤规则*/
             bpf_filter_string,                         /*BPF 过滤规则字符串*/
             0,                                         /*优化参数*/
             net_ip);                                   /*网络地址*/
/*编译过滤规则*/
pcap_setfilter(pcap_handle,                             /*Libpcap 句柄*/
               &bpf_filter);                            /*BPF 过滤规则*/
/*设置过滤规则*/
packet_content = pcap_next(pcap_handle,                 /*Libpcap 句柄*/
                           &protocol_header);           /*数据包信息*/
/*捕获一个数据包，返回此数据包的内容*/
printf("Capture a packet from : %s\n",net_interface);       /*输出网络接口名字*/
printf("The packet length is :%d\n", protocol_header.len);  /*输出捕获的数据包的长度*/
pcap_close(pcap_handle);                                    /*关闭 Libpcap 操作*/
}
```

编译此文件，编译命令如下。

```
gcc -o getapacket get_a_packet_code.c -lpcap
```

程序运行结果如图 3-7 所示。

图 3-7 捕获一个数据包的运行结果

在这个程序中使用了设置过滤规则的功能。在设置过滤规则之前，必须对过滤规则的字符串形式进行编译。首先调用函数 pcap_compile() 对过滤规则进行编译，然后调用函数 pcap_setfilter() 设置过滤规则。

3. 捕获多个数据包

第 2 个例子只是捕获一个数据包，使用的是捕获函数 pcap_next()。要捕获多个网络数据包，可以使用捕获函数 pcap_loop() 来实现。函数 pcap_loop() 的功能是捕获多个数据包，数据包个数可以指定，也可以不指定。如果不指定，就无限循环地捕获网络数据包，直到出

现错误或强制退出为止。

函数 pcap_loop() 的使用方法与 pcap_next() 是不同的。在使用 pcap_loop() 时，应该使用回调函数来对捕获的网络数据包进行分析。在函数 pcap_loop() 中先注册回调函数，注册之后，每捕获一个数据包，就调用回调函数一次，所以可以在回调函数中对捕获到的每个数据包进行处理。回调函数是用户自己定义的，但必须满足 Libpcap 规定的函数类型。Libpcap 规定的回调函数的类型定义如下：

```
typedef void  (*pcap_handler)    (u_char *,
                                  const struct pcap_pkthdr *,
                                  const u_char *);
```

其中第一个参数表示从函数 pcap_loop()传递进来的参数；第二个参数是描述捕获到的数据包的信息；第三个参数是 u_char *类型，它表示捕获到的网络数据包的内容。把该数据包的内容取出，就可以对它进行分析了。第三个参数是最重要的参数。

可以看到，使用 Libpcap 捕获到的数据包内容是一个字节流数据，所以要分析它里面到底是什么内容，就必须进行严格的分析，读出每个字节的内容，然后判断它到底代表什么信息。

捕获多个数据包的例子如下：

```
/*文件名字：pcap_loop_code.c*/
    #include "pcap.h"                                          /*Libpcap 头文件*/
/*下面是回调函数的定义，在此处实现的功能是记录捕获到的网络数据包的个数*/
void packet_callback(u_char *argument,const struct pcap_pkthdr* packet_header,const
u_char*
packet_content)
{
    static int packet_number = 1;          /*静态局部变量，用来存放捕获到的数据包的个数*/
    printf("The %d packet is captured.\n",packet_number);     /*输出捕获数据包的个数*/
    packet_number++;                                           /*个数增加*/
}
void  main()
{
    pcap_t* pcap_handle;                                       /*Libpcap 句柄*/
    char error_content[PCAP_ERRBUF_SIZE];                      /*用来存储错误信息*/
    char *net_interface;                                       /*用来存储网络接口*/
    struct bpf_program bpf_filter;                             /*BPF 过滤规则*/
    char bpf_filter_string[] = "ip";                           /*过滤规则字符串形式*/
    bpf_u_int32 net_mask;                                      /*网络掩码*/
    bpf_u_int32 net_ip;                                        /*网络地址*/
    net_interface = pcap_lookupdev(error_content);             /*获取网络接口*/
    pcap_lookupnet(net_interface,                              /*网络接口*/
                   &net_ip,                                    /*网络地址*/
                   &net_mask,                                  /*网络掩码*/
                   error_content);                             /*错误信息*/
    /*获取网络地址和掩码*/
```

```
    pcap_handle = pcap_open_live(net_interface,                    /*网络接口*/
                                 BUFSIZ,                           /*数据包大小*/
                                 1,                                /*混杂模式*/
                                 0,                                /*等待时间*/
                                 error_content);                   /*错误信息*/
    /*打开网络接口*/
    pcap_compile(pcap_handle,                                      /*网络接口*/
                 &bpf_filter,                                      /*BPF 过滤规则*/
                 bpf_filter_string,                                /*过滤规则字符串*/
                 0,                                                /*优化参数*/
                 net_ip);                                          /*网络地址*/
    /*编译过滤规则*/
    pcap_setfilter(pcap_handle,                                    /*Libpcap 句柄*/
                   &bpf_filter);                                   /*BPF 过滤规则*/
    /*设置过滤规则*/
    pcap_loop(pcap_handle,                                         /*Libpcap 句柄*/
              10,                                                  /*捕获数据包的个数*
              packet_callback,                                     /*回调函数*/
              NULL);                                               /*传递给回调函数的参数*/
    /*注册回调函数 packet_callback ()，然后循环捕获网络数据包，每捕获一个数据包就调用回调函数进
    行处理，在这里捕获数据包的个数是 10，所以当捕获到 10 个网络数据包的时候就退出循环，如果个数设为-1，就
    表示无限循环。*/
    pcap_close(pcap_handle);                                       /*关闭 Libpcap 操作*/
}
```

在 Linux 下编译此程序，编译命令如下：

```
gcc -o pcaploop pcap_loop_code.c -lpcap
```

运行此程序，其运行结果如图 3-8 所示。可以看到，此程序的功能是捕获 10 个数据包，然后每捕获一个数据包，就输出一条信息。

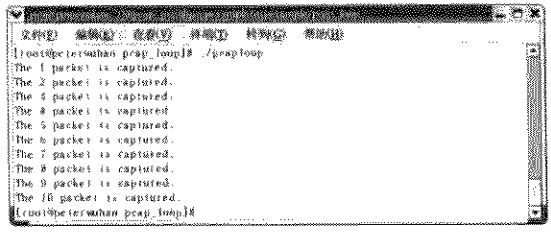

图 3-8　循环捕获多个数据包的运行结果

本程序演示了函数 pcap_loop() 的使用方法。从程序的运行过程可以看出，函数 pcap_loop() 的功能是注册一个回调函数。此回调函数是用户自己定义的，它必须满足 Libpcap 定义的函数类型，这样就可以利用此回调函数对捕获到的数据包进行处理。在本例子中，为了测试，回调函数的功能设置得非常简单，只是记录了数据包的个数。在后面的例

子中，会对此回调函数进行更复杂的实现，使它实现更多的功能。

3.5.2 以太网数据包的捕获

以太网协议是链路层协议，它的协议格式在此不介绍，读者可以查看其他资料。本节使用 Libpcap 开发包实现对网络数据包的以太网协议进行分析。

根据 TCP/IP 分层的概念，TCP/IP 协议族共有 4 个层次，分别为链路层、网络层、传输层和应用层。在后面的例子中主要对 TCP/IP 协议族中用得最多的几个协议进行讨论，它们是以太网协议、ARP 协议、IP 协议、TCP 协议、UDP 协议和 ICMP 协议。其中，以太网协议属于链路层。一般情况下，认为 ARP 协议也属于链路层，但有些人把 ARP 放在网络层进行讨论，因为根据协议的分析，ARP 是基于以太网协议的，但这里把它放在网络层讨论。同样，一般认为 ICMP 协议属于网络层，但是有些人认为它是基于 IP 协议传输的，故应属于上一层协议，在此把它放在传输层讨论。读者稍微注意一下就可以了。

下面是使用函数 pcap_next() 来实现以太网协议分析的例程，该例程只捕获一个以太网协议数据包。

```
/*文件名字: get_ethernet1_code.c*/
#include <pcap.h>
/*Libpcap 头文件*/
struct ether_header                                  /*以太网协议的数据结构*/
{
    u_int8_t  ether_dhost[6];                        /*目的以太网地址*/
    u_int8_t  ether_shost[6];                        /*源以太网地址*/
    u_int16_t ether_type;                            /*以太网类型*/
};
void main( )
{
    char error_content[PCAP_ERRBUF_SIZE];            /*错误信息*/
    pcap_t* pcap_handle;                             /*Libpcap 句柄*/
    const u_char *packet_content;                    /*数据包缓存*/
    u_char *mac_string;                              /*以太网地址*/
    u_short ethernet_type;                           /*以太网类型*/
    bpf_u_int32 net_mask;                            /*网络掩码*/
    bpf_u_int32 net_ip;                              /*网络地址*/
    char *net_interface;                             /*网络接口*/
    struct pcap_pkthdr protocol_header;              /*数据包信息*/
    struct ether_header *ethernet_protocol;          /*以太网协议变量*/
    struct bpf_program bpf_filter;                   /*BPF 过滤规则*/
    char bpf_filter_string[] = "ip";
    /*过滤规则，此处表示只捕获 IP 数据包，当然它也是以太网数据包，因为本程序的环境为以太网环境。IP
数据包是基于以太网协议的，所以捕获 IP 数据包，就相当于捕获以太网数据包。此处只是为了能够捕获到以太网
数据包，别无他意。当然，还可以使用其他过滤规则，只要能捕获到以太网协议就行，因为本程序的目的就是为了
捕获到以太网数据包*/
    net_interface = pcap_lookupdev(error_content);       /*获取网络接口*/
```

```
pcap_lookupnet(net_interface,                          /*网络接口*/
               &net_ip,                                /*网络地址*/
               &net_mask,                              /*网络掩码*/
               error_content);                         /*错误信息*/
      /*获取网络地址和掩码地址*/
pcap_handle = pcap_open_live(net_interface,            /*网络接口*/
                             BUFSIZ,                   /*数据包大小*/
                             1,                        /*混杂模式*/
                             0,                        /*等待时间*/
                             error_content);           /*错误信息*/
      /*打开网络接口*/
pcap_compile(pcap_handle,                              /*Libpcap 句柄*/
             &bpf_filter,                              /*BPF 过滤规则*/
             bpf_filter_string,                        /*过滤规则字符串*/
             0,                                        /*优化参数*/
             net_ip);                                  /*网络地址*/
      /*编译 BPF 过滤规则*/
pcap_setfilter(pcap_handle,                            /*Libpcap 句柄*/
             &bpf_filter                               /*BPF 过滤规则*/
             );
      /*设置过滤规则*/
if(pcap_datalink(pcap_handle)!=DLT_EN10MB)
          return;
packet_content = pcap_next(pcap_handle,                /*Libpcap 句柄*/
&protocol_header                                       /*数据包信息*/
);
/*捕获一个网络数据包，返回此数据包的缓存，此缓存存储的就是捕获到的网络数据包的字节流，接下来就是要分析此缓存的内容*/
printf("-------------------****-----------------------\n");
printf("Capture a Packet from net_interface :\n");
printf("%s \n",net_interface);                         /*输出网络接口*/
printf("Capture Time is:\n");
printf("%s",ctime((const time_t*)&protocol_header.ts.tv_sec));  /*输出此数据包的时间信息*/
printf("Packet Length is:\n");
printf("%d\n",protocol_header.len);                    /*输出此数据包的长度信息*/
ethernet_protocol = (struct ether_header *) packet_content;
/*把此数据包缓存进行类型的强制转换，使它变成以太网协议格式的数据类型，然后就可以对它的各个字段进行访问了。注意此处的强制转换功能，是对网络数据包进行分析的重点操作*/
printf("Ethernet type is :\n");
ethernet_type= ntohs(ethernet_protocol->ether_type);
      /*获得以太网的类型，它表示上层协议的类型，即网络层的协议类型*/
printf("%04x\n",ethernet_type);                        /*输出以太网类型*/
switch(ethernet_type)
```

```
    {
    case 0x0800:  printf("The network layer is IP protocol\n");break;
    /*如果以太网类型是 0x0800 就表示上层协议类型为 IP 协议*/
        case 0x0806:  printf("The network layer is ARP protocol\n"); break;
            /*如果以太网类型是 0x0806 就表示上层协议类型为 ARP 协议*/
        case 0x8035:  printf("The network layer is RARP protocol\n");break;
            /*如果以太网类型是 0x8035 就表示上层协议类型为 RARP 协议*/
        default :break;
            /*其他以太网类型，在这里没有分析*/
    }
    printf("Mac Source Address is : \n");
    mac_string=ethernet_protocol->ether_shost;          /*获得源以太网地址*/
    printf("%02x:%02x:%02x:%02x:%02x:%02x\n",*mac_string,*(mac_string+1),
                *(mac_string+2),*(mac_string+3),
                *(mac_string+4),*(mac_string+5));
/*要对以太网地址进行转换，使它变成字符串形式进行显示，如 11:11:11:11:11:11。因为读到的源以太网地址
是字节流顺序的*/
printf("Mac Destination Address is : \n");
mac_string=ethernet_protocol->ether_dhost;              /*获得目的以太网地址*/
    printf("%02x:%02x:%02x:%02x:%02x:%02x\n",*mac_string,*(mac_string+1),
                *(mac_string+2),*(mac_string+3),
                *(mac_string+4),*(mac_string+5));       /*同样，对其地址进行转换*/
    printf("--------------------*****-----------------------\n");
    pcap_close(pcap_handle);                            /*关闭 Libpcap 操作*/
}
```

本程序的功能是捕获一个以太网协议数据包，然后分析出各个字段的内容，包括目的以太网地址、源以太网地址和以太网类型。对以太网类型进行了进一步判断，分析出上层协议的类型。根据以太网类型，可以知道数据包上层协议的类型，然后可以对数据包进行更进一步的分析，这些分析在后面将要涉及到。

编译此程序，编译命令如下：

```
Gcc -o getethernet get_ethernet1_code.c -lpcap
```

运行此程序，其运行结果如下：

```
--------------------*****-----------------------
Capture a Packet from net_interface :
eth0
Capture Time is:
Tue Apr 12 18:21:48 2005
Packet Length is:
78
Ethernet type is :
0800
The network layer is IP protocol
```

```
Mac Source Address is :
00:09:7b:52:e4:0a
Mac Destination Address is :
00:e0:4c:80:03:4b
-----------------------*****-----------------------
```

程序运行结果如图 3-9 所示。

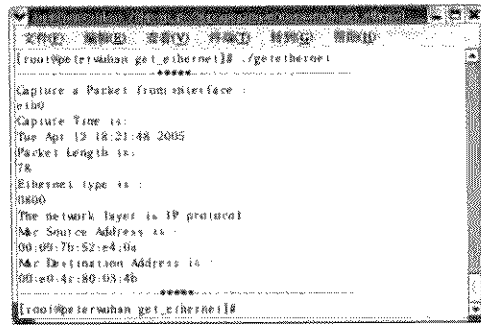

图 3-9　捕获一个以太网数据包的运行结果

从运行的结果分析，捕获的以太网数据包，其网络接口是 eth0，长度是 78，以太网类型是 0800，表示上层协议类型为 IP 协议。源以太网地址是 00:09:7b:52:e4:0a，目的以太网地址是 00:e0:4c:80:03:4b，每个字段的内容都分析出来了。

上面的例子只捕获了一个以太网数据包，要捕获多个以太网数据包，可以使用函数 pcap_loop() 来实现，而且要使用回调函数。

下面是捕获多个以太网数据包的代码。

```
/*文件名字：get_ethernet2_code.c*/
#include "pcap.h"                    /*Libpcap 头文件*/
struct ether_header                  /*以太网协议的数据结构*/
{
    u_int8_t  ether_dhost[6];        /*目的以太网地址*/
    u_int8_t  ether_shost[6];        /*源以太网地址*/
    u_int16_t ether_type;            /*以太网类型*/
};
/*下面的函数是回调函数，其功能是实现捕获以太网数据包，分析其各个字段的内容。注意，其中参数
packet_content 表示的就是捕获到的网络数据包内容；参数 argument 是从函数 pcap_loop() 传递过来的；
参数 pcap_pkthdr 表示捕获到的数据包基本信息，包括时间，长度等信息。*/
void ethernet_protocol_packet_callback(u_char *argument,const struct pcap_pkthdr*
packet_header,const u_char* packet_content)
{
    u_short ethernet_type;                          /*以太网类型*/
    struct ether_header *ethernet_protocol;         /*以太网协议格式*/
```

```
    net_interface = pcap_lookupdev(error_content);          /*获得网络接口*/
    pcap_lookupnet(net_interface,                           /*网络接口*/
            &net_ip,                                        /*网络地址*/
            &net_mask,                                      /*网络掩码*/
            error_content);                                 /*错误信息*/
    /*获得网络地址和网络掩码*/
    pcap_handle = pcap_open_live(net_interface,         /*网络接口*/
                        BUFSIZ,                         /*数据包大小*/
                        1,                              /*混杂模式*/
                        0,                              /*等待时间*/
                        error_content);                 /*错误信息*/
        /*打开网络接口*/
    pcap_compile(pcap_handle,                           /*Libpcap 句柄*/
            &bpf_filter,                                /*BPF 过滤规则*/
            bpf_filter_string,                          /*过滤规则字符串*/
            0,                                          /*优化参数*/
            net_ip);                                    /*网络地址*/
        /*编译过滤规则*/
    pcap_setfilter(pcap_handle,                         /*Libpcap 句柄*/
            &bpf_filter);                               /*BPF 过滤规则*/
        /*设置规律规则*/
    if(pcap_datalink(pcap_handle)!=DLT_EN10MB)
        return;
    pcap_loop(pcap_handle,                      /*Libpcap 句柄*/
        -1,                                     /*捕获数据包的个数，此处为-1 表示无限捕获*/
        ethernet_protocol_packet_callback,      /*回调函数*/
        NULL);                                  /*传递给回调函数的参数*/
/*无限循环捕获网络数据包，注册回到函数 ethernet_protocol_callback()，捕获每个数据包都要调用此回
调函数进行操作*/
    pcap_close(pcap_handle);                    /*关闭 Libpcap 操作*/
}
```

此程序的功能与前一个例子相似，都是对捕获的以太网数据包进行分析，只不过这个程序可以捕获任意个数据包，使用的是回调函数的形式。可以看到，使用回调函数可以捕获多个数据包，并且使程序的结构非常简洁，功能分布很合理。从这点来看，使用回调函数是 Libpcap 的一个优点。

在回调函数中可以对网络数据包进行各种处理。在这个程序中，分析了以太网协议，把它的各个字段的内容都进行了分析。后面所有的例子，都使用回调函数的形式来进行数据包的捕获和分析。

编译此例程，编译命令如下：

```
gcc -o getethernte2 get_ethernet2_code.c -lpcap
```

运行此程序，其输出结果如下（只给出 3 个包的捕获结果）：

```
# ./getethernet2
**************************************************
The 1 Ethernet  packet is captured.
-----------   Ethernet Protocol (Link Layer)  ------------
The 1 Ethernet packet is captured.
Ethernet type is :
0800
The network layer is IP protocol
Mac Source Address is :
00:09:7b:52:e4:0a
Mac Destination Address is :
00:e0:4c:80:03:4b
**************************************************
**************************************************
The 2 Ethernet  packet is captured.
-----------   Ethernet Protocol (Link Layer)  ------------
The 2 Ethernet packet is captured.
Ethernet type is :
0800
The network layer is IP protocol
Mac Source Address is :
00:09:7b:52:e4:0a
Mac Destination Address is :
00:e0:4c:80:03:4b
**************************************************
**************************************************
The 3 Ethernet  packet is captured.
-----------   Ethernet Protocol (Link Layer)  ------------
The 3 Ethernet packet is captured.
Ethernet type is :
0800
The network layer is IP protocol
Mac Source Address is :
00:09:7b:52:e4:0a
Mac Destination Address is :
00:e0:4c:80:03:4b
**************************************************
```

其运行结果如图 3-10 所示。

从程序的运行结果，可以看到，它可以捕获多个数据包，然后显示数据包的以太网协议内容。

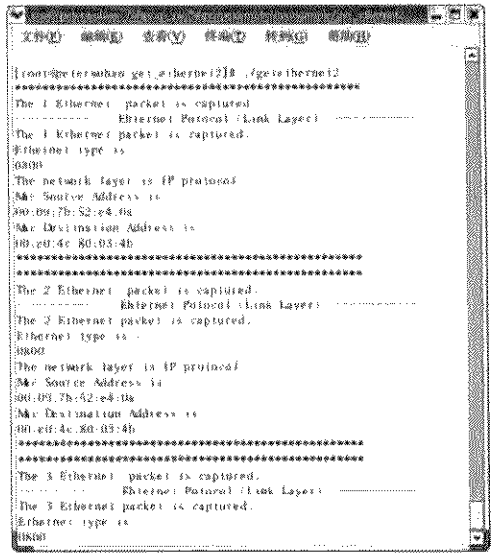

图 3-10 捕获和分析以太网数据包的运行结果

3.5.3 ARP 数据包的捕获

本节举例说明怎样捕获 ARP 数据包，并对 ARP 数据包进行分析。同样使用 Libpcap 的函数 pcap_loop() 来实现。

由于 ARP 是基于以太网协议的，所以必须先分析出以太网协议，然后根据以太网协议类型的值来判断上层协议是否是 ARP 协议。如果是，就分析 ARP 协议；如果不是，就跳过。

在以太网协议中，用一个字段来判断上层协议是什么协议，即以太网协议类型字段 ether_type。如果其值等于 0x0806，就表示上层协议是 ARP 协议。

这里先用一个函数来实现分析 ARP 协议的功能，然后在回调函数中直接调用此函数来实现 ARP 协议分析。回调函数的功能仍然是分析以太网协议。实现 ARP 协议分析的函数定义为 arp_protocol_packet_callback()。

整个程序的源代码如下：

```
/*文件名字：get_arp_code.c*/
#include "pcap.h"
/*下面是以太网协议格式*/
struct ether_header
{
    u_int8_t  ether_dhost[6];                    /*目的以太网地址*/
    u_int8_t  ether_shost[6];                    /*源以太网地址*/
    u_int16_t ether_type;                        /*以太网类型*/
}
```

```
/*下面是IP地址格式*/
typedef u_int32_t  in_addr_t;
struct in_addr
{
    in_addr_t s_addr;                          /*存放IP地址*/
};
/*下面是ARP协议格式*/
struct arp_header
{
    u_int16_t  arp_hardware_type;              /*硬件地址类型*/
    u_int16_t  arp_protocol_type;              /*协议地址类型*/
    u_int8_t   arp_hardware_length;            /*硬件地址长度*/
    u_int8_t   arp_protocol_length;            /*协议地址长度*/
    u_int16_t  arp_operation_code;             /*操作类型*/
    u_int8_t arp_source_ethernet_address[6];   /*源以太网地址*/
    u_int8_t arp_source_ip_address [4];        /*源IP地址*/
    u_int8_t  arp_destination_ethernet_address [6];     /*目的以太网地址*/
    u_int8_t arp_destination_ip_address[4];    /*目的IP地址*/
};
/*下面是实现ARP协议分析的函数定义*/
void arp_protocol_packet_callback(u_char *argument,const struct pcap_pkthdr*
packet_header,const u_char* packet_content)
{
    struct arp_header *arp_protocol;           /*ARP协议变量*/
    u_short protocol_type;                     /*协议类型*/
    u_short hardware_type;                     /*硬件类型*/
    u_short operation_code;                    /*操作类型*/
    u_char *mac_string;                        /*以太网地址*/
    struct in_addr source_ip_address;          /*源IP地址*/
    struct in_addr destination_ip_address;     /*目的IP地址*/
    u_char hardware_length;                    /*硬件地址长度*/
    u_char protocol_length;                    /*协议地址长度*/
    printf("--------   ARP Protocol (Network Layer)    --------\n");
    arp_protocol = (struct arp_header *) (packet_content+14);
    /*获得ARP协议数据。注意在这里要跳过以太网数据部分，它的长度刚好是14，所以在这里加上14，使指
针跳过14字节*/
    hardware_type = ntohs(arp_protocol->arp_hardware_type);        /*获得硬件类型*/
    protocol_type = ntohs(arp_protocol->arp_protocol_type);        /*获得协议类型*/
    operation_code = ntohs(arp_protocol->arp_operation_code);      /*获得操作码*/
    hardware_length=arp_protocol->arp_hardware_length;             /*获得硬件地址长度*/
    protocol_length=arp_protocol->arp_protocol_length;             /*获得协议地址长度*/
    printf("ARP Hardware Type:%d\n", hardware_type);
    printf("ARP Protocol Type:%d\n", protocol_type);
```

```
    printf("ARP Hardware Length:%d\n",hardware_length);
    printf("ARP Protocol Length:%d\n",protocol_length);
    printf("ARP Operation:%d\n", operation_code);
    switch (operation_code)                    /*根据操作码进行判断是 ARP 什么类型协议*/
    {
        case 1: printf("ARP Request Protocol\n");break;          /*ARP 查询协议*/
        case 2: printf("ARP Reply Protocol\n");break;            /*ARP 应答协议*/
        case 3: printf("RARP Request Protocol\n");break;         /*RARP 查询协议*/
        case 4: printf("RARP Reply Protocol\n");break;           /*RARP 应答协议*/
        default:break;
    }
    printf("Ethernet Source Address is : \n");
    mac_string=arp_protocol->arp_source_ethernet_address;
    printf("%02x:%02x:%02x:%02x:%02x:%02x\n",*mac_string,*(mac_string+1),
                *(mac_string+2),*(mac_string+3),
                *(mac_string+4),*(mac_string+5));                /*获得源以太网地址*/
    memcpy((void *) &source_ip_address, (void *) &arp_protocol->
            arp_source_ip_address, sizeof (struct in_addr));
    printf("Source IP Address:%s\n", inet_ntoa(source_ip_address)); /*获得源 IP 地址*/
    printf("Ethernet Destination Address is : \n");
    mac_string=arp_protocol-> arp_destination_ethernet_address
    printf("%02x:%02x:%02x:%02x:%02x:%02x\n",*mac_string,*(mac_string+1),
                *(mac_string+2),*(mac_string+3),
                *(mac_string+4),*(mac_string+5));        /*获得目的以太网地址*/
    memcpy((void *) &destination_ip_address, (void *)
    &arp_protocol->arp_destination_ip_address, sizeof (struct in_addr));
    printf("Destination IP Address:%s\n", inet_ntoa(destination_ip_address));
                                                         /*获得目的 IP 地址*/
}
/*下面是回调函数，实现以太网协议分析*/
void ethernet_protocol_packet_callback(u_char *argument,const struct pcap_pkthdr*
packet_header,const u_char* packet_content)
{
    u_short ethernet_type;                          /*以太网类型*/
    struct ether_header *ethernet_protocol;         /*以太网协议变量*/
    u_char *mac_string;                             /*以太网地址*/
    static int packet_number = 1;
    printf("**************************************************\n");
    printf("The %d  ARP  packet is captured.\n",packet_number);
    printf("--------   Ethernet Protocol (Link Layer)    --------\n");
    ethernet_protocol = (struct ether_header *) packet_content;
        /*获得以太网协议数据*/
    printf("Ethernet type is :\n");
```

```
    ethernet_type= ntohs(ethernet_protocol->ether_type);          /*获得以太网类型*/
    printf("%04x\n",ethernet_type);
    switch(ethernet_type)                                         /*根据以太网类型判断上层协议类型*/
    {
        case 0x0800:  printf("The network layer is IP protocol\n");break;
        case 0x0806:  printf("The network layer is ARP protocol\n"); break;
        case 0x8035:  printf("The network layer is RARP protocol\n");break;
        default :break;
    }
    printf("Mac Source Address is : \n");
    mac_string=ethernet_protocol->ether_shost;
    printf("%02x:%02x:%02x:%02x:%02x:%02x\n",*mac_string,*(mac_string+1),
                *(mac_string+2),*(mac_string+3),
                *(mac_string+4),*(mac_string+5));
    /*获得源以太网地址*/
    printf("Mac Destination Address is : \n");
    mac_string=ethernet_protocol->ether_dhost;
    printf("%02x:%02x:%02x:%02x:%02x:%02x\n",*mac_string,*(mac_string+1),
                *(mac_string+2),*(mac_string+3),
                *(mac_string+4),*(mac_string+5));
    /*获得目的以太网地址*/
    switch(ethernet_type)
    {
        case 0x0806:  arp_protocol_packet_callback(argument,
                                                   packet_header,
                                                   packet_content);
                  break;
    /*如果以太网类型是 0x0806，表示上层协议是 ARP 协议，应该调用分析 ARP 协议的函数。注意此时的参数
    传递，全部是回调函数的参数，它代表的是同一个网络数据包，所以在这里，既分析此数据包的以太网协议
    部分，又分析此数据包的 ARP 协议部分*/
    default :break;
    }
    printf("*************************************************\n");
    packet_number++;
}
void  main()
{
    pcap_t* pcap_handle;                                          /*Libpcap 句柄*/
    char error_content[PCAP_ERRBUF_SIZE];                         /*错误信息*/
    char *net_interface;                                          /*网络接口*/
    struct bpf_program bpf_filter;                                /*BPF 过滤规则*/
    char bpf_filter_string[] = "arp";
    /*过滤规则字符串，这里表示本程序只是捕获所有 ARP 协议的网络数据包*/
```

```
    bpf_u_int32 net_mask;                                   /*网络掩码*/
    bpf_u_int32 net_ip;                                     /*网络地址*/
    net_interface = pcap_lookupdev(error_content);          /*获得网络接口*/
    pcap_lookupnet(net_interface,
                        &net_ip,
                        &net_mask,
                        error_content);
    /*获得网络地址和网络掩码*/
    pcap_handle = pcap_open_live(net_interface,BUFSIZ,1,0, error_content);
    /*打开网络接口*/
    pcap_compile(pcap_handle, &bpf_filter, bpf_filter_string, 0, net_ip);
    /*编译过滤规则*/
    pcap_setfilter(pcap_handle, &bpf_filter);
    /*设置过滤规则*/
    if(pcap_datalink(pcap_handle)!=DLT_EN10MB)
        return;
    pcap_loop(pcap_handle,-1,ethernet_protocol_packet_callback,NULL);
    /*注册回调函数，循环捕获数据包*/
    pcap_close(pcap_handle);                /*关闭 Libpcap 操作*/
}
```

此程序的功能是捕获 ARP 数据包，然后分析此数据包的内容，包括它的以太网部分和 ARP 协议部分的各个字段的内容。其执行过程，首先利用 pcap_loop()注册回调函数 ethernet_protocol_packet_callback() 分析以太网协议，然后根据分析结果，调用函数 arp_protocol_packet_callback() 继续分析 ARP 协议。注意，分析的对象是同一个网络数据包。

编译此程序，编译命令如下：

```
gcc -o getarp get_arp_code.c -lpcap
```

执行此程序，其输出的内容如下，只给出两个 ARP 数据包的内容：

```
***********************************************
The 1  ARP  packet is captured.
--------  Ethernet Protocol (Link Layer)   --------
Ethernet type is :
0806
The network layer is ARP protocol
Mac Source Address is :
00:0a:e6:5d:2b:3d
Mac Destination Address is :
ff:ff:ff:ff:ff:ff
--------  ARP Protocol (Network Layer)   --------
ARP Hardware Type:1
ARP Protocol Type:2048
ARP Hardware Length:6
ARP Protocol Length:4
```

```
ARP Operation:1
ARP Request Protocol
Ethernet Source Address is :
00:0a:e6:5d:2b:3d
Source IP Address:192.168.0.5
Ethernet Destination Address is :
00:00:00:00:00:00
Destination IP Address:192.168.0.6
**********************************************
**********************************************
The 2  ARP  packet is captured.
--------   Ethernet Protocol (Link Layer)   --------
Ethernet type is :
0806
The network layer is ARP protocol
Mac Source Address is :
00:0a:e6:5d:2b:3d
Mac Destination Address is :
ff:ff:ff:ff:ff:ff
--------   ARP Protocol (Network Layer)   --------
ARP Hardware Type:1
ARP Protocol Type:2048
ARP Hardware Length:6
ARP Protocol Length:4
ARP Operation:1
ARP Request Protocol
Ethernet Source Address is :
00:0a:e6:5d:2b:3d
Source IP Address:192.168.0.5
Ethernet Destination Address is :
00:00:00:00:00:00
Destination IP Address:192.168.0.6
**********************************************
```

其执行结果如图 3-11 所示。

从运行结果分析，此程序的功能是不断捕获 ARP 数据包，然后解析其数据包的内容，把其链路层部分和网络层部分都解析出来。链路层协议是以太网协议，包括以太网协议的协议类型、源以太网地址和目的以太网地址。上层协议是 ARP 协议，包括硬件类型、协议类型、硬件地址长度、协议地址长度、操作码、源以太网地址、源 IP 地址、目的以太网地址和目的 IP 地址。

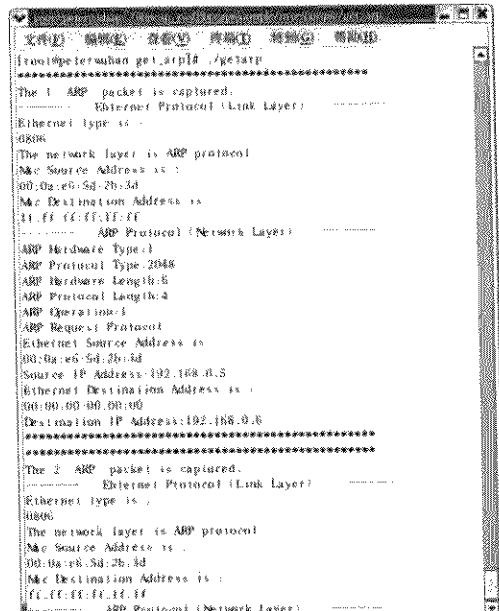

图 3-11 捕获和分析 ARP 数据包

3.5.4 IP 数据包的捕获

在分析以太网协议的时候，如果以太网协议类型是 0x0800，就表示上层协议类型是 IP 协议。所以要捕获和分析 IP 数据包，就必须先分析以太网协议，然后再分析 IP 协议。

同样，还是使用函数 pcap_loop()来进行数据包捕获，回调函数的功能是分析以太网数据包，然后根据分析的结果，再调用分析 IP 协议的函数。实现分析 IP 协议的函数定义为 ip_protocol_packet_callback()。

捕获和分析 IP 协议数据包的源代码如下：

```
/*文件名字: get_ip_code.c */
#include "pcap.h"                    /*Libpcap 头文件*/
/*下面是以太网协议格式*/
struct ether_header
{
    u_int8_t  ether_dhost[6];        /*目的以太网地址*/
    u_int8_t  ether_shost[6];        /*源以太网地址*/
    u_int16_t ether_type;            /*以太网类型*/
};
/*下面是 IP 地址格式*/
typedef u_int32_t  in_addr_t;
struct in_addr
```

```
{
    in_addr_t s_addr;
};
/*下面是IP协议格式*/
struct ip_header
{
    #ifdef  WORDS_BIGENDIAN
    u_int8_t      ip_version:4,                /*IP协议版本*/
                  ip_header_length:4;          /*IP协议首部长度*/
    #else
    u_int8_t      ip_header_length:4,
                  ip_version:4;
    #endif
    u_int8_t   ip_tos;                         /*TOS服务质量*/
    u_int16_t  ip_length;                      /*总长度*/
    u_int16_t  ip_id;                          /*标识*/
    u_int16_t  ip_off;                         /*偏移*/
    u_int8_t   ip_ttl;                         /*生存时间*/
    u_int8_t   ip_protocol;                    /*协议类型*/
    u_int16_t  ip_checksum;                    /*校验和*/
    struct in_addr  ip_souce_address;          /*源IP地址*/
    struct in_addr  ip_destination_address;    /*目的IP地址*/
};
/*下面是实现IP数据包分析的函数定义*/
void ip_protocol_packet_callback(u_char *argument,const struct pcap_pkthdr*
packet_header,const u_char* packet_content)
{
    struct ip_header *ip_protocol;             /*IP协议变量*/
    u_int header_length;                       /*长度*/
    u_int offset;                              /*偏移*/
    u_char tos;                                /*服务质量*/
    u_int16_t checksum;                        /*校验和*/
    ip_protocol = (struct ip_header *) (packet_content+14);
        /*获得IP协议数据内容，去掉以太网头*/
    checksum = ntohs(ip_protocol->ip_checksum);                /*获得校验和*/
    header_length = ip_protocol->ip_header_length*4;           /*获得长度*/
    tos = ip_protocol->ip_tos;                                 /*获得TOS*/
    offset = ntohs(ip_protocol->ip_off);                       /*获得偏移量*/
    printf("----------- IP Protocol  (Network Layer)  -----------\n");
    printf("IP Version:%d\n", ip_protocol->ip_version);
        printf("Header length:%d\n", header_length);
        printf("TOS:%d\n", tos);
        printf("Total length:%d\n", ntohs(ip_protocol->ip_length));  /*获得总长度*/
```

```
        printf("Identification:%d\n", ntohs(ip_protocol->ip_id));     /*获得标识*/
        printf("Offset:%d\n", (offset & 0x1fff) * 8);
        printf("TTL:%d\n", ip_protocol->ip_ttl);                      /*获得 TTL*/
        printf("Protocol:%d\n", ip protocol->ip_protocol);            /*获得协议类型*/
    switch (ip_protocol->ip_protocol)                                 /*判断协议类型的值*/
        {
            case 6:  printf("The Transport Layer Protocol is TCP\n");break;
                /*如果协议类型为 6，表示上层协议为 TCP 协议*/
            case 17: printf("The Transport Layer Protocol is UDP\n");break;
                /*如果协议类型为 17，表示上层协议为 UDP 协议*/
            case 1:  printf("The Transport Layer Protocol is ICMP\n");break;
                /*如果协议类型为 1，表示上层协议为 ICMP 协议*/
            default:break;
    }
    printf("Header checksum:%d\n", checksum);
    printf("Source address:%s\n", inet_ntoa(ip_protocol->ip_souce_address));
        /*获得源 IP 地址*/
    printf("Destination address:%s\n", inet_ntoa(ip_protocol->ip_destination_address));
        /*获得目的 IP 地址*/
}
/*下面是回调函数，实现捕获和分析以太网协议*/
void ethernet_protocol_packet_callback(u_char *argument,const struct pcap_pkthdr*
packet_header,const u_char* packet_content)
{
    u_short ethernet_type;                              /*以太网类型*/
    struct ether_header *ethernet_protocol;             /*以太网协议变量*/
    u_char *mac_string;
    static int packet_number = 1;
        printf("*************************************************\n");
    printf("The %d IP  packet is captured.\n",packet_number);
    printf("-------------  Ethernet Protocol (Link Layer)     ------------\n");
    ethernet_protocol = (struct ether_header *) packet_content;
        /*获得以太网协议数据内容*/
    printf("Ethernet type is :\n");
    ethernet_type= ntohs(ethernet_protocol->ether_type);
        /*获得以太网类型*/
    printf("%04x\n",ethernet_type);
    switch(ethernet_type)   /*判断以太网类型的值*/
    {
        case 0x0800:  printf("The network layer is IP protocol\n");break;
            /*如果为 0x0800，表示上层协议为 IP 协议*/
        case 0x0806:  printf("The network layer is ARP protocol\n"); break;
            /*如果为 0x0806，表示上层协议为 ARP 协议*/
```

```
        case 0x8035:  printf("The network layer is RARP protocol\n");break;
            /*如果为 0x8035，表示上层协议为 RARP 协议*/
        default :break;
    }
    printf("Mac Source Address is : \n");
    mac_string=ethernet_protocol->ether_shost;
    printf("%02x:%02x:%02x:%02x:%02x:%02x\n",*mac_string,*(mac_string+1),
                *(mac_string+2),*(mac_string+3),
                *(mac_string+4),*(mac_string+5));
    /*获得源以太网地址*/
    printf("Mac Destination Address is : \n");
    mac_string=ethernet_protocol->ether_dhost;
    printf("%02x:%02x:%02x:%02x:%02x:%02x\n",*mac_string,*(mac_string+1),
                *(mac_string+2),*(mac_string+3),
                *(mac_string+4),*(mac_string+5));
        /*获得目的以太网地址*/
    switch(ethernet_type)
    {
        case 0x0800:
        ip_protocol_packet_callback (argument,
                                     packet_header,
                                     packet_content);
        break;
        /*如果上层协议是 IP 协议，就调用分析 IP 协议的函数对 IP 协议进行分析。注意，此时传递的参数，与回
        调函数的参数是一样的，表示代表同一个数据包*/
        default :break;
    }
    printf("**************************************************\n");
    packet_number++;
}
void  main()
{
    pcap_t* pcap_handle;                                /*Libpcap 句柄*/
    char error_content[PCAP_ERRBUF_SIZE];               /*错误信息*/
    char *net_interface;                                /*网络接口*/
    struct bpf_program bpf_filter;                      /*BPF 过滤规则*/
    char bpf_filter_string[] = "ip";
    /*过滤规则字符串，此时表示本程序只是捕获 IP 协议的网络数据包*/
    bpf_u_int32 net_mask;                               /*网络掩码*/
    bpf_u_int32 net_ip;                                 /*网络地址*/
    net_interface = pcap_lookupdev(error_content);      /*获得网络接口*/
    pcap_lookupnet(net_interface,
                   &net_ip,
```

```
                &net_mask,
                error_content);                          /*获得网络地址和掩码*/
    pcap_handle = pcap_open_live(net_interface,
                    BUFSIZ,
                    1,
                    0,
                    error_content);                      /*打开网络接口*/
    pcap_compile(pcap_handle,
                &bpf_filter,
                bpf_filter_string,
                0,
                net_ip);                                 /*编译 BPF 过滤规则*/
    pcap_setfilter(pcap_handle,
                  &bpf_filter);                          /*设置过滤规则*/
    if(pcap_datalink(pcap_handle)!=DLT_EN10MB)
            return;
    pcap_loop(pcap_handle,
                -1,
                ethernet_protocol_packet_callback,
                NULL);
        /*注册回调函数，循环捕获数据包*/
        pcap_close(pcap_handle);                         /*关闭 Libpcap 操作*/
}
```

此程序的功能是捕获和分析 IP 数据包，由于 IP 数据包是基于以太网协议的，所以应该在分析以太网协议的基础上再分析 IP 协议。

在回调函数中分析以太网协议，并根据以太网协议的类型进行进一步的分析，如果其值为 0x0800，就表示网络层协议是 IP 层协议。然后调用分析 IP 协议的函数，实现对 IP 协议的分析。

编译此程序，其编译命令如下：

```
gcc -o getip get_ip_code.c -lpcap
```

运行此程序，其输出内容如下（只给出两个数据包的内容）：

```
************************************************
The 1 IP  packet is captured.
-------------  Ethernet Protocol (Link Layer)     ------------
Ethernet type is :
0800
The network layer is IP protocol
Mac Source Address is :
00:0a:e6:5d:2b:3d
Mac Destination Address is :
00:00:e8:e7:b9:46
----------- IP Protocol  (Network Layer)  -----------
```

```
IP Version:4
Header length:20
TOS:0
Total length:84
Identification:0
Offset:0
TTL:64
Protocol:1
The Transport Layer Protocol is ICMP
Header checksum:47439
Source address:192.168.0.5
Destination address:192.168.0.4
**************************************************
**************************************************
The 2 IP packet is captured.
------------  Ethernet Protocol (Link Layer)     ------------
Ethernet type is :
0800
The network layer is IP protocol
Mac Source Address is :
00:00:e8:e7:b9:46
Mac Destination Address is :
00:0a:e6:5d:2b:3d
----------- IP Protocol (Network Layer) -----------
IP Version:4
Header length:20
TOS:0
Total length:84
Identification:5227
Offset:0
TTL:128
Protocol:1
The Transport Layer Protocol is ICMP
Header checksum:25828
Source address:192.168.0.4
Destination address:192.168.0.5
**************************************************
```

程序的执行结果如图 3-12 所示。

从运行结果分析，捕获 IP 数据包是在分析以太网数据包的基础上进行的。在分析以太网协议时，以以太网协议类型来进行判断，如果是 IP 协议类型，就表示上层协议类型是 IP 协议。然后，就可以继续分析 IP 协议。

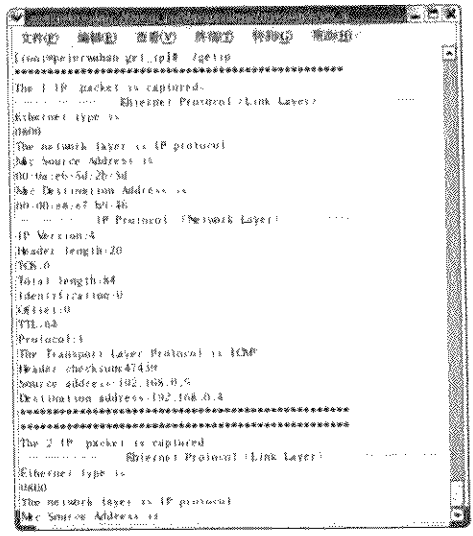

图 3-12　捕获和分析 IP 数据包

3.5.5　TCP 数据包的捕获

捕获 TCP 数据包，是在分析 IP 协议的基础上进行的，因为 TCP 协议是基于 IP 协议的。在上面的例子中可以看到，在分析 IP 协议时，有一个字段是协议字段，它表示的就是上层协议类型。如果它表示的是 TCP 协议，那么就可以继续分析 TCP 协议了。而分析 IP 协议是在分析以太网协议基础上进行的，所以也要分析以太网协议。

在这里以一个函数来实现分析 TCP 协议的功能，然后在分析 IP 协议的函数中被调用，而分析 IP 协议的函数在回调函数中被调用。实现分析 TCP 协议的函数定义为 tcp_protocol_packet_callback()，分析 IP 协议的函数定义为 ip_protocol_packet_callback()。回调函数定义为 ethernet_protocol_packet_callback()。这几个函数的类型都是一样的，其目的是为了表示分析的对象是同一个网络数据包。

捕获 TCP 数据包的源代码如下：

```
/*文件名字：get_tcp_code.c*/
#include "pcap.h"                          /*libpcap 头文件*/
/*以下是以太网协议格式的定义*/
struct ether_header
{
    u_int8_t  ether_dhost[6];              /*目的以太网地址*/
    u_int8_t  ether_shost[6];              /*源以太网地址*/
    u_int16_t ether_type;                  /*以太网类型*/
};
/*IP 地址格式的定义*/
typedef u_int32_t  in_addr_t;
```

```
struct in_addr
{
    in_addr_t s_addr;
};
/*IP 协议格式的定义*/
struct ip_header
{
#if de f  WORDS_BIGENDIAN
    u_int8_t      ip_version:4,                    /*版本*/
                  ip_header_length:4;              /*首部长度*/
    #else
    u_int8_t      ip_header_length:4,
                  ip_version:4;
    #endif
    u_int8_t   ip_tos_taowenliu;                   /*服务质量*/
    u_int16_t  ip_length;                          /*总长度*/
    u_int16_t  ip_id;                              /*标识*/
    u_int16_t  ip_off;                             /*偏移*/
    u_int8_t   ip_ttl;                             /*生存时间*/
    u_int8_t   ip_protocol;                        /*协议类型*/
    u_int16_t  ip_checksum;                        /*校验和*/
    struct in_addr   ip_souce_address;             /*源 IP 地址*/
    struct in_addr   ip_destination_address;       /*目的 IP 地址*/
};
/*下面是 TCP 协议格式的定义*/
struct tcp_header
{
    u_int16_t tcp_source_port;                     /*源端口*/
    u_int16_t tcp_destination_port;                /*目的端口*/
    u_int32_t tcp_acknowledgement;                 /*序列号*/
    u_int32_t tcp_ack;                             /*确认号*/
    #ifdef WORDS_BIGENDIAN
    u_int8_t  tcp_offset:4,                        /*偏移*/
              tcp_reserved:4;                      /*保留*/
    #else
    u_int8_t  tcp_reserved:4,                      /*保留*/
              tcp_offset:4;                        /*偏移*/
    #endif
    u_int8_t  tcp_flags;                           /*标志*/
    u_int16_t tcp_windows;                         /*窗口大小*/
    u_int16_t tcp_checksum;                        /*校验和*/
    u_int16_t tcp_urgent_pointer;                  /*紧急指针*/
};
```

```
/*下面是实现捕获和分析TCP数据包的函数定义*/
void tcp_protocol_packet_callback(u_char *argument,const struct pcap_pkthdr*
packet_header,const u_char* packet_content)
{
    struct tcp_header *tcp_protocol;                    /*TCP协议变量*/
    u_char flags;                                       /*标记*/
    int header_length;                                  /*首部长度*/
    u_short source_port;                                /*源端口号*/
    u_short destination_port;                           /*目的端口号*/
    u_short  windows;                                   /*窗口大小*/
    u_short urgent_pointer;                             /*紧急指针*/
    u_int sequence;                                     /*序列号*/
    u_int acknowledgement;                              /*确认号*/
    u_int16_t checksum;                                 /*校验和*/
    tcp_protocol = (struct tcp_header *) (packet_content+14+20);
        /*获得TCP协议数据内容，应该跳过以太网头和IP头部分*/
    source_port = ntohs(tcp_protocol->tcp_source_port);                /*获得源端口号*/
    destination_port = ntohs(tcp_protocol->tcp_destination_port);      /*获得目的端口号*/
    header_length = tcp_protocol->tcp_offset * 4;                      /*获得首部长度*/
    sequence = ntohl(tcp_protocol->tcp_acknowledgement);               /*获得序列号*/
    acknowledgement = ntohl(tcp_protocol->tcp_ack);                    /*获得确认号*/
    windows = ntohs(tcp_protocol->tcp_windows);                        /*获得窗口大小*/
    urgent_pointer = ntohs(tcp_protocol->tcp_urgent_pointer);          /*获得紧急指针*/
    flags = tcp_protocol->tcp_flags;                                   /*获得标记*/
    checksum =ntohs(tcp_protocol->tcp_checksum);                       /*获得校验和*/
    printf("-------  TCP Protocol  (Transport Layer)  -------\n");
    printf("Source Port:%d\n", source_port);                           /*输出源端口号*/
    printf("Destination Port:%d\n", destination_port);                 /*输出目的端口号*/
    switch (destination_port)                                          /*判断上层协议类型*/
        {
            case 80:printf("HTTP protocol\n"); break;
                /*端口是80，表示上层协议是HTTP协议*/
            case 21:printf("FTP protocol\n");break;
                /*端口是21，表示上层协议是FTP协议*/
            case 23:printf("TELNET protocol\n");break;
                /*端口是23，表示上层协议是Telnet协议*/
            case 25:printf("SMTP protocol\n");break;
                /*端口是25，表示上层协议是SMTP协议*/
            case 110:printf("POP3 protocol\n");break;
                /*端口是110，表示上层协议是POP3协议*/
            default:break;
            }
            printf("Sequence Number:%u\n", sequence);                  /*输出序列号*/
```

```
            printf("Acknowledgement Number:%u\n", acknowledgement);  /*输出确认号*/
            printf("Header Length:%d\n", header_length);             /*输出首部长度*/
            printf("Reserved:%d\n", tcp_protocol->tcp_reserved);
            printf("Flags:");                                        /*输出标记*/
        if (flags & 0x08) printf("PSH ");
        if (flags & 0x10) printf("ACK ");
        if (flags & 0x02) printf("SYN ");
        if (flags & 0x20) printf("URG ");
        if (flags & 0x01) printf("FIN ");
        if (flags & 0x04) printf("RST ");
            printf("\n");
            printf("Window Size:%d\n", windows);                     /*输出窗口大小*/
            printf("Checksum:%d\n", checksum);                       /*输出校验和*/
            printf("Urgent pointer:%d\n", urgent_pointer);           /*输出紧急指针*/
}
/*下面是实现分析 IP 协议的函数定义*/
void ip_protocol_packet_callback(u_char *argument,const struct pcap_pkthdr*
packet_header,const u_char* packet_content)
{
    struct ip_header *ip_protocol;                                   /*IP 协议变量*/
    u_int header_length;                                             /*首部长度*/
    u_int offset;                                                    /*偏移*/
    u_char tos;                                                      /*服务质量*/
    u_int16_t checksum;                                              /*校验和*/
    printf("-------  IP Protocol  (Network Layer)  --------\n");
    ip_protocol = (struct ip_header *) (packet_content+14);
        /*获得 IP 协议数据内容，应该跳过以太网协议头*/
    checksum = ntohs(ip_protocol->ip_checksum);                      /*获取校验和*/
    header_length = ip_protocol->ip_header_length*4;                 /*获取首部长度*/
    tos = ip_protocol->ip_tos_taowenliu;                             /*获取服务质量*/
    offset = ntohs(ip_protocol->ip_off);                             /*获取偏移量*/
        printf("IP Version:%d\n", ip_protocol->ip_version);          /*获取版本*/
        printf("Header length:%d\n", header_length);                 /*输出长度*/
        printf("TOS:%d\n", tos);                                     /*输出服务质量*/
        printf("Total length:%d\n", ntohs(ip_protocol->ip_length));  /*获取总长度*/
        printf("Identification:%d\n", ntohs(ip_protocol->ip_id));    /*获取标识*/
        printf("Offset:%d\n", (offset & 0x1fff) * 8);                /*输出偏移*/
        printf("TTL:%d\n", ip_protocol->ip_ttl);                     /*获取 TTL*/
        printf("Protocol:%d\n", ip_protocol->ip_protocol);           /*获取协议类型*/
    switch (ip_protocol->ip_protocol)                                /*判断上层协议类型*/
        {
            case 6:  printf("The Transport Layer Protocol is TCP\n");break;
            /*如果协议值是 6，表示上层协议是 TCP 协议*/
```

```
            case 17:  printf("The Transport Layer Protocol is UDP\n");break;
            /*如果协议值是 17，表示上层协议是 UDP 协议*/
            case 1:   printf("The Transport Layer Protocol is ICMP\n");break;
            /*如果是 1，表示上层协议是 ICMP 协议*/
            default:break;
        }
    printf("Header checksum:%d\n", checksum);
    printf("Source address:%s\n", inet_ntoa(ip_protocol->ip_souce_address));
        /*获得源 IP 地址*/
    printf("Destination address:%s\n", inet_ntoa(ip_protocol->ip_destination_address));
        /*获得目的 IP 地址*/
    switch (ip_protocol->ip_protocol)
        {
            case 6:
        tcp_protocol_packet_callback(argument,packet_header,packet_content);break;
        /*如果判断上层协议是 TCP 协议，就调用分析 TCP 协议的函数。注意此时的参数传递*/
        default:break;
        }
}
/*下面是实现分析以太网协议的函数的定义*/
void ethernet_protocol_packet_callback(u_char *argument,const struct pcap_pkthdr*
packet_header,const u_char* packet_content)
{
    u_short ethernet_type;                          /*以太网类型*/
    struct ether_header *ethernet_protocol;         /*以太网协议变量*/
    u_char *mac_string;                             /*以太网地址*/
    static int packet_number = 1;
    printf("*****************************************************\n");
    printf("The %d  TCP  packet is captured.\n",packet_number);
    printf("-------   Ethernet Protocol (Link Layer)   -----\n");
    ethernet_protocol = (struct ether_header *) packet_content;
        /*获得以太网协议数据内容*/
    printf("Ethernet type is :\n");
    ethernet_type= ntohs(ethernet_protocol->ether_type);
        /*获得以太网类型*/
    printf("%04x\n",ethernet_type);
    switch(ethernet_type)   /*判断以太网类型*/
    {
        case 0x0800:  printf("The network layer is IP protocol\n");break;
            /*判断上层协议为 IP 协议*/
        case 0x0806:  printf("The network layer is ARP protocol\n"); break;
            /*判断上层协议为 ARP 协议*/
        case 0x8035:  printf("The network layer is RARP protocol\n");break;
```

```
            /*判断上层协议为 RARP 协议*/
        default :break;
    }
    printf("Mac Source Address is : \n");
    mac_string=ethernet_protocol->ether_shost;
    printf("%02x:%02x:%02x:%02x:%02x:%02x\n",*mac_string,*(mac_string+1),
                *(mac_string+2),*(mac_string+3),
                *(mac_string+4),*(mac_string+5));
        /*获取源以太网地址*/
    printf("Mac Destination Address is : \n");
    mac_string=ethernet_protocol->ether_dhost;
    printf("%02x:%02x:%02x:%02x:%02x:%02x\n",*mac_string,*(mac_string+1),
                *(mac_string+2),*(mac_string+3),
                *(mac_string+4),*(mac_string+5));
        /*获取目的以太网地址*/
    switch(ethernet_type)
    {
        case 0x0800:
        ip_protocol_packet_callback(argument,
                                    packet_header,
                                    packet_content);
        break;
        /*判断上层协议为 IP 协议，所以调用分析 IP 协议的函数，注意参数传递*/
        default :break;
    }
    printf("*************************************************\n");
    packet_number++;
}
/*下面是主函数*/
void  main()
{
    pcap_t* pcap_handle;                                /*Libpcap 句柄*/
    char error_content[PCAP_ERRBUF_SIZE];               /*存储错误信息*/
    char *net_interface;                                /*网络接口*/
    struct bpf_program bpf_filter;                      /*BPF 过滤规则*/
    char bpf_filter_string[] = "tcp";
    /*过滤规则字符串，此时表示本程序只捕获 TCP 协议的网络数据包*/
    bpf_u_int32 net_mask;                               /*网络掩码*/
    bpf_u_int32 net_ip;                                 /*网络地址*/
    net_interface = pcap_lookupdev(error_content);      /*获得网络接口*/
    pcap_lookupnet(net_interface,
                   &net_ip,
                   &net_mask,
```

```
                error_content);                         /*获得网络地址和网络掩码*/
    pcap_handle = pcap_open_live(net_interface,
                                 BUFSIZ,
                                 1,
                                 0,
                                 error_content);        /*打开网络设备*/
    pcap_compile(pcap_handle,
                 &bpf_filter,
                 bpf_filter_string,
                 0,
                 net_ip);                               /*编译过滤规则*/
    pcap_setfilter(pcap_handle, &bpf_filter);           /*设置过滤规则*/
    if(pcap_datalink(pcap_handle)!=DLT_EN10MB)
        return;
    pcap_loop(pcap_handle,
              -1,
              ethernet_protocol_packet_callback,
              NULL);
        /*注册回调函数，循环捕获网络数据包，利用回调函数对数据包进行处理*/
        pcap_close(pcap_handle);
        /*关闭Libpcap操作*/
}
```

本程序的功能是捕获和分析 TCP 网络数据包，传输层协议 TCP 基于网络层协议 IP，而网络层协议 IP 又基于链路层以太网协议。所以，在分析数据包时，首先分析以太网协议部分，再根据以太网协议中的类型字段，进一步分析 IP 协议，最后根据 IP 协议中的协议类型字段，进一步分析 TCP 协议。

在本程序中，共定义了三个函数，它们分别实现了 TCP、IP、以太网协议的分析。其中，回调函数 ethernet_protocol_packet_callback()实现以太网协议的分析，而在此回调函数中，又调用了函数 ip_protocol_packet_callback()来实现 IP 协议的分析，然后又在此函数中调用分析 TCP 协议的函数 tcp_protocol_packet_callback()。

编译此程序，其编译命令如下：

```
Gcc -o gettcp get_tcp_code.c -lpcap
```

运行此程序，输出内容如下：

```
[root@peterwuhan get_tcp]# ./gettcp
**********************************************
The 1  TCP  packet is captured.
-------    Ethernet Protocol (Link Layer)    -----
Ethernet type is :
0800
The network layer is IP protocol
Mac Source Address is :
00:0a:e6:5d:2b:3d
```

```
Mac Destination Address is :
00:00:e8:e7:b9:46
-------  IP Protocol  (Network Layer)  --------
IP Version:4
Header length:20
TOS:16
Total length:60
Identification:25669
Offset:0
TTL:64
Protocol:6
The Transport Layer Protocol is TCP
Header checksum:21773
Source address:192.168.0.5
Destination address:192.168.0.4
-------  TCP Protocol  (Transport Layer)  -------
Source Port:34841
Destination Port:135
Sequence Number:2384117109
Acknowledgement Number:0
Header Length:40
Reserved:0
Flags:SYN
Window Size:5840
Checksum:621
Urgent pointer:0
************************************************
************************************************
The 2  TCP  packet is captured.
-------   Ethernet Protocol (Link Layer)   -----
Ethernet type is :
0800
The network layer is IP protocol
Mac Source Address is :
00:0a:e6:5d:2b:3d
Mac Destination Address is :
00:00:e8:e7:b9:46
-------  IP Protocol  (Network Layer)  --------
IP Version:4
Header length:20
TOS:16
Total length:52
Identification:25670
```

```
Offset:0
TTL:64
Protocol:6
The Transport Layer Protocol is TCP
Header checksum:21780
Source address:192.168.0.5
Destination address:192.168.0.4
-------- TCP Protocol (Transport Layer) ---------
Source Port:34841
Destination Port:135
Sequence Number:2384117110
Acknowledgement Number:1397715131
Header Length:32
Reserved:0
Flags:ACK
Window Size:5840
Checksum:27927
Urgent pointer:0
*****************************************************
```

程序的运行结果如图 3-13 所示。

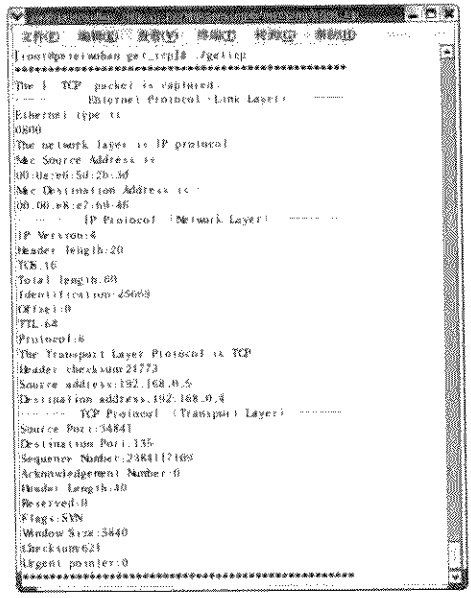

图 3-13　捕获和分析 TCP 数据包

从运行结果分析，此程序捕获和分析了 TCP 协议数据包，其分析的过程从以太网开始，然后是 IP 协议，最后是对 TCP 协议的分析。此程序把一个 TCP 数据包内容的三部分协议都分析出来了。

3.5.6 UDP 数据包的捕获

对于 UDP 协议数据，使用 Libpcap 中的函数 pcap_loop()进行循环捕获，然后对捕获到的网络数据包进行分析。由于 UDP 协议是基于 IP 协议的，所以，在分析 UDP 协议时，同样要分析 IP 协议。又由于 IP 协议是基于以太网协议，所以 UDP 数据的分析过程是最开始分析以太网协议，再分析 IP 协议，最后分析 UDP 协议。

分析 UDP 协议的功能由自定义函数 udp_protocol_packet_callback() 来完成，分析 IP 协议的功能由自定义函数 ip_protocol_packet_callback()来完成，而分析以太网协议的功能由函数 ethernet_protocol_callback() 来完成。同时，函数 ethernet_protocol_callback() 也是 pcap_loop()注册的回调函数。

捕获和分析 UDP 数据包的源代码如下：

```
#include "pcap.h"                          /*Libpcap 头文件*/
    /*下面是以太网协议格式的定义*/
struct ether_header
{
    u_int8_t  ether_dhost[6];              /*目的以太网地址*/
    u_int8_t  ether_shost[6];              /* 源以太网地址*/
    u_int16_t ether_type;                  /* 以太网类型 */
};
/*下面是 IP 地址格式的定义*/
typedef u_int32_t  in_addr_t;
struct in_addr
{
    in_addr_t s_addr;
};
/*下面是 IP 协议格式的定义*/
struct ip_header
{
    #if defined(WORDS_BIGENDIAN)
    u_int8_t     ip_version:4,             /*版本*/
                 ip_header_length:4;       /*首部长度*/
    #else
    u_int8_t     ip_header_length:4,
                 ip_version:4;
    #endif
    u_int8_t   ip_tos;                     /*服务质量*/
    u_int16_t  ip_length;                  /*总长度*/
    u_int16_t  ip_id;                      /*标识*/
    u_int16_t  ip_off;                     /*偏移*/
```

```
    u_int8_t    ip_ttl;                          /*生存时间*/
    u_int8_t    ip_protocol;                     /*协议类型*/
    u_int16_t   ip_checksum;                     /*校验和*/
    struct in_addr   ip_souce_address;           /*源 IP 地址*/
    struct in_addr   ip_destination_address;     /*目的 IP 地址*/
};
/*下面是 UDP 协议格式的定义*/
struct udp_header
{
    u_int16_t udp_source_port;                   /*源端口号*/
    u_int16_t udp_destination_port;              /*目的端口号*/
    u_int16_t udp_length;                        /*长度*/
    u_int16_t udp_checksum;                      /*校验和*/
};
/*下面是实现捕获和分析 UDP 协议数据包的函数定义*/
void udp_protocol_packet_callback(u_char *argument,const struct pcap_pkthdr*
packet_header,const u_char* packet_content)
{
    struct udp_header *udp_protocol;             /*UDP 协议数据变量*/
    u_short source_port;                         /*源端口号*/
    u_short  destination_port;                   /*目的端口号*/
    u_short length;                              /*长度*/
    udp_protocol = (struct udp_header *) (packet_content+14+20);   /*获得 UDP 协议数据内容*/
    source_port = ntohs(udp_protocol->udp_source_port);            /*获得源端口号*/
    destination_port = ntohs(udp_protocol->udp_destination_port);  /*获得目的端口号*/
    length = ntohs(udp_protocol->udp_length);                      /*获得长度*/
    printf("---------- UDP Protocol (Transport Layer) ----------\n");
    printf("Source port:%d\n", source_port);
    printf("Destination port:%d\n", destination_port);
    switch (destination_port)
        /*根据端口号判断应用层协议类型*/
        {
            case 138: printf("NETBIOS Datagram Service\n");  break;
            /*端口号为 138，表示上层协议为 NetBIOS 数据报服务*/
            case 137: printf("NETBIOS Name Service\n");break;
            /*端口号为 137，表示上层协议为 NetBIOS 名字服务*/
            case 139: printf("NETBIOS session service\n");break;
            /*端口号为 139，表示上层协议为 NetBIOS 会话服务*/
            case 53:  printf("name-domain server \n");break;
            /*端口号为 53，表示上层协议为域名服务*/
            default:break;                       /*其他没有分析*/
        }
    printf("Length:%d\n", length);
```

```
    printf("Checksum:%d\n", ntohs(udp_protocol->udp_checksum));
        /*获得校验和*/
}
/*下面是分析 IP 协议的函数的定义*/
void ip_protocol_packet_callback(u_char *argument,const struct pcap_pkthdr*
packet_header,const u_char* packet_content)
{
    struct ip_header *ip_protocol;                  /*IP 协议变量*/
    u_int header_length;                            /*首部长度*/
    u_int offset;                                   /*偏移*/
    u_char tos;                                     /*服务质量*/
    u_int16_t checksum;                             /*校验和*/
    printf("---------- IP Protocol (Network Layer) ----------\n");
    ip_protocol = (struct ip_header *) (packet_content+14);      /*获得 IP 协议数据内容*/
    checksum = ntohs(ip_protocol->ip_checksum);                  /*获得校验和*/
    header_length = ip_protocol->ip_header_length*4;             /*获得首部长度*/
    tos = ip_protocol->ip_tos;                                   /*服务质量*/
    offset = ntohs(ip_protocol->ip_off);                         /*获得偏移量*/
        printf("IP Version:%d\n", ip_protocol->ip_version);      /*获得版本号*/
        printf("Header length:%d\n", header_length);
        printf("TOS:%d\n", tos);
        printf("Total length:%d\n", ntohs(ip_protocol->ip_length));  /*获得总长度*/
        printf("Identification:%d\n", ntohs(ip_protocol->ip_id));    /*获得标识*/
        printf("Offset:%d\n", (offset & 0x1fff) * 8);
        printf("TTL:%d\n", ip_protocol->ip_ttl);                     /*获得 TTL*/
        printf("Protocol:%d\n", ip_protocol->ip_protocol);           /*获得协议类型*/
    switch (ip_protocol->ip_protocol)                                /*根据协议类型判断*/
        {
            case 6:  printf("The Transport Layer Protocol is TCP\n");break;
            /*上层协议为 TCP 协议*/
            case 17: printf("The Transport Layer Protocol is UDP\n");break;
            /*上层协议为 UDP 协议*/
            case 1:  printf("The Transport Layer Protocol is ICMP\n");break;
            /*上层协议为 ICMP 协议*/
            default:break;
        }
    printf("Header checksum:%d\n", checksum);
    printf("Source address:%s\n", inet_ntoa(ip_protocol->ip_souce_address));
        /*获得源 IP 地址*/
    printf("Destination address:%s\n", inet_ntoa(ip_protocol->ip_destination_address));
        /*获得目的 IP 地址*/
    switch (ip_protocol->ip_protocol)
        {
```

```
            case 17:
        udp_protocol_packet_callback(argument,
                                     packet_header,
                                     packet_content);
        break;
        /*如果上层协议为 UDP 协议，就调用分析 UDP 协议的函数，注意此时的参数传递形式*/
        default:break;
        }
}
/*下面是分析以太网协议的回调函数的定义*/
void ethernet_protocol_packet_callback(u_char *argument,const struct pcap_pkthdr*
packet_header,const u_char* packet_content)
{
    u_short ethernet_type;                          /*以太网类型*/
    struct ether_header *ethernet_protocol;         /*以太网协议变量*/
    u_char *mac_string;                             /*以太网地址*/
    static int packet_number = 1;
    printf("**************************************************\n");
    printf("The %d  UDP  packet is captured.\n",packet_number);
    printf("--------  Ethernet Protocol (Link Layer)   --------\n");
        ethernet_protocol = (struct ether_header *) packet_content;
        /*获得以太网协议数据内容*/
    printf("Ethernet type is :\n");
    ethernet_type= ntohs(ethernet_protocol->ether_type);
        /*获得以太网类型*/
    printf("%04x\n",ethernet_type);
    switch(ethernet_type)
    {
        case 0x0800:  printf("The network layer is IP protocol\n");break;
            /*上层协议为 IP 协议*/
        case 0x0806:  printf("The network layer is ARP protocol\n"); break;
            /*上层协议为 ARP 协议*/
        case 0x8035:  printf("The network layer is RARP protocol\n");break;
            /*上层协议为 RARP 协议*/
        default :break;
    }
    printf("Mac Source Address is : \n");
    mac_string=ethernet_protocol->ether_shost;
    printf("%02x:%02x:%02x:%02x:%02x:%02x\n",*mac_string,*(mac_string+1),
                *(mac_string+2),*(mac_string+3),
                *(mac_string+4),*(mac_string+5));
        /*获得源以太网地址*/
    printf("Mac Destination Address is : \n");
```

```
    mac_string=ethernet_protocol->ether_dhost;
    printf("%02x:%02x:%02x:%02x:%02x:%02x\n",*mac_string,*(mac_string+1),
                *(mac_string+2),*(mac_string+3),
                *(mac_string+4),*(mac_string+5));
        /*获得目的以太网地址*/
    switch(ethernet_type)
    {
        case 0x0800:
        ip_protocol_packet_callback(argument,
                            packet_header,
                            packet_content);
        break;
        /*上层协议为 IP 协议，就调用分析 IP 协议的函数，注意参数的传递*/
        default :break;
    }
    printf("*************************************************\n");
    packet_number++;
}
/*下面是主函数的定义*/
void  main()
{
    pcap_t* pcap_handle;                                /*Libpcap 的句柄*/
    char error_content[PCAP_ERRBUF_SIZE];               /*存储错误信息*/
    char *net_interface;                                /*网络接口*/
    struct bpf_program bpf_filter;                      /*BPF 过滤规则*/
    char bpf_filter_string[] = "udp";
    /*过滤规则字符串，此时表示本程序只捕获 UDP 网络数据包*/
    bpf_u_int32 net_mask;                               /*网络掩码*/
    bpf_u_int32 net_ip;                                 /*网络地址*/
    net_interface = pcap_lookupdev(error_content);      /*获得网络接口*/
    pcap_lookupnet(net_interface,
                &net_ip,
                &net_mask,
                error_content);                         /*获得网络地址和网络掩码*/
    pcap_handle = pcap_open_live(net_interface,
                                 BUFSIZ,
                                 1,
                                 0,
                                 error_content);        /*打开网络接口*/
    pcap_compile(pcap_handle,
                 &bpf_filter,
                 bpf_filter_string,
                 0,
```

```
TTL:128
Protocol:17
The Transport Layer Protocol is UDP
Header checksum:41788
Source address:192.168.0.4
Destination address:192.168.0.255
-------- UDP Protocol (Transport Layer) --------
Source port:138
Destination port:138
NETBIOS Datagram Service
Length:209
Checksum:36026
*****************************************
```

程序的运行结果如图 3-14 所示。

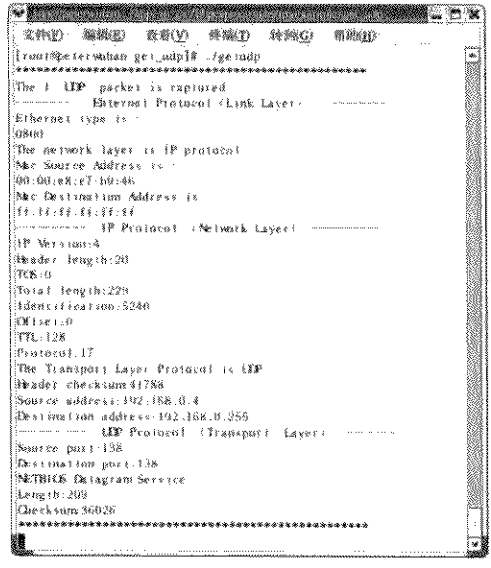

图 3-14 捕获和分析 UDP 数据包

从运行结果分析，此程序实现了捕获 UDP 数据包的功能，并且对捕获到的 UDP 数据包进行了详细的分析，即 UDP 数据包包括三个部分：以太网协议部分、IP 协议部分和 UDP 协议部分。

3.5.7 ICMP 数据包的捕获

Internet 控制报文协议（ICMP）是一个应用非常广泛的协议。ICMP 协议是基于 IP 协议

的，在 IP 报文中传输，所以必须对 IP 协议进行分析，然后再分析 ICMP 协议。而 IP 协议是在以太网帧中传输的，所以又应该先分析以太网协议。

由于 ICMP 报文类型非常多，在此只举例说明其中一种类型，即回显报文。ICMP 回显报文是 ping 程序使用的报文，也是最典型的 ICMP 报文形式，其他 ICMP 报文的分析与此类似。

ICMP 数据包的捕获也是使用 Libpcap 中的函数 pcap_loop() 来实现的。先分析以太网协议函数 ethernet_protocol_packet_callback()，再在此函数中调用分析 IP 协议的函数 ip_protocol_packet_callback ()，然后在此函数中调用分析 ICMP 协议的函数 icmp_protocol_packet_callback()。

捕获和分析 ICMP 数据包的源代码如下：

```
/*文件名字：get_icmp_code.c*/
#include "pcap.h"
/*下面是以太网协议格式的定义*/
struct ether_header
{
    u_int8_t  ether_dhost[6];          /*目的以太网地址*/
    u_int8_t  ether_shost[6];          /*源以太网地址*/
    u_int16_t ether_type;              /*以太网类型*/
};
/*下面是 IP 地址格式的定义*/
typedef u_int32_t  in_addr_t;
struct in_addr
{
    in_addr_t s_addr;
};
/*下面是 IP 协议格式的定义*/
struct ip_header
{
    #if defined(WORDS_BIGENDIAN)
    u_int8_t     ip_version:4,                /*版本号*/
                 ip_header_length:4;          /*首部长度*/
    #else
    u_int8_t     ip_header_length:4,          /*首部长度*/
                 ip_version:4;                /*版本号*/
    #endif
    u_int8_t   ip_tos;                        /*服务质量*/
    u_int16_t  ip_length;                     /*总长度*/
    u_int16_t  ip_id;                         /*标识*/
    u_int16_t  ip_off;                        /*偏移*/
    u_int8_t   ip_ttl;                        /*生存时间*/
    u_int8_t   ip_protocol;                   /*协议类型*/
    u_int16_t  ip_checksum;                   /*校验和*/
    struct in_addr  ip_souce_address;         /*源 IP 地址*/
```

```
    struct in_addr   ip_destination_address;   /*目的IP地址*/
};
/*下面是ICMP协议格式的定义*/
struct icmp_header
{
    u_int8_t icmp_type;                        /*ICMP类型*/
    u_int8_t icmp_code;                        /*ICMP代码*/
    u_int16_t icmp_checksum;                   /*校验和*/
    u_int16_t icmp_id_lliiuuwweennttaaoo;      /*标识符*/
    u_int16_t icmp_sequence;                   /*序列号*/
};
/*下面是实现分析ICMP协议的函数定义*/
void icmp_protocol_packet_callback(u_char *argument,const struct pcap_pkthdr*
packet_header,const u_char* packet_content)
{
    struct icmp_header *icmp_protocol;         /*ICMP协议变量*/
    icmp_protocol = (struct icmp_header *) (packet_content+14+20);
        /*获取ICMP协议数据内容，跳过以太网和IP协议部分*/
    printf("----------  ICMP Protocol  (Transport Layer)  ----------\n");
    printf("ICMP Type:%d\n", icmp_protocol->icmp_type);
        /*获得ICMP类型*/
    switch (icmp_protocol->icmp_type)  /*根据ICMP类型进行判断*/
    {
        case 8:
        /*类型为8，表示是回显请求报文*/
        printf("ICMP Echo Request Protocol \n");
        printf("ICMP Code:%d\n", icmp_protocol->icmp_code);
            /*获得ICMP代码*/
        printf("Identifier:%d\n", icmp_protocol->icmp_id_lliiuuwweennttaaoo);
            /*获得标识符*/
        printf("Sequence Number:%d\n", icmp_protocol->icmp_sequence);
            /*获得序列号*/
        break;
        case 0:
            /*类型为0，表示是回显应答报文*/
        printf("ICMP Echo Reply Protocol \n");
        printf("ICMP Code:%d\n", icmp_protocol->icmp_code);
            /*获得ICMP代码*/
        printf("Identifier:%d\n", icmp_protocol->icmp_id_lliiuuwweennttaaoo);
            /*获得标识符*/
        printf("Sequence Number:%d\n", icmp_protocol->icmp_sequence);
            /*获得序列号*/
        break;
```

```
        default :break;                    /*类型为其他值，在这里没有分析*/
    }
    printf("ICMP Checksum:%d\n", ntohs(icmp_protocol->icmp_checksum));
    /*获得校验和*/
}
/*下面是实现分析 IP 协议的函数定义*/
void ip_protocol_packet_callback(u_char *argument,const struct pcap_pkthdr*
packet_header,const u_char* packet_content)
{
    struct ip_header *ip_protocol;        /*IP 协议变量*/
    u_int header_length;                  /*首部长度*/
    u_int offset;                         /*偏移*/
    u_char tos;                           /*服务质量*/
    u_int16_t checksum;                   /*校验和*/
    printf("---------- IP Protocol (Network Layer) ----------\n");
    ip_protocol = (struct ip_header *) (packet_content+14);
        /*获得 IP 协议数据内容，跳过以太网协议部分*/
    checksum = ntohs(ip_protocol->ip_checksum);              /*获得校验和*/
    header_length = ip_protocol->ip_header_length*4;         /*获得首部长度*/
    tos = ip_protocol->ip_tos;                               /*获得服务质量*/
    offset = ntohs(ip_protocol->ip_off);                     /*获得偏移*/
    printf("IP Version:%d\n", ip_protocol->ip_version);      /*获得版本*/
    printf("Header length:%d\n", header_length);
    printf("TOS:%d\n", tos);
    printf("Total length:%d\n", ntohs(ip_protocol->ip_length));       /*获得总长度*/
    printf("Identification:%d\n", ntohs(ip_protocol->ip_id));
    printf("Offset:%d\n", (offset & 0x1fff) * 8);
    printf("TTL:%d\n", ip_protocol->ip_ttl);                     /*获得 TTL*/
    printf("Protocol:%d\n", ip_protocol->ip_protocol);           /*获得协议类型*/
    switch (ip_protocol->ip_protocol)  /*判断协议类型*/
    {
        case 6:  printf("The Transport Layer Protocol is TCP\n");break;
            /*上层协议为 TCP 协议*/
        case 17: printf("The Transport Layer Protocol is UDP\n");break;
            /*上层协议为 UDP 协议*/
        case 1:  printf("The Transport Layer Protocol is ICMP\n");break;
            /*上层协议为 ICMP 协议*/
        default:break;
    }
    printf("Header checksum:%d\n", checksum);
    printf("Source address:%s\n", inet_ntoa(ip_protocol->ip_souce_address));
        /*获得源 IP 地址*/
    printf("Destination address:%s\n", inet_ntoa(ip_protocol->ip_destination_address));
```

```
        /*获得目的IP地址*/
    switch (ip_protocol->ip_protocol)
    {
        case 1:
        icmp_protocol_packet_callback(argument,
                                      packet_header,
                                      packet_content);
        break;
        /*如果上层协议为ICMP协议，就调用分析ICMP协议的函数，注意此时的参数传递形式，它表示分析的对
        象是同一个网络数据包*/
        default:break;
    }
}
/*下面是实现分析以太网协议的函数定义，也是回调函数*/
void ethernet_protocol_packet_callback(u_char *argument,const struct pcap_pkthdr*
packet_header,const u_char* packet_content)
{
    u_short ethernet_type;                          /*以太网类型*/
    struct ether_header *ethernet_protocol;         /*以太网协议*/
    u_char *mac_string;                             /*以太网地址*/
    static int packet_number = 1;
    printf("**************************************************\n");
    printf("The %d  ICMP  packet is captured.\n",packet_number);
    printf("--------  Ethernet Protocol (Link Layer)   --------\n");
    ethernet_protocol = (struct ether_header *) packet_content;  /*获得以太网协议数据内容*/
    printf("Ethernet type is :\n");
    ethernet_type= ntohs(ethernet_protocol->ether_type);        /*获得以太网类型*/
    printf("%04x\n",ethernet_type);
    switch(ethernet_type)                                       /*根据以太网类型进行判断*/
    {
        case 0x0800:  printf("The network layer is IP protocol\n");break;
            /*上层协议为IP协议*/
        case 0x0806:  printf("The network layer is ARP protocol\n"); break;
            /*上层协议为ARP协议*/
        case 0x8035:  printf("The network layer is RARP protocol\n");break;
            /*上层协议为RARP协议*/
        default :break;
    }
    printf("Mac Source Address is : \n");
    mac_string=ethernet_protocol >ether_shost;
    printf("%02x:%02x:%02x:%02x:%02x:%02x\n",*mac_string,*(mac_string+1),
                *(mac_string+2),*(mac_string+3),
                *(mac_string+4),*(mac_string+5));
```

```
        /*获得源以太网地址*/
    printf("Mac Destination Address is : \n");
    mac_string=ethernet_protocol->ether_dhost;
    printf("%02x:%02x:%02x:%02x:%02x:%02x\n",*mac_string,*(mac_string+1),
                *(mac_string+2),*(mac_string+3),
                *(mac_string+4),*(mac_string+5));
        /*获得目的以太网地址*/
    switch(ethernet_type)
    {
        case 0x0800:
        ip_protocol_packet_callback(argument,
                                    packet_header,
                                    packet_content);
        break;
        /*如果上层协议是 IP 协议，就调用分析 IP 协议的函数*/
        default :break;
    }
    printf("**************************************************\n");
    packet_number++;
}
void  main()
{
    pcap_t* pcap_handle;                                /*Libpcap 句柄*/
    char error_content[PCAP_ERRBUF_SIZE];               /*存储错误信息*/
    char *net_interface;                                /*网络接口*/
    struct bpf_program bpf_filter;                      /*BPF 过滤规则*/
    char bpf_filter_string[] = "icmp";
    /*过滤规则字符串，此时表示本程序只捕获 ICMP 网络数据包*/
    bpf_u_int32 net_mask;                               /*网络掩码*/
    bpf_u_int32 net_ip;                                 /*网络地址*/
    net_interface = pcap_lookupdev(error_content);      /*获得网络接口*/
    pcap_lookupnet(net_interface,
                   &net_ip,
                   &net_mask,
                   error_content);                      /*获得网络地址和网络掩码*/
    pcap_handle = pcap_open_live(net_interface,
                                 BUFSIZ,
                                 1,
                                 0,
                                 error_content);        /*打开网络接口*/
    pcap_compile(pcap_handle,
                 &bpf_filter,
                 bpf_filter_string,
```

```
                    0,
                    net_ip);                          /*编译过滤规则*/
    pcap_setfilter(pcap_handle,
                &bpf_filter);                         /*设置过滤规则*/
    if(pcap_datalink(pcap_handle)!=DLT_EN10MB)
            return;
    pcap_loop(pcap_handle,
            -1,
            ethernet_protocol_packet_callback,
            NULL);
    /*注册回调函数，循环捕获网络数据包，然后调用回调函数对网络数据包进行处理*/
    pcap_close(pcap_handle);                          /*关闭Libpcap操作*/
}
```

本程序的功能是实现捕获和分析 ICMP 数据包，它的过程主要由三个函数来完成。回调函数先完成了对以太网协议的分析，再调用分析 IP 协议的函数，最后调用分析 ICMP 协议的函数。

在分析 ICMP 协议时，只对 ICMP 回显报文进行了分析，包括回显请求和回显应答。回显请求的 ICMP 类型为 8，回显应答的 ICMP 报文类型为 0。

编译此程序，编译命令如下：

```
gcc -o geticmp get_icmp_code.c -lpcap
```

运行此程序，其输出内容如下：

```
***********************************************
The 1  ICMP  packet is captured.
--------  Ethernet Protocol (Link Layer)   --------
Ethernet type is :
0800
The network layer is IP protocol
Mac Source Address is :
00:0a:e6:5d:2b:3d
Mac Destination Address is :
00:00:e8:e7:b9:46
----------  IP Protocol  (Network Layer)  ----------
IP Version:4
Header length:20
TOS:0
Total length:84
Identification:0
Offset:0
TTL:64
Protocol:1
The Transport Layer Protocol is ICMP
Header checksum:47439
```

```
Source address:192.168.0.5
Destination address:192.168.0.4
---------- ICMP Protocol (Transport Layer) ----------
ICMP Type:8
ICMP Echo Request Protocol
ICMP Code:0
Identifier:4987
Sequence Number:256
ICMP Checksum:64837
**************************************************
**************************************************
The 2 ICMP packet is captured.
-------- Ethernet Protocol (Link Layer) --------
Ethernet type is :
0800
The network layer is IP protocol
Mac Source Address is :
00:00:e8:e7:b9:46
Mac Destination Address is :
00:0a:e6:5d:2b:3d
---------- IP Protocol (Network Layer) ----------
IP Version:4
Header length:20
TOS:0
Total length:84
Identification:5209
Offset:0
TTL:128
Protocol:1
The Transport Layer Protocol is ICMP
Header checksum:25846
Source address:192.168.0.4
Destination address:192.168.0.5
-- -- -- ICMP Protocol (Transport Layer) ----------
ICMP Type:0
ICMP Echo Reply Protocol
ICMP Code:0
Identifier:4987
Sequence Number:256
ICMP Checksum:1350
**************************************************
```

运行结果如图 3-15 所示。

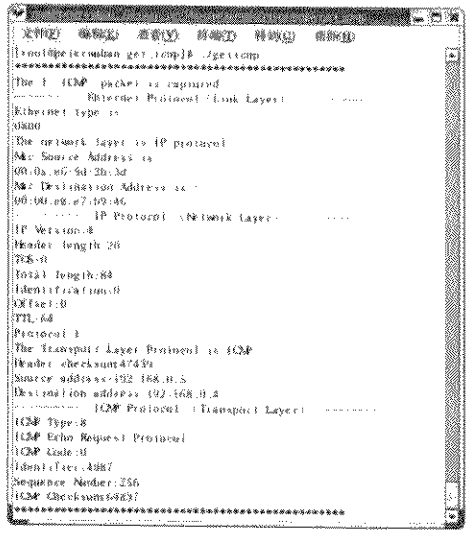

图 3-15　捕获和分析 ICMP 数据包

从运行结果可以分析得到，本程序捕获和分析了 ICMP 网络数据包，包括三部分内容，分别是以太网协议部分、IP 协议部分和 ICMP 协议部分。

3.5.8　综合程序

上面所举的例子，都是对某一个特定的网络数据包进行捕获和分析，但是网络中的数据包种类是极其丰富的。要判断捕获到的任何一个网络数据包是什么类型的网络数据包，它里面使用的是什么协议，由哪几个协议部分组成。

要解决这些问题，就不应限定捕获的对象，而是要捕获所有的网络数据包，然后对捕获的网络数据包进行分析，判断其是什么类型的网络数据包。

在前面的例子中，都是通过设置过滤规则来捕获特定的数据包的。例如，在捕获和分析 IP 网络数据包时，过滤规则设为“ip”，表示只捕获 IP 协议数据包；在捕获和分析 ARP 网络数据包时，过滤规则设为“arp”，表示只捕获 ARP 协议的数据包；在捕获和分析 TCP 网络数据包时，过滤规则设为“tcp”，表示只捕获 TCP 协议的数据包；在捕获和分析 UDP 网络数据包时，过滤规则设为“udp”，表示只捕获 UDP 协议的数据包；在捕获和分析 ICMP 网络数据包时，过滤规则设为“icmp”，表示只捕获 ICMP 协议的数据包。

要实现捕获所有的数据包的功能，只需把过滤规则设为空即可。

对所有网络数据包的分析，必须从链路层开始分析，然后分析网络层，再分析传输层，最后分析应用层。这是由于捕获到的网络数据包是一个数据帧，其中最外层的协议是链路层协议，然后从里到外分别是网络层、传输层和应用层，如图 3-16 所示。

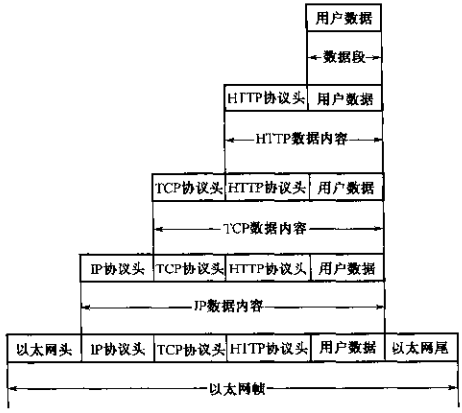

图 3-16　数据帧的分解过程

从图 3-16 可以看出，捕获到的是一个以太网帧，对于链路层本程序只分析了以太网协议。去掉以太网协议部分，剩下的是 IP 协议数据；再分析 IP 协议，去掉 IP 协议部分，剩下的是 TCP 或 UDP 协议数据；再分析 TCP 协议或 UDP 协议等传输层协议，然后去掉传输层协议部分，剩下的是应用层协议；最后分析应用层协议。

具体的分析流程如图 3-17 所示。

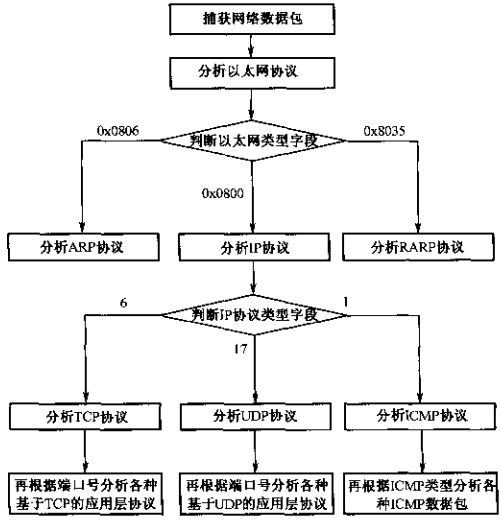

图 3-17　数据包的分析流程

从流程图可以看到，在捕获了数据包之后，就对网络数据包进行分析，在这里链路层只分析了以太网协议。根据以太网协议内容的分析结果，判断以太网类型的值：如果是0x0806，表示协议为 ARP 协议，则分析 ARP 协议；如果是 0x0800，表示协议为 IP 协议，则分析 IP 协议，如果是 0x8035，表示协议为 RARP 协议，则分析 RARP 协议。在分析 IP 协议时，根据协议类型的值判断传输层协议类型：如果 IP 协议类型字段的值是 6，表示协议为 TCP 协议，则分析 TCP 协议；如果 IP 协议类型字段的值是 17，表示协议为 UDP 协议，则分析 UDP 协议；如果 IP 协议类型字段的值是 1，表示协议为 ICMP 协议，则分析 ICMP 协议。在分析 TCP 或者 UDP 协议时，可以根据其端口号进一步分析应用层的协议类型。在分析 ICMP 协议时，可以根据其 ICMP 类型来判断是什么类型的 ICMP 数据包。

下面所举的例子，可捕获所有的网络数据包，分析了数据包的各种网络协议，包括以太网协议、ARP 协议、RARP 协议、IP 协议、TCP 协议、UDP 协议和 ICMP 协议。分析以太网协议的函数定义为 ethernet_protocol_packet_callback()，它也是回调函数；分析 ARP 协议的函数定义为 arp_protocol_packet_callback()；分析 IP 协议的函数定义为 ip_protocol_packet_callback()；分析 TCP 协议的函数定义为 tcp_protocol_packet_callback()；分析 UDP 协议的函数定义为 udp_protocol_packet_callback()；分析 ICMP 协议的函数定义为 icmp_protocol_packet_callback()。这几个函数的类型都是一样的，其目的是为了表示同一个网络数据包，它们的调用关系如图 3-18 所示。

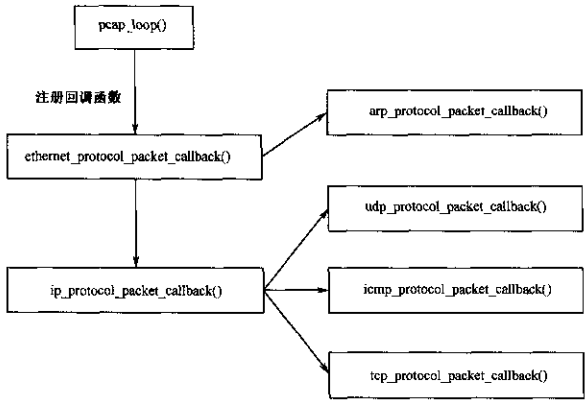

图 3-18　函数的调用关系

下面是此程序的源代码：

```
/*文件名字: get_packet_code.c  刘文涛编写*/
#include "pcap.h"                          /*Libpcap 的头文件*/
/*下面是以太网协议格式的定义*/
struct ether_header
{
    u_int8_t  ether_dhost[6];              /*目的以太网地址*/
    u_int8_t  ether_shost[6];              /*源以太网地址*/
```

```
    u_int16_t ether_type;                    /*以太网类型*/
};
/*下面是IP地址格式的定义*/
typedef u_int32_t  in_addr_t;
struct in_addr
{
    in_addr_t s_addr;
};
/*下面是ARP协议格式的定义*/
struct arp_header
{
    u_int16_t  arp_hardware_type;                    /*硬件类型*/
    u_int16_t  arp_protocol_type;                    /*协议类型*/
    u_int8_t   arp_hardware_length;                  /*硬件地址长度*/
    u_int8_t   arp_protocol_length;                  /*协议地址长度*/
    u_int16_t  arp_operation_code;                   /* ARP操作码*/
    u_int8_t arp_source_ethernet_address[6];         /*源以太网地址*/
    u_int8_t arp_source_ip_address [4];              /*源IP地址*/
    u_int8_t arp_destination_ethernet_address [6];   /*目的以太网地址*/
    u_int8_t arp_destination_ip_address[4];          /*目的IP地址*/
};
/*下面是IP协议格式的定义*/
struct ip_header
{
    #if defined(WORDS_BIGENDIAN)
    u_int8_t      ip_version:4,                      /*版本*/
                  ip_header_length:4;                /*首部长度*/
    #else
    u_int8_t      ip_header_length:4,                /*首部长度*/
                  ip_version:4;                      /*版本*/
    #endif
    u_int8_t   ip_tos;                               /*服务质量*/
    u_int16_t  ip_length;                            /*总长度*/
    u_int16_t  ip_id;                                /*标识*/
    u_int16_t  ip_off;                               /*偏移*/
    u_int8_t   ip_ttl;                               /*生存时间*/
    u_int8_t   ip_protocol;                          /*协议类型*/
    u_int16_t  ip_checksum;                          /*校验和*/
    struct in_addr  ip_souce_address;                /*源IP地址*/
    struct in_addr  ip_destination_address;          /*目的IP地址*/
};
/*下面是UDP协议格式的定义*/
struct udp_header_liuwentao
```

```
{
    u_int16_t udp_source_port;                          /*源端口号*/
    u_int16_t udp_destination_port;                     /*目的端口号*/
    u_int16_t udp_length;                               /*长度*/
    u_int16_t udp_checksum;                             /*校验和*/
};
/*下面是TCP协议格式的定义*/
struct tcp_header
{
    u_int16_t tcp_source_port;                          /*源端口号*/
    u_int16_t tcp_destination_port;                     /*目的端口号*/
    u_int32_t tcp_acknowledgement;                      /*序列号*/
    u_int32_t tcp_ack;                                  /*确认码*/
    #ifdef WORDS_BIGENDIAN
    u_int8_t  tcp_offset:4,                             /*偏移*/
              tcp_reserved:4;                           /*保留*/
    #else
    u_int8_t  tcp_reserved:4,                           /*保留*/
              tcp_offset:4;                             /*偏移*/
    #endif
    u_int8_t  tcp_flags;                                /*标记*/
    u_int16_t tcp_windows;                              /*窗口大小*/
    u_int16_t tcp_checksum;                             /*校验和*/
    u_int16_t tcp_urgent_pointer;                       /*紧急指针*/
};
/*下面是ICMP协议格式的定义*/
struct icmp_header
{
    u_int8_t icmp_type;                                 /*ICMP类型*/
    u_int8_t icmp_code;                                 /*ICMP代码*/
    u_int16_t icmp_checksum;                            /*校验和*/
    u_int16_t icmp_id;                                  /*标识*/
    u_int16_t icmp_sequence;                            /*序列号*/
};
/*下面是分析TCP协议的函数定义*/
void tcp_protocol_packet_callback(u_char *argument,const struct pcap_pkthdr*
packet_header,const u_char* packet_content)
{
    struct tcp_header *tcp_protocol;                    /*定义TCP协议变量*/
    u_char flags;                                       /*标记*/
    int header_length;                                  /*首部长度*/
    u_short source_port;                                /*源端口号*/
    u_short destination_port;                           /*目的端口号*/
```

```
u_short  windows;                                          /*窗口*/
u_short urgent_pointer;                                    /*紧急指针*/
u_int sequence;                                            /*序列号*/
u_int acknowledgement;                                     /*确认号*/
u_int16_t checksum;                                        /*校验和*/
tcp_protocol = (struct tcp_header *) (packet_content+14+20);
    /*获得 TCP 协议数据内容，跳过以太网协议和 IP 协议部分*/
source_port = ntohs(tcp protocol->tcp_source_port);                /*获得源端口号*/
destination_port = ntohs(tcp_protocol->tcp_destination_port);      /*获得目的端口号*/
header_length = tcp_protocol->tcp_offset * 4;                      /*获得首部长度*/
sequence = ntohl(tcp_protocol->tcp_acknowledgement);               /*获得序列号*/
acknowledgement = ntohl(tcp_protocol->tcp_ack);                    /*获得确认号*/
windows = ntohs(tcp_protocol->tcp_windows);                        /*获得窗口大小*/
urgent_pointer = ntohs(tcp_protocol->tcp_urgent_pointer);          /*获得紧急指针*/
flags = tcp_protocol->tcp_flags;                                   /*获得标记*/
checksum =ntohs(tcp_protocol->tcp_checksum);                       /*获得校验和*/
printf("-------  TCP Protocol  (Transport Layer)  -------\n");
printf("Source Port:%d\n", source_port);
printf("Destination Port:%d\n", destination_port);
switch (destination_port)
    /*根据端口号判断应用层协议类型*/
    {
        case 80:printf("HTTP protocol\n");   break;
    /*上层协议为 HTTP 协议，可以在此调用分析 HTTP 协议的函数，读者可以自己尝试实现*/
    case 21:printf("FTP protocol\n");break;
    /*上层协议为 FTP 协议，可以在此调用分析 HTTP 协议的函数*/
    case 23:printf("TELNET protocol\n");break;
    /*上层协议为 TELNET 协议，可以在此调用分析 HTTP 协议的函数*/
    case 25:printf("SMTP protocol\n");break;
    /*上层协议为 SMTP 协议，可以在此调用分析 HTTP 协议的函数*/
    case 110:printf("POP3 protocol\n");break;
    /*上层协议为 POP3 协议，可以在此调用分析 HTTP 协议的函数*/
    default:break;
    /*其他端口号在这里没有分析，读者可以在此分析其他端口号代表的应用层协议*/
    }
    printf("Sequence Number:%u\n", sequence);
    printf("Acknowledgement Number:%u\n", acknowledgement);
    printf("Header Length:%d\n", header_length);
    printf("Reserved:%d\n", tcp_protocol->tcp_reserved);
    printf("Flags:");                          /*判断标记的种类*/
 if (flags & 0x08) printf("PSH ");
 if (flags & 0x10) printf("ACK ");
 if (flags & 0x02) printf("SYN ");
```

```
    if (flags & 0x20) printf("URG ");
    if (flags & 0x01) printf("FIN ");
    if (flags & 0x04) printf("RST ");
        printf("\n");
        printf("Window Size:%d\n", windows);
        printf("Checksum:%d\n", checksum);
        printf("Urgent pointer:%d\n", urgent_pointer);
}
/*下面是实现分析 UDP 协议的函数定义*/
void udp_protocol_packet_callback(u_char *argument,const struct pcap_pkthdr*
packet_header,const u_char* packet_content)
{
    struct udp_header_liuwentao *udp_protocol;          /*UDP 协议变量*/
    u_short source_port;                                /*源端口号*/
    u_short  destination_port;                          /*目的端口号*/
    u_short length;                                     /*长度*/
    udp_protocol = (struct udp_header_liuwentao *) (packet_content+14+20);
        /*获得 UDP 协议数据内容，跳过以太网协议和 IP 协议部分*/
    source_port = ntohs(udp_protocol->udp_source_port);              /*获得源端口号*/
    destination_port = ntohs(udp_protocol->udp_destination_port);    /*获得目的端口号*/
    length = ntohs(udp_protocol->udp_length);                        /*获得长度*/
    printf("---------- UDP Protocol (Transport Layer) ----------\n");
    printf("Source port:%d\n", source_port);
    printf("Destination port:%d\n", destination_port);
    switch (destination_port)
        /*根据端口号来判断应用层协议类型*/
        {
            case 138: printf("NETBIOS Datagram Service\n");  break;
        /*端口号是 138，表示上层协议为 NetBIOS 数据报服务，在此可以调用分析此协议的函数，读者自己可以
        试着实现。*/
        case 137: printf("NETBIOS Name Service\n");break;
        /*端口号是 137，表示上层协议为 NetBIOS 名字服务，在此可以调用分析此协议的函数*/
        case 139: printf("NETBIOS session service\n");break;
        /*端口号是 139，表示上层协议为 NetBIOS 会话服务，在此可以调用分析此协议的函数。*/
        case 53:  printf("name-domain server \n");break;
        /*端口号是 53，表示上层协议为域名服务，在此可以调用分析此协议的函数。*/
        default:break;
            /*其他的端口号在此没有分析，读者可以在此进一步分析*/
        }
    printf("Length:%d\n", length);
    printf("Checksum:%d\n", ntohs(udp_protocol->udp_checksum));
        /*获得校验和*/
}
```

```
/*下面是实现分析 ICMP 协议的函数的定义*/
void icmp_protocol_packet_callback(u_char *argument,const struct pcap_pkthdr*
packet_header,const u_char* packet_content)
{
    struct icmp_header *icmp_protocol;                    /*ICMP 协议变量*/
    icmp_protocol = (struct icmp_header *) (packet_content+14+20);
        /*获得 ICMP 协议数据内容，跳过以太网协议和 IP 协议部分*/
    printf("---------- ICMP Protocol (Transport Layer) ----------\n");
    printf("ICMP Type:%d\n", icmp_protocol->icmp_type);   /*获得 ICMP 类型*/
    switch (icmp_protocol->icmp_type)                     /*根据类型判断 ICMP 数据包的种类*/
    {
        case 8:
            /*类型是 8，表示是回显请求 ICMP 数据包*/
        printf("ICMP Echo Request Protocol \n");
        printf("ICMP Code:%d\n", icmp_protocol->icmp_code);      /*获得 ICMP 代码*/
        printf("Identifier:%d\n", icmp_protocol->icmp_id);       /*获得标识符*/
        printf("Sequence Number:%d\n", icmp_protocol->icmp_sequence);
            /*获得序列号*/
        break;
        case 0:
            /*类型为 0，表示是回显应答 ICMP 数据包*/
        printf("ICMP Echo Reply Protocol \n");
        printf("ICMP Code:%d\n", icmp_protocol->icmp_code);              /*获得 ICMP 代码*/
        printf("Identifier:%d\n", icmp_protocol->icmp_id);               /*获得标识符*/
        printf("Sequence Number:%d\n", icmp_protocol->icmp_sequence);    /*获得序列号*/
            break;
        default :break;
        /*其他类型的 ICMP 数据包在此没有分析，读者可以在此添加其他类型的 ICMP 数据包的分析*/
    }
    printf("ICMP Checksum:%d\n", ntohs(icmp_protocol->icmp_checksum));
        /*获得校验和*/
}
/*下面是实现 ARP 协议分析的函数定义*/
void arp_protocol_packet_callback(u_char *argument,const struct pcap_pkthdr*
packet_header,const u_char* packet_content)
{
    struct arp_header *arp_protocol;                  /*ARP 协议变量*/
    u_short protocol_type;                            /*协议类型*/
    u_short hardware_type;                            /*硬件类型*/
    u_short operation_code;                           /*操作码*/
    u_char *mac_string;                               /*以太网地址*/
    struct in_addr source_ip_address;                 /*源 IP 地址*/
    struct in_addr destination_ip_address;            /*目的 IP 地址*/
```

```
u_char hardware_length;                          /*硬件地址长度*/
u_char protocol_length;                          /*协议地址长度*/
printf("--------  ARP Protocol (Network Layer)   --------\n");
arp_protocol = (struct arp_header *) (packet_content+14);
    /*获得 ARP 协议数据内容，跳过以太网协议部分*/
hardware_type = ntohs(arp_protocol->arp_hardware_type);      /*获得硬件类型*/
protocol_type = ntohs(arp_protocol->arp_protocol_type);      /*获得协议类型*/
operation_code = ntohs(arp_protocol->arp_operation_code);    /*获得操作码*/
hardware_length=arp_protocol->arp_hardware_length;           /*获得硬件地址长度*/
protocol_length=arp_protocol->arp_protocol_length;           /*获得协议地址长度*/
printf("ARP Hardware Type:%d\n", hardware_type);
printf("ARP Protocol Type:%d\n", protocol_type);
printf("ARP Hardware Length:%d\n",hardware_length);
printf("ARP Protocol Length:%d\n",protocol_length);
printf("ARP Operation:%d\n", operation_code);
switch (operation_code)                                  /*根据操作码判断 ARP 协议类型*/
{
    case 1: printf("ARP Request Protocol\n");break;
        /*操作码为 1，表示是 ARP 请求协议*/
    case 2: printf("ARP Reply Protocol\n");break;
        /*操作码为 2，表示是 ARP 应答协议*/
    case 3: printf("RARP Request Protocol\n");break;
        /*操作码为 3，表示是 RARP 请求协议*/
    case 4: printf("RARP Reply Protocol\n");break;
        /*操作码为 4，表示是 RARP 应答协议*/
    default:break;
}
printf("Ethernet Source Address is : \n");
mac_string=arp_protocol->arp_source_ethernet_address;
printf("%02x:%02x:%02x:%02x:%02x:%02x\n",*mac_string,*(mac_string+1),
            *(mac_string+2),*(mac_string+3),
            *(mac_string+4),*(mac_string+5));      /*获得源以太网地址*/
memcpy((void *) &source_ip_address, (void *) &arp_protocol-> arp_source_ip_address,
sizeof (struct in_addr));
printf("Source IP Address:%s\n", inet_ntoa(source_ip_address));   /*获得源 IP 地址*/
printf("Ethernet Destination Address is : \n");
mac_string=arp_protocol->arp_destination_ethernet_address;
printf("%02x:%02x:%02x:%02x:%02x:%02x\n",*mac_string,*(mac_string+1),
            *(mac_string+2),*(mac_string+3),
            *(mac_string+4),*(mac_string+5));                /*获得目的以太网地址*/
memcpy((void *) &destination_ip_address, (void *) &arp_protocol-> arp_destination_
ip_address, sizeof (struct in_addr));
printf("Destination IP Address:%s\n", inet_ntoa(destination_ip_address));
```

```
        /*获得目的 IP 地址*/
}
/*下面是实现分析 IP 协议的函数定义*/
void ip_protocol_packet_callback(u_char *argument,const struct pcap_pkthdr*
packet_header,const u_char* packet_content)
{
    struct ip_header *ip_protocol;                          /*IP 协议变量*/
    u_int header_length;                                    /*首部长度*/
    u_int offset;                                           /*偏移*/
    u_char tos;                                             /*服务质量*/
    u_int16_t checksum;                                     /*校验和*/
    ip_protocol = (struct ip_header *) (packet_content+14);
        /*获得 IP 协议数据内容，跳过以太网协议部分*/
    checksum = ntohs(ip_protocol->ip_checksum);             /*获得校验和*/
    header_length = ip_protocol->ip_header_length*4;        /*获得 IP 首部长度*/
    tos = ip_protocol->ip_tos;                              /*获得服务质量*/
    offset = ntohs(ip_protocol->ip_off);                    /*活动偏移*/
    printf("----------- IP Protocol  (Network Layer)  -----------\n");
    printf("IP Version:%d\n", ip_protocol->ip_version);      /*获得版本号*/
        printf("Header length:%d\n", header_length);
        printf("TOS:%d\n", tos);
        printf("Total length:%d\n", ntohs(ip_protocol->ip_length));  /*获得总长度*/
        printf("Identification:%d\n", ntohs(ip_protocol >ip_id));    /*获得标识*/
        printf("Offset:%d\n", (offset & 0x1fff) * 8);
        printf("TTL:%d\n", ip_protocol->ip_tt1);                     /*获得生存时间*/
        printf("Protocol:%d\n", ip_protocol->ip_protocol);           /*获得协议类型*/
    switch (ip_protocol->ip_protocol)
        /*根据协议类型判断上层协议类型*/
        {
            case 6:  printf("The Transport Layer Protocol is TCP\n");break;
            /*上层协议为 TCP 协议*/
            case 17: printf("The Transport Layer Protocol is UDP\n");break;
            /*上层协议为 UDP 协议*/
            case 1:  printf("The Transport Layer Protocol is ICMP\n");break;
            /*上层协议为 ICMP 协议*/
            default:break;
        }
    printf("Header checksum:%d\n", checksum);                /*校验和*/
    printf("Source address:%s\n", inet_ntoa(ip_protocol->ip_souce_address));
        /*获得源 IP 地址*/
    printf("Destination address:%s\n", inet_ntoa(ip_protocol->ip_destination_address));
        /*获得目的 IP 地址*/
    switch (ip_protocol->ip_protocol)
```

```
        /*判断上层协议类型，然后调用相应的函数进行分析*/
        {
            case 6:
            /*上层协议为 TCP 协议*/
        tcp_protocol_packet_callback(argument,
                                     packet_header,
                                     packet_content);
        break;
        /*调用分析 TCP 协议的函数，注意参数的传递，表示分析的是同一个网络数据包*/
        case 17:
        /*上层协议为 UDP 协议*/
        udp_protocol_packet_callback(argument,
                                     packet_header,
                                     packet_content);
        break;
            /*调用分析 UDP 协议的函数，注意参数的传递方式*/
            case 1:
        /*上层协议为 ICMP 协议*/
        icmp_protocol_packet_callback(argument,
                                      packet_header,
                                      packet_content);
        break;
        /*调用分析 ICMP 协议的函数，注意参数的传递*/
            default:break;
        /*其他类型的协议在此没有分析，读者可以在此进一步分析其他传输层协议*/
        }
}
/*下面是分析以太网协议的函数定义，同时它也是回调函数*/
void ethernet_protocol_packet_callback(u_char *argument,const struct pcap_pkthdr*
packet_header,const u_char* packet_content)
{
    u_short ethernet_type;                          /*以太网类型*/
    struct ether_header *ethernet_protocol;         /*以太网协议变量*/
    u_char *mac_string;                             /*以太网地址*/
    static int packet_number = 1;
    printf("***************************************************\n");
    printf("The %d  packet is captured.\n",packet_number);
    printf("--------  Ethernet Protocol (Link Layer)   --------\n");
        ethernet_protocol = (struct ether_header *) packet_content;
        /*获得以太网协议数据内容*/
    printf("Ethernet type is :\n");
    ethernet_type= ntohs(ethernet_protocol->ether_type);          /*获得以太网类型*/
    printf("%04x\n",ethernet_type);
```

```
    switch(ethernet_type)
        /*根据以太网类型字段判断上层协议类型*/
    {
        case 0x0800:  printf("The network layer is IP protocol\n");break;
            /*上层协议为 IP 协议*/
        case 0x0806:  printf("The network layer is ARP protocol\n"); break;
            /*上层协议为 ARP 协议*/
        case 0x8035:  printf("The network layer is RARP protocol\n");break;
            /*上层协议为 RARP 协议*/
        default :break;
    }
    printf("Mac Source Address is : \n");
    mac_string=ethernet_protocol->ether_shost;
    printf("%02x:%02x:%02x:%02x:%02x:%02x\n",*mac_string,*(mac_string+1),
                *(mac_string+2),*(mac_string+3),
                *(mac_string+4),*(mac_string+5));        /*获得源以太网地址*/
    printf("Mac Destination Address is : \n");
    mac_string=ethernet_protocol->ether_dhost;
    printf("%02x:%02x:%02x:%02x:%02x:%02x\n",*mac_string,*(mac_string+1),
                *(mac_string+2),*(mac_string+3),
                *(mac_string+4),*(mac_string+5));        /*获得目的以太网地址*/
    switch(ethernet_type)
        /*根据以太网类型字段判断上层协议，然后调用相应函数进行分析*/
    {
        case 0x0806:                    /*上层是 ARP 协议*/
        arp_protocol_packet_callback(argument,
                                        packet_header,
                                        packet_content);
        break;                          /*调用分析 ARP 协议的函数*/
        case 0x0800:                    /*上层是 IP 协议*/
        ip_protocol_packet_callback(argument,
                                        packet_header,
                                        packet_content);
        break;                          /*调用分析 IP 协议的函数*/
        default :break;
            /*其他协议在此没有分析，读者可以试着进一步分析*/
    }
    printf("*************************************************\n");
    packet_number++;
}
/*下面是主函数的定义*/
void  main()
{
```

```
    pcap_t* pcap_handle;                        /*Libpcap 句柄*/
    char error_content[PCAP_ERRBUF_SIZE];       /*存储错误内容*/
    char *net_interface;                        /*网络接口*/
    struct bpf_program bpf_filter;              /*BPF 过滤规则*/
    char bpf_filter_string[] = "";
    /*过滤规则字符串，此时为空的，表示捕获所有的网络数据包，而不是捕获特定的网络数据包*/
    bpf_u_int32 net_mask;                       /*网络掩码*/
    bpf_u_int32 net_ip;                         /*网络地址*/
    net_interface = pcap_lookupdev(error_content);      /*获得网络接口*/
    pcap_lookupnet(net_interface,
                   &net_ip,
                   &net_mask,
                   error_content);              /*获得网络地址和网络掩码*/
    pcap_handle = pcap_open_live(net_interface,
                                 BUFSIZ,
                                 1,
                                 0,
                                 error_content);        /*打开网络接口*/
    pcap_compile(pcap_handle,
                 &bpf_filter,
                 bpf_filter_string,
                 0,
                 net_ip);                           /*编译过滤规则*/
    pcap_setfilter(pcap_handle,
                   &bpf_filter);                    /*设置过滤规则*/
    if(pcap_datalink(pcap_handle)!=DLT_EN10MB)
        return;
    pcap_loop(pcap_handle,
              -1,
              ethernet_protocol_packet_callback,
              NULL);
    /*注册回调函数，循环捕获网络数据包，然后调用回调函数对捕获的网络数据包进行分析*/
    pcap_close(pcap_handle);                        /*关闭 Libpcap 操作*/
}
```

此程序的功能是捕获所有的网络数据包，然后分析此数据包是一个什么协议的数据包。其分析了常用的几个协议，包括以太网协议、ARP 协议、IP 协议、UDP 协议、TCP 协议和 ICMP 协议。而且分析了每个数据包的不同协议部分，并详细显示出来。

编译此程序，编译命令如下：

```
gcc -o getpacket get_packet_code.c -lpcap
```

运行此程序，其输出内容摘录如下：

```
*************************************************
The 1  packet is captured.
```

```
--------  Ethernet Protocol (Link Layer)   --------
Ethernet type is :
0800
The network layer is IP protocol
Mac Source Address is :
00:0a:e6:5d:2b:3d
Mac Destination Address is :
00:00:e8:e7:b9:46
----------- IP Protocol  (Network Layer)  -----------
IP Version:4
Header length:20
TOS:16
Total length:55
Identification:25671
Offset:0
TTL:64
Protocol:6
The Transport Layer Protocol is TCP
Header checksum:21776
Source address:192.168.0.5
Destination address:192.168.0.4
-------  TCP Protocol  (Transport Layer)  -------
Source Port:34841
Destination Port:135
Sequence Number:2384117110
Acknowledgement Number:1397715131
Header Length:32
Reserved:0
Flags:PSH ACK
Window Size:5840
Checksum:1926
Urgent pointer:0
************************************************
************************************************
The 2  packet is captured.
--------  Ethernet Protocol (Link Layer)   --------
Ethernet type is :
0806
The network layer is ARP protocol
Mac Source Address is :
00:00:e8:e7:b9:46
Mac Destination Address is :
ff:ff:ff:ff:ff:ff
```

```
--------  ARP Protocol (Network Layer)   --------
ARP Hardware Type:1
ARP Protocol Type:2048
ARP Hardware Length:6
ARP Protocol Length:4
ARP Operation:1
ARP Request Protocol
Ethernet Source Address is :
00:00:e8:e7:b9:46
Source IP Address:192.168.0.4
Ethernet Destination Address is :
00:00:00:00:00:00
Destination IP Address:192.168.0.5
**************************************************
**************************************************
The 3  packet is captured.
--------  Ethernet Protocol (Link Layer)   --------
Ethernet type is :
0806
The network layer is ARP protocol
Mac Source Address is :
00:0a:e6:5d:2b:3d
Mac Destination Address is :
00:00:e8:e7:b9:46
--------  ARP Protocol (Network Layer)   --------
ARP Hardware Type:1
ARP Protocol Type:2048
ARP Hardware Length:6
ARP Protocol Length:4
ARP Operation:2
ARP Reply Protocol
Ethernet Source Address is :
00:0a:e6:5d:2b:3d
Source IP Address:192.168.0.5
Ethernet Destination Address is :
00:00:e8:e7:b9:46
Destination IP Address:192.168.0.4
**************************************************
**************************************************
The 4  packet is captured.
--------  Ethernet Protocol (Link Layer)   --------
Ethernet type is :
0800
```

```
The network layer is IP protocol
Mac Source Address is :
00:00:e8:e7:b9:46
Mac Destination Address is :
00:0a:e6:5d:2b:3d
----------- IP Protocol  (Network Layer)  -----------
IP Version:4
Header length:20
TOS:0
Total length:52
Identification:5241
Offset:0
TTL:128
Protocol:6
The Transport Layer Protocol is TCP
Header checksum:25841
Source address:192.168.0.4
Destination address:192.168.0.5
-------  TCP Protocol  (Transport Layer)  -------
Source Port:135
Destination Port:34841
Sequence Number:1397715131
Acknowledgement Number:2384117113
Header Length:32
Reserved:0
Flags:ACK
Window Size:17517
Checksum:40074
Urgent pointer:0
************************************************
************************************************
The 5  packet is captured.
--------   Ethernet Protocol (Link Layer)    --------
Ethernet type is :
0800
The network layer is IP protocol
Mac Source Address is :
00:0a:e6:5d:2b:3d
Mac Destination Address is :
00:00:e8:e7:b9:46
----------- IP Protocol  (Network Layer)  -----------
IP Version:4
Header length:20
```

```
TOS:0
Total length:84
Identification:0
Offset:0
TTL:64
Protocol:1
The Transport Layer Protocol is ICMP
Header checksum:47439
Source address:192.168.0.5
Destination address:192.168.0.4
---------- ICMP Protocol (Transport Layer) ----------
ICMP Type:8
ICMP Echo Request Protocol
ICMP Code:0
Identifier:5119
Sequence Number:256
ICMP Checksum:27393
************************************************
************************************************
The 6 packet is captured.
-------- Ethernet Protocol (Link Layer) --------
Ethernet type is :
0800
The network layer is IP protocol
Mac Source Address is :
00:00:e8:e7:b9:46
Mac Destination Address is :
00:0a:e6:5d:2b:3d
- ------ - IP Protocol (Network Layer) -----------
IP Version:4
Header length:20
TOS:0
Total length:84
Identification:5242
Offset:0
TTL:128
Protocol:1
The Transport Layer Protocol is ICMP
Header checksum:25813
Source address:192.168.0.4
Destination address:192.168.0.5
---------- ICMP Protocol (Transport Layer) ----------
ICMP Type:0
```

```
ICMP Echo Reply Protocol
ICMP Code:0
Identifier:5119
Sequence Number:256
ICMP Checksum:29441
**************************************************
**************************************************
The 7  packet is captured.
--------  Ethernet Protocol (Link Layer)   --------
Ethernet type is :
0806
The network layer is ARP protocol
Mac Source Address is :
00:0a:e6:5d:2b:3d
Mac Destination Address is :
00:00:e8:e7:b9:46
--------  ARP Protocol (Network Layer)   --------
ARP Hardware Type:1
ARP Protocol Type:2048
ARP Hardware Length:6
ARP Protocol Length:4
ARP Operation:1
ARP Request Protocol
Ethernet Source Address is :
00:0a:e6:5d:2b:3d
Source IP Address:192.168.0.5
Ethernet Destination Address is :
00:00:00:00:00:00
Destination IP Address:192.168.0.4
**************************************************
**************************************************
The 8  packet is captured.
--------  Ethernet Protocol (Link Layer)   --------
Ethernet type is :
0806
The network layer is ARP protocol
Mac Source Address is :
00:00:e8:e7:b9:46
Mac Destination Address is :
00:0a:e6:5d:2b:3d
--------  ARP Protocol (Network Layer)   --------
ARP Hardware Type:1
ARP Protocol Type:2048
```

```
ARP Hardware Length:6
ARP Protocol Length:4
ARP Operation:2
ARP Reply Protocol
Ethernet Source Address is :
00:00:e8:e7:b9:46
Source IP Address:192.168.0.4
Ethernet Destination Address is :
00:0a:e6:5d:2b:3d
Destination IP Address:192.168.0.5
*************************************************
```

程序的运行结果如图 3-19 所示。

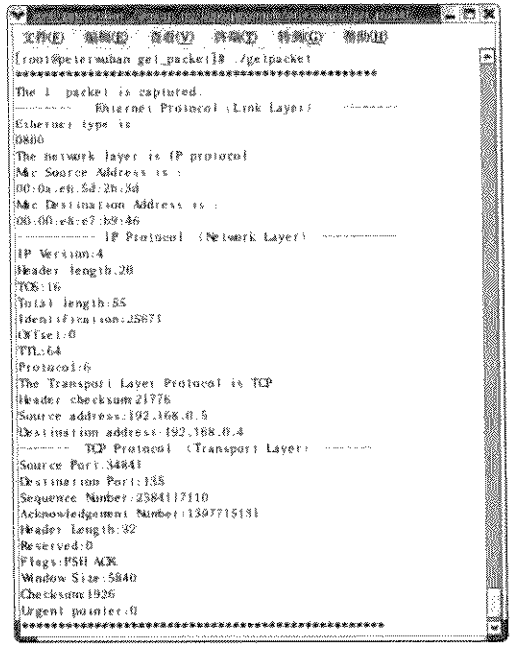

图 3-19　捕获所有的网络数据包

从程序的运行结果来分析，此程序捕获所有的网络数据包，从链路层、网络层、传输层到应用层对网络数据包进行协议分析，详细分析了以太网协议、ARP 协议、IP 协议、TCP 协议、UDP 协议和 ICMP 协议。

此程序是一个比较综合的例子，它比较典型地反应了使用 Libpcap 捕获网络数据包以及对网络数据包进行分析的过程。虽然程序比较长，但结构清晰，它是基于 Libpcap 开发包进行程序开发的一个典型模式。很多现在比较著名的网络协议分析工具都是按照这种模式来实

现的，理解了这个程序，对于理解很多基于 Libpcap 的程序是非常具有借鉴意义的。

3.6 小结

本章对网络数据包捕获开发包 Libpcap 进行了详细的分析、讨论，包括对 Libpcap 开发包的概述、对 Libpcap 的数据结构以及对 Libpcap 所有输出函数的详细阐述（包括函数返回值、函数参数的定义以及函数的功能）。最后，用一些典型的例子来说明使用开发包 Libpcap 的方法。在这些例子中，都使用了结构化程序设计，思路清楚，程序简明，容易理解。对每个例子的核心语句进行了比较详细的注解，分析了程序的设计思路，对程序的运行结果也进行了详细讨论。

Libpcap 开发包的应用非常广泛，迄今为止，很多优秀的网络安全软件都是基于 Libpcap 来开发的，如 Linux 下著名的网络数据包捕获和分析工具 Tcpdump，它可以对网络数据包进行捕获和协议分析；还有著名的入侵检测系统 Snort，也是基于 Libpcap 开发包来进行开发的。当然，还有很多其他优秀的基于 Libpcap 的网络安全软件，只要涉及到网络数据包的捕获，就可以使用 Libpcap 开发包。由于 Libpcap 开发包使用起来非常简单，并且其效率也不错，所以很多软件都用它来实现数据包的捕获功能。

后面讨论的 WinPcap 开发包，是 Libpcap 在 Windows 平台下的实现，并进行了其他方面的改进。WinPcap 专门针对 Windows 平台，但是它的某些使用方法跟 Libpcap 完全一样。后面还要讨论的数据包构造和发送开发包 Libnet，也用到了 Libpcap 开发包；入侵检测开发包 Libnids，也是在 Libpcap 基础上开发的，并用到了 Libnet 开发包。

第 4 章　Windows 网络数据包捕获开发包 WinPcap

4.1　WinPcap 介绍

1. WinPcap 和 Libpcap 的关系

WinPcap（Windows Packet Capture）是 Windows 平台下一个专业网络数据包捕获开发包，是为 Libpcap 在 Windows 平台下实现数据包的捕获而设计的。在设计 WinPcap 时参照了 Libpcap，使用方法与 Libpcap 相似。使用 WinPcap 开发包，可以把在 Linux 下基于 Libpcap 的程序很容易地移植到 Windows 平台下。WinPcap 是在 BSD 许可证下发布的，它主要由加利福尼亚大学的 Lawrence Berkeley Laboratory 开发。WinPcap 的下载网址为：http://winpcap.polito.it/。

使用 WinPcap 开发包的软件主要有：

- Windump：是 Windows 下一个著名的网络协议分析软件，与 Linux 下的 Tcpdump 的功能几乎一致；
- Analyzer：一个 Windows 下的嗅探器；
- Ethereal：一个免费的网络协议分析软件；
- Dsniff：也是一个网络嗅探器。

除了上述软件之外，还有很多软件都使用了 WinPcap 开发包。WinPcap 开发包的使用非常广泛，几乎所有在 Windows 平台下要使用数据包捕获功能的软件都可以使用 WinPcap 开发包。

WinPcap 的最新稳定版本为 3.0，测试版本为 3.0 beta 4。本书以 3.0 版本的 WinPcap 开发包为讨论对象。

2. WinPcap 的功能和好处

开发 WinPcap 这个项目的目的在于为 win32 应用程序提供访问网络底层的能力，其核心功能是捕获网络数据包，其他功能包括数据包过滤、数据包发送、流量统计和数据包存储等，详见 2.2.2 节。

使用 WinPcap 有很多好处，主要体现在以下几个方面。

- WinPcap 给程序员提供了一套标准的网络数据包捕获的编程接口，并且与 Libpcap 兼容，使得原来许多 Linux 平台下的网络安全程序可很快地移植到 Windows 平台下。
- 使用 WinPcap 可以提供很高的应用效率。WinPcap 充分考虑了各种性能和效率的优化，在内核层实现了数据包的捕获和过滤。这是由 NPF 来实现的，NPF 是 WinPcap 的核心部分，它实现了内核层次的统计功能，对于设计网络流量的程序很有好处。
- WinPcap 还提供了发送数据包的能力。

3. WinPcap 的安装

从 WinPcap 网站上可以下载三个文件。

第 1 个文件是 WinPcap_3_0.exe，它是一个自动安装文件，其包括重要的动态链接库（DLL）和驱动程序，双击此文件就可以按照提示进行安装了，其安装界面如图 4-1 所示。

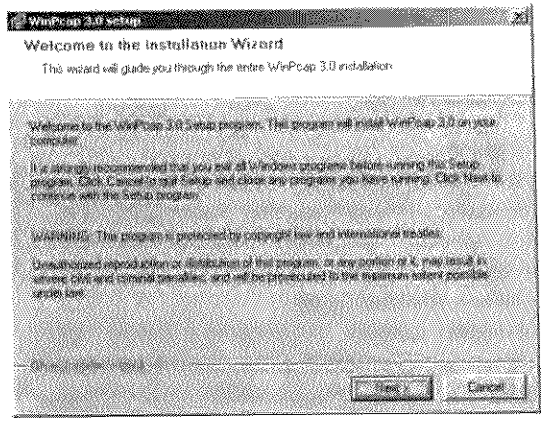

图 4-1　WinPcap 的安装界面

这个安装文件必须安装，它是所有基于 WinPcap 的应用程序运行所必需的。如果不安装此文件，就不能运行基于 WinPcap 的应用程序。

第 2 个文件是 wpdpack_3_0.zip，它是 WinPcap 开发包文件，是设计基于 WinPcap 的应用程序的开发文件。该文件包括开发用的头文件和静态链接库文件，是开发者要用到的。如果不需要开发基于 WinPcap 的程序，而只是运行基于 WinPcap 的软件，就不需要此文件。

此文件只需解压就可以了。解压后，其中有几个子文件夹，如图 4-2 所示。

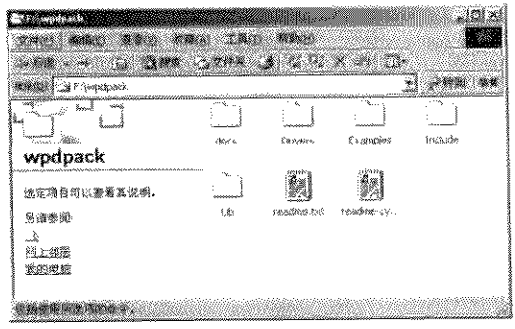

图 4-2　WinPcap 开发包内容

文件夹 docs 中的文件是参考文档；文件夹 Examples 中的是一些很好的参考例子；文件夹 Include 中的是 WinPcap 的一些头文件，编程时必须把此文件夹路径包含到 Visual C++ 的包含头文件的路径中；文件夹 Lib 中是开发程序时用到的静态链接库，编程时此文件夹也必须包含到 Visual C++ 库文件路径中。

第 3 个文件是 wpcapsrc_3_0.zip。它是 WinPcap 的源程序文件，是一个开放源代码软件，提供了完整的源代码。如果想更深入地了解 WinPcap 的运行机理，可以查看其源代码，对使用 WinPcap 也会有帮助。

在使用 WinPcap 进行程序开发时，只需第 1 个文件和第 2 个文件就可以了。

4.2 WinPcap 的组成

WinPcap 包括三部分内容。

第一部分是内核层的数据包过滤模块。它相当于在 Linux 下 Libpcap 使用的 BPF 过滤模块，实现了高效的网络数据包的捕获和过滤功能，其过滤规则跟 BPF 过滤规则是一样的。此过滤模块实际是一个驱动程序，被称为 NPF（Netgroup Packet Filter）数据包驱动程序。

第二部分是动态链接库 packet.dll。它是提供给开发者的一个接口，使用它就可以调用 WinPcap 的函数，它是一个较低层的开发接口。

第三部分是动态链接库 wpcap.dll。它也是提供给开发者的一个接口，但它是一个更高层的编程接口，其调用与系统无关。它是基于 Libpcap 而设计的，所以基于 Libpcap 的程序可以使用 wpcap.dll 来移植到 Windows 平台下。或者说，可以在 Windows 平台下设计类似于 Libpcap 的程序，其实现过程跟 Libpcap 的使用非常相似。使用此接口进行编程，其函数的调用与 Libpcap 的调用几乎完全一样，函数名称和参数的定义也一样。几乎可以把在 Linux 下使用 Libpcap 写的程序原封不动地搬到 Windows 平台下。可以说，WinPcap 就是 Windows 下的 Libpcap 开发包，其使用方法完全跟 Libpcap 一样。在 WinPcap 中虽然屏蔽了操作系统的不同之处，但它提供给用户的接口跟 Libpcap 的接口完全一样。

1. 网络驱动程序接口规范（NDIS）

网络驱动程序接口规范（NDIS，Network Driver Interface Specification）是由微软公司和 3Com 公司共同制定的网络接口驱动规范。它在 Windows 操作系统中的位置如图 4-3 所示。

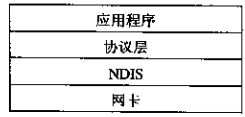

图 4-3 NDIS 的位置

从图 4-3 中可以看到，NDIS 位于网卡和协议层之间。NDIS 提供了一个接口，它为上层的协议提供服务，并且屏蔽了下层各种网卡的差别。NDIS 支持多种网络协议，如 TCP/IP，IPX/SPX，NetBEUI，AppleTalk 等，并且支持不同厂家生产的各种网卡。NDIS 还支持多种工作模式，支持多处理器。

NDIS 为网络驱动的开发提供了一套标准的接口，使得网络驱动程序的跨平台性更好。NDIS 支持以下网络驱动类型：

（1）小端口驱动，也称网卡驱动。

（2）中间层驱动：介于协议驱动和小端口驱动之间，它能够截获所有的网络数据包，如以太网帧。NDIS 中间层驱动的应用很广泛，可以用于个人防火墙，也可以用来实现 VPN，NAT，PPP Over Ethernet 以及 VLan 等。

（3）协议驱动，如 TCP/IP 协议驱动。WinPcap 的 NPF 就属于协议驱动类型。

2. NPF

NPF（Netgroup Packet Filter）是 WinPcap 的核心部分。它的主要功能就是捕获数据包，还可以发送数据包、存储数据包以及对网络进行统计分析。图 4-4 所示是 NPF 的结构。

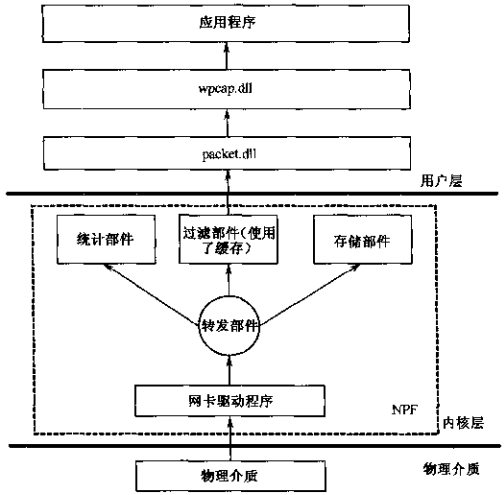

图 4-4　NPF 的结构

图 4-4 中的箭头表示网络数据包的流动方向。从图 4-4 可以看到，NPF 工作在内核层，与前面介绍的 BPF（BSD Packet Filter）非常相似，其实它是根据 BPF 来设计的。NPF 有一个网络转发部件，从网卡驱动程序收集网络数据包，既可以发送给过滤部件，对网络数据包进行过滤，也可以发送给统计部件，对网络进行统计分析，还可以发送给存储部件，把网络数据包直接存储到磁盘。数据包在 NPF 中使用了缓存机制，主要是为了提高效率和速度。

3. packet.dll

从图 4-4 可以看到，packet.dll 位于 wpcap.dll 的下层，是较低层的编程接口。使用它可以跟内核打交道，它是开发者与内核交互的一个很好的接口。packet.dll 提供了很多功能复杂的编程接口，其接口形式与微软的 DDK 提供的 packet32.dll 差不多。使用 packet.dll 比较烦琐，如果无特殊要求，一般不用 packet.dll 进行编程开发。用得最多的还是 wpcap.dll，因为它提供了一个更高级、更方便的编程接口。

4. wpcap.dll

从图 4-4 可以看到，wpcap.dll 是最上层的一个接口，与 Libpcap 兼容，所以只要掌握了 Libpcap 的使用方法，那么对 wpcap.dll 的使用就很容易了。

wpcap.dll 是用得最多的一种编程接口，由于它与 Libpcap 兼容，所以在 Linux 下使用的 Libpcap 程序非常容易移植到 Windows 平台下。使用 wpcap.dll 进行数据包捕获的编程，与

使用 Libpcap 一样简单。

本章将着重讨论 wpcap.dll 的使用方法，对 WinPcap 其他部分的功能，读者如有兴趣，可以参考有关资料。

4.3 WinPcap 数据结构

由于 WinPcap 的设计是基于 Libpcap 的，所以它使用了与 Libpcap 相同的数据结构，读者可以查看前面 3.3 节对它们的详细介绍。在此只列出所有的 WinPcap 核心数据结构，并进行简单介绍。

1. pcap_addr

pcap_addr 数据结构的定义如下：

```
typedef struct pcap_addr pcap_addr_t;
struct pcap_addr {
        struct pcap_addr *next;          /*指向下一个地址结点*/
        struct sockaddr *addr;           /*网络接口地址*/
        struct sockaddr *netmask;        /*地址掩码*/
        struct sockaddr *broadaddr;      /*广播地址*/
        struct sockaddr *dstaddr;        /*目的地址*/
};
```

此数据结构描述的是网络接口的地址。

2. pcap_file_header

pcap_file_header 数据结构的定义如下：

```
struct pcap_file_header {
        bpf_u_int32 magic;          /*文件类型*/
        u_short version_major;      /*主版本号*/
        u_short version_minor;      /*次版本号*/
        bpf_int32 thiszone;         /*区域时间*/
        bpf_u_int32 sigfigs;        /*时间戳*/
        bpf_u_int32 snaplen;        /*捕获长度*/
        bpf_u_int32 linktype;       /*链路层类型*/
};
```

此数据结构用来描述一个 WinPcap 存储文件类型。不同的 WinPcap 存储形式使用不同的存储文件类型，一个存储文件类型用惟一值来表示。

3. pcap_if

pcap_if 数据结构定义如下：

```
typedef struct pcap_if pcap_if_t;
struct pcap_if {
        struct pcap_if *next;            /*下一个网络接口结点*/
        char *name;                      /* 网络接口名字*/
        char *description;               /* 描述信息*/
```

```
        struct pcap_addr *addresses;        /*网络接口地址*/
        bpf_u_int32 flags;                  /*标记*/
};
```

此数据结构描述的是一个网络接口，它其实是网络接口链表中的一个结点。一个结点就描述了一个网络接口，当网络接口比较多时，就把它们组成一个链表，用此链表来存储所有的网络接口。

4. pcap_pkthdr

pcap_pkthdr 数据结构的定义如下：

```
struct pcap_pkthdr {
        struct timeval ts;                  /*时间戳*/
        bpf_u_int32 caplen;                 /*捕获长度*/
        bpf_u_int32 len;                    /*数据包长度*/
};
```

此数据结构用来描述每个捕获到的数据包的一些基本信息。每个数据包都有此数据结构。

5. pcap_stat

pcap_stat 数据结构的定义如下：

```
struct pcap_stat {
        u_int ps_recv;              /*捕获到的数据包个数*/
        u_int ps_drop;              /*丢失的数据包个数*/
        u_int ps_ifdrop;            /*未用*/
    #ifdef WIN32
        u_int bs_capt;              /*数据包到达应用层的个数*/
    #endif
};
```

此数据结构主要描述的是 WinPcap 的状态信息。

4.4 WinPcap 函数

WinPcap 提供的输出函数与 Libpcap 的函数是完全一样的，它们的使用方式也完全一样。在此只列出所有函数名称及功能介绍，其详细说明请查看前面对 Libpcap 函数的介绍。与 Windows 相关的输出函数单独列出，并进行详细介绍。

1. 网络接口函数

（1）int pcap_findalldevs(pcap_if_t **alldevsp, char *errbuf)

此函数的功能是查找机器的所有可用的网络接口，用一个网络接口链表返回。

（2）void pcap_freealldevs(pcap_if_t *alldevs)

此函数的功能是释放网络接口链表中的所有网络接口。

（3）char * pcap_lookupdev(char *errbuf)

此函数的功能是查询本机的网络接口名字。

（4）int pcap_lookupnet(register const char *device,register bpf_u_int32 *netp, register bpf_u_int32 *maskp,register char *errbuf)

此函数的功能是获取网络地址和网络掩码。

（5）pcap_t * pcap_open_live(const char *device, int snaplen, int promisc, int to_ms,char *ebuf)

此函数的功能是打开一个网络接口进行数据包捕获。打开的模式由“promisc”表示：如果是1，就表示以混杂模式打开此网络接口；否则，以非混杂模式打开网络接口。

2. 规则函数

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

此函数的功能是设置BPF过滤规则，由参数fp确定。

（2）int pcap_compile(pcap_t *p, struct bpf_program *program,
char *buf, int optimize, bpf_u_int32 mask)

此函数的功能是编译BPF过滤规则。

（3）int pcap_compile_nopcap(int snaplen_arg, int linktype_arg,struct bpf_program *program, char *buf, int optimize, bpf_u_int32 mask)

此函数的功能也是编译BPF过滤规则。

（4）void pcap_freecode(struct bpf_program *program)

此函数的功能是释放过滤规则的内存。

3. 数据包捕获函数

（1）int pcap_loop(pcap_t *p, int cnt, pcap_handler callback, u_char *user)

此函数是循环捕获网络数据包，直到遇到错误或满足退出条件。每捕获一个数据包就调用callback指示的回调函数。所以，可以在回调函数中对捕获到的数据包进行操作。

（2）int pcap_dispatch(pcap_t *p, int cnt, pcap_handler callback, u_char *user)

此函数的功能是捕获网络数据包。

（3）const u_char * pcap_next(pcap_t *p, struct pcap_pkthdr *h)

此函数的功能是捕获一个网络数据包，然后返回此数据包。

（4）int pcap_next_ex(pcap_t *p, struct pcap_pkthdr **pkt_header,const u_char **pkt_data)

此函数的功能是捕获一个网络数据包。

（5）void pcap_breakloop(pcap_t *p)

此函数的功能是退出循环捕获数据包状态。

（6）pcap_t * pcap_open_dead(int linktype, int snaplen)

此函数的功能是构造一个Libpcap句柄。

（7）pcap_t * pcap_open_offline(const char *fname, char *errbuf)

此函数的功能是打开一个文件，此文件的内容是网络数据包。

（8）void pcap_close(pcap_t *p)

此函数的功能是关闭Libpcap操作，并销毁相应资源。

4. 文件相关函数

（1）FILE * pcap_file(pcap_t *p)

此函数的功能是返回 Libpcap 的文件句柄。

（2）int pcap_fileno(pcap_t *p)

此函数的功能是返回 Libpcap 的文件描述符号。

（3）pcap_dumper_t * pcap_dump_open(pcap_t *p, const char *fname)

此函数的功能是打开一个文件，准备向其写网络数据包数据。

（4）int pcap_dump_flush(pcap_dumper_t *p)

此函数的功能是把数据包数据存入文件中。

（5）void pcap_dump_close(pcap_dumper_t *p)

此函数的功能关闭文件。

（6）void pcap_dump(u_char *user, const struct pcap_pkthdr *h, const u_char *sp)

此函数的功能是向文件中写网络数据包内容。

（7）FILE * pcap_dump_file(pcap_dumper_t *p)

此函数的功能是返回一个标准的文件句柄。

5. 错误处理函数

（1）void pcap_perror(pcap_t *p, char *prefix)

此函数的功能是获取最后一个错误信息。

（2）char * pcap_strerror(int errnum)

此函数的功能是获取对应的错误信息。

（3）char * pcap_geterr(pcap_t *p)

此函数的功能是获取最后一个错误信息。

6. 辅助函数

（1）int pcap_stats(pcap_t *p, struct pcap_stat *ps)

此函数的功能是获得统计信息。

（2）int pcap_datalink(pcap_t *p)

此函数的功能是获取链路层状态。例如，如果类型为 DLT_EN10MB，就表示以太网。

（3）int pcap_list_datalinks(pcap_t *p, int **dlt_buffer)

此函数的功能是返回支持的所有链路层类型的链表。

（4）int pcap_set_datalink(pcap_t *p, int dlt)

此函数的功能是设置链路层类型。

（5）int pcap_datalink_name_to_val(const char *name)

此函数的功能是把链路层类型对应的名字转换成标识。

（6）const char * pcap_datalink_val_to_name(int dlt)

此函数的功能是把链路层标识转换成对应的链路层名字字符串。

（7）const char * pcap_datalink_val_to_description(int dlt)

此函数的功能是把链路层类型的描述求出来。

（8）int pcap_snapshot(pcap_t *p)

此函数的功能是返回 Libpcap 句柄中的 snapshot 数值。

（9）int　pcap_is_swapped(pcap_t *p)

此函数的功能是返回文件类型是否正确的信息。

（10）int　pcap_major_version(pcap_t *p)

此函数的功能是返回文件的主版本号。

（11）int　pcap_minor_version(pcap_t *p)

此函数的功能是返回文件的次版本号。

（12）const char *　pcap_lib_version(void)

此函数的功能是返回 Libpcap 的版本信息。

（13）int　pcap_getnonblock(pcap_t *p, char *errbuf)

此函数的功能是返回 Libpcap 的非阻塞状态。

（14）int　pcap_setnonblock(pcap_t * p, int nonblock, char * errbuf);

此函数的功能是设置 Libpcap 是否为非阻塞模式。

7. 与 Windows 相关的函数

这部分函数在 Linux 下的 Libpcap 中无法使用，而主要在 Windows 平台下使用，所以它们是与 Windows 平台相关的函数。它们在 Libpcap 中也有定义，但由于在 Windows 平台下使用，所以在前面的 Libpcap 函数介绍中没有介绍，而把它们放在此处介绍。

（1）int pcap_setbuff(pcap_t *p, int dim);

函数返回值：函数操作成功就返回 0，失败就返回－1。

参数描述：参数 p 表示 WinPcap 句柄，参数 dim 表示字节数目。

此函数的功能是设置内核中与网卡相关的缓冲区的大小。

（2）int pcap_setmode(pcap_t *p, int mode);

函数返回值：函数操作成功就返回 0，失败就返回－1。

参数描述：参数 p 表示 WinPcap 句柄。参数 mode 表示工作模式，有两种工作模式：捕获模式和统计模式，分别用 MODE_CAPT 和 MODE_STAT 表示。如果是捕获模式，网络接口就只捕获网络数据包；如果是统计模式，网络接口可以进行网络信息统计。

此函数的功能是设置网络接口的工作模式。

（3）int pcap_sendpacket(pcap_t *p, u_cbar *buf, int size);

函数返回值：函数操作成功就返回 0，失败就返回－1。

参数描述：参数 p 表示 WinPcap 句柄，参数 buf 表示要发送的数据包缓存数据，参数 size 表示缓存 buf 的大小。

此函数的功能是发送一个原始数据包。

（4）int pcap_setmintocopy(pcap_t *p, int size);

函数返回值：函数操作成功就返回 0，失败就返回－1。

参数描述：参数 p 表示 WinPcap 句柄，参数 size 表示缓存大小。

此函数的功能是在内核设置一个最小的缓存大小。此缓存用来存放数据包内容，如果缓存装满，内核就把此数据内容复制给应用层；如果缓存还有空闲，内核就一直等到缓存装满后再把它们送到应用层，而不是一有数据到达就把它们送入应用层。

4.5 WinPcap 开发包的使用

4.5.1 使用 WinPcap 的一般流程

先举一个简单的例子：获取网络地址和掩码地址。该程序演示了怎样使用 WinPcap 来开发应用程序的过程，其核心源代码如下：

```
#include <pcap.h>
/*WinPcap 头文件*/
void  main()
{
    char error_content[PCAP_ERRBUF_SIZE];          /*存储错误信息*/
    struct in_addr net_ip_address;                 /*网络地址*/
    struct in_addr net_mask_address;               /*掩码地址*/
    char *net_interface;                           /*网络接口*/
    char *net_ip_string;                           /*网络地址*/
    char *net_mask_string;                         /*网络掩码*/
    u_int32_t net_ip;
    u_int32_t net_mask;
    net_interface = pcap_lookupdev(error_content);          /*获得可用的网络接口*/
    pcap_lookupnet(net_interface,&net_ip,&net_mask,error_content);
        /*获得网络接口的网络地址和网络掩码*/
    net_ip_address.s_addr = net_ip;
    net_ip_string = inet_ntoa(net_ip_address);              /*获得网络地址*/
    printf("网络地址:%s\n",net_ip_string);
    net_mask_address.s_addr = net_mask;
    net_mask_string = inet_ntoa(net_mask_address);          /*获得掩码地址*/
    printf("网络掩码:%s\n",net_mask_string);
}
```

使用 Visual C++ 6.0 来编译此源文件。生成一个工程，在此工程中必须进行一些设置，包括添加头文件路径和静态链接库路径。

在 WinPcap 的安装过程中，已经把 WinPcap 开发包解压缩在一个路径下面，如 F:\wpdpack。在此文件夹下有子文件夹“include”和“lib”，分别把它们添加到 Visual C++ 的头文件路径和静态链接库路径下面。

具体过程是，选择 tools 菜单的子菜单 options，打开对话框 Options，选择 Directory 页面，然后在“Iuclude files”中添加路径 F:\WPDPACK\INCLUDE，如图 4-5 所示。

然后在 Library files 中添加路径 F:\WPDPACK\LIB，如图 4-6 所示。

最后，把 WinPcap 的静态链接库 wpcap.lib 添加到 Visual C++ 之中。具体过程是，选择菜单 Project 的子菜单 Settings，打开对话框 Project Settings，选择页面 Link，在项目 Object/library modules 中添加 wpcap.lib。由于在 Windows 中要使用其他网络函数操作，如 ntohs()，必须使用套接字函数，所以还应该添加静态链接库 wsock32.lib，如图 4-7 所示。

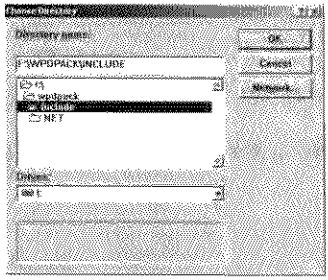

图 4-5　添加 WinPcap 的 Include 路径

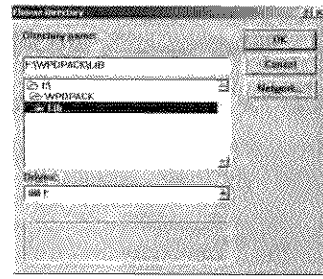

图 4-6　添加 WinPcap 的 Library 路径

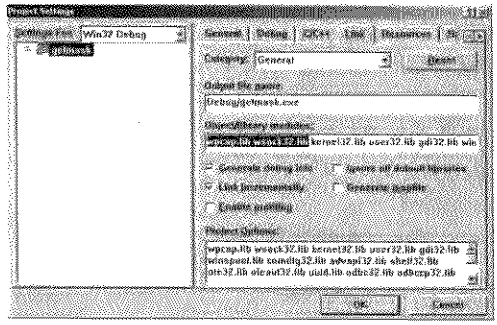

图 4-7　添加静态链接库 wpcap.lib 和 wsock32.lib

以后在使用 WinPcap 进行程序设计时，都应该进行这样的设置。

设置完之后，就可以编译了。编译运行的结果如图 4-8 所示。

图 4-8　获得掩码地址

上面的例子是获得网络地址和网络掩码，下面举一个例子，使用 WinPcap 捕获一个网络数据包。此例子的源代码如下：

```
#include <pcap.h>                          /*WinPcap 头文件*/
    /*以下是以太网协议格式*/
struct ether_header
{
    u_int8_t  ether_dhost[6];              /*以太网目的地址*/
    u_int8_t  ether_shost[6];              /*源以太网地址*/
    u_int16_t ether_type;                  /*以太网类型*/
};
```

```
void main( )
{
    char error_content[PCAP_ERRBUF_SIZE];          /*存储错误信息*/
    pcap_t* pcap_handle;                           /*WinPcap 句柄*/
    const u_char *packet_content;                  /*数据包内容*/
    u_char *mac_string;                            /*以太网地址*/
    u_short ethernet_type;                         /*以太网类型*/
    bpf_u_int32 net_mask;                          /*掩码地址*/
    bpf_u_int32 net_ip;                            /*网络地址*/
    char *net_interface;                           /*网络接口*/
    struct pcap_pkthdr protocol_header;            /*数据包头部信息*/
    struct ether_header *ethernet_protocol;            /*以太网协议变量*/
    struct bpf_program bpf_filter;                     /*BPF 过滤规则*/
    char bpf_filter_string[] = "ip";                   /*过滤规则字符串*/
    net_interface = pcap_lookupdev(error_content);     /*获得网络接口*/
    pcap_lookupnet(net_interface, &net_ip, &net_mask, error_content);
        /*获得网络地址和网络掩码*/
    pcap_handle = pcap_open_live(net_interface,BUFSIZ,1,0,error_content);
        /*打开网络接口*/
    pcap_compile(pcap_handle, &bpf_filter, bpf_filter_string, 0, net_ip);
        /*编译过滤规则*/
    pcap_setfilter(pcap_handle, &bpf_filter);          /*设置过滤规则*/
        if(pcap_datalink(pcap_handle)!=DLT_EN10MB)
                return;
    packet_content = pcap_next(pcap_handle,&protocol_header);
        /*捕获一个数据包，数据包内容返回给 packet_content*/
    printf("--------------------*****----------------------\n");
        printf    ("捕获到一个网络数据包\n");
    printf("捕获时间:\n");
    printf("%s",ctime((const time_t*)&protocol_header.ts.tv_sec));
    printf("数据包长度:\n");
    printf("%d\n",protocol header.len);
    ethernet_protocol = (struct ether_header *) packet_content;  /*获得数据包内容*/
    printf("以太网类型:\n");
    ethernet_type= ntohs(ethernet_protocol->ether_type);         /*获得以太网类型*/
    printf("%04x\n",ethernet_type);
    switch(ethernet_type)
    {
        case 0x0800:   printf("上层协议是 IP 协议\n");break;
        case 0x0806:   printf("上层协议是 ARP 协议\n"); break;
        case 0x8035:   printf("上层协议是 RARP 协议\n");break;
        default :break;
    }
```

```
    printf("源以太网地址: \n");
    mac_string=ethernet_protocol->ether_shost;
    printf("%02x:%02x:%02x:%02x:%02x:%02x\n",*mac_string,*(mac_string+1),
                *(mac_string+2),*(mac_string+3),
                *(mac_string+4),*(mac_string+5));
        /*获得源以太网地址*/
    printf("目的以太网地址: \n");
    mac_string=ethernet_protocol->ether_dhost;
    printf("%02x:%02x:%02x:%02x:%02x:%02x\n",*mac_string,*(mac_string+1),
                *(mac_string+2),*(mac_string+3),
                *(mac_string+4),*(mac_string+5));
        /*获得目的以太网地址*/
    printf("-----------------------------------------------------------\n");
    pcap_close(pcap_handle);                    /*关闭WinPcap操作*/
}
```

编译此程序，编译过程同上，运行结果如图 4-9 所示。

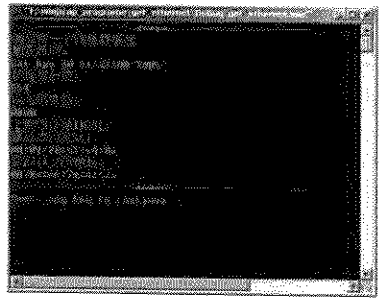

图 4-9　捕获一个数据包

此程序的功能是捕获一个网络数据包，然后对它的以太网协议进行分析。

上面的例子是使用函数 pcap_next() 捕获一个网络数据包。下面再举一个例子，它实现了捕获多个网络数据包的功能。此例子的源代码如下：

```
#include "pcap.h"                                   /*WinPcap头文件*/
struct ether_header
{
    u_int8_t  ether_dhost[6];                       /*目的以太网地址*/
    u_int8_t  ether_shost[6];                       /*源以太网地址*/
    u_int16_t ether_type;                           /*以太网类型*/
};
void ethernet_protocol_packet_callback(u_char *argument_wentaoliu,const struct
pcap_pkthdr* packet_header,const u_char* packet_content)
{
    u_short ethernet_type;                          /*以太网类型*/
```

```
    struct ether_header *ethernet_protocol;                /*以太网协议变量*/
    u_char *mac_string;                                    /*以太网地址*/
    static int packet_number = 1;                          /*捕获数据包个数*/
    printf("**************************************************\n");
    printf("捕获第%d个以太网数据包\n",packet_number);
        printf("捕获时间:\n");
    printf("%s",ctime((const time_t*)&packet_header->ts.tv_sec));
        printf("数据包长度:\n");
    printf("%d\n",packet_header->len);
    printf("-----------    以太网协议   ------------\n");
    ethernet_protocol = (struct ether_header *) packet_content;
        /*获得以太网协议内容*/
    printf("以太网类型:\n");
    ethernet_type= ntohs(ethernet_protocol->ether_type);          /*获得以太网类型*/
    printf("%04x\n",ethernet_type);
    switch(ethernet_type)
    {
        case 0x0800:   printf("上层协议为 IP 协议\n");break;
        case 0x0806:   printf("上层协议是 ARP 协议\n"); break;
        case 0x8035:   printf("上层协议为 RARP 协议\n");break;
        default :break;
    }
    printf("源以太网地址: \n");
    mac_string=ethernet_protocol->ether_shost;
    printf("%02x:%02x:%02x:%02x:%02x:%02x\n",*mac_string,*(mac_string+1),
                *(mac_string+2),*(mac_string+3),
                *(mac_string+4),*(mac_string+5));
        /*获得源以太网地址*/
    printf("目的以太网地址: \n");
    mac_string=ethernet_protocol->ether_dhost;
    printf("%02x:%02x:%02x:%02x:%02x:%02x\n",*mac_string,*(mac_string+1),
                *(mac_string+2),*(mac_string+3),
                *(mac_string+4),*(mac_string+5));
        /*获得目的以太网地址*/
    printf("**************************************************\n");
    packet_number++;
}
void  main()
{
    pcap_t* pcap_handle;                          /*WinPcap 句柄*/
    char error_content[PCAP_ERRBUF_SIZE];         /*存储错误信息*/
    char *net_interface;                          /*网络接口*/
    struct bpf_program bpf_filter;                /*BPF 过滤规则*/
```

```
char bpf_filter_string[] = "ip";                    /*过滤规则字符串*/
bpf_u_int32 net_mask;                               /*掩码地址*/
bpf_u_int32 net_ip;                                 /*网络地址*/
net_interface = pcap_lookupdev(error_content);      /*获得可用的网络接口*/
pcap_lookupnet(net_interface, &net_ip, &net_mask, error_content);
    /*获得网络地址和掩码地址*/
pcap_handle = pcap_open_live(net_interface,BUFSIZE,1, error_content);
    /*打开网络接口*/
pcap_compile(pcap_handle, &bpf_filter, bpf_filter_string, 0, net_ip);
    /*编译 BPF 过滤规则*/
pcap_setfilter(pcap_handle, &bpf_filter);           /*设置过滤规则*/
    if(pcap_datalink(pcap_handle)!=DLT_EN10MB)
        return;
pcap_loop(pcap_handle,-1,ethernet_protocol_packet_callback,NULL);
    /*注册回调函数，循环捕获数据包，然后调用回调函数处理此数据包*/
    pcap_close(pcap_handle);
    /*关闭 WinPcap 操作*
}
```

编译此程序，编译过程同上，运行结果如图 4-10 所示。

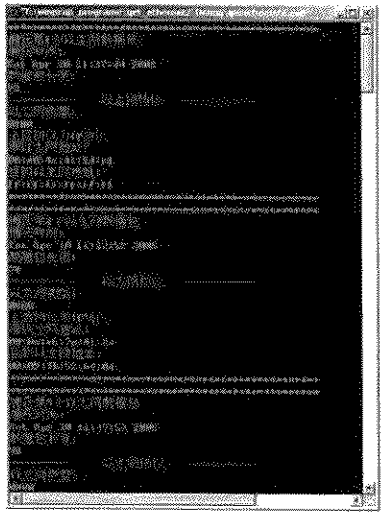

图 4-10　捕获多个数据包

4.5.2　捕获数据包

第 3 章在介绍使用 Libpcap 开发包时，举过一个综合的例子，此例子是捕获网络中的所有数据包，然后分析数据包使用了哪些协议。在此也举相同的例子，比较一下使用 WinPcap

和 Libpcap 的区别。其实，它们基本上没有什么区别，只是修改了一小部分内容。例如，IP 地址格式的定义在此删除了，因为在 Windows 平台下它已经被定义了，可以直接使用。另外一些网络函数必须加上它们的静态库文件，如函数 ntohs()，它可以使用静态库文件 wsock32.lib 加以解决。其他地方基本上是一样的。

下面是此例子的源代码：

```
/*文件名字: getpacket.c 刘文涛编写*/
#include "pcap.h"                          /*WinPcap 头文件*/
/*以下是以太网协议格式的定义*/
struct ether_header
{
    u_int8_t  ether_dhost[6];              /*目的以太网地址*/
    u_int8_t  ether_shost[6];              /*源以太网地址*/
    u_int16_t ether_type;                  /* 以太网类型*/
};
/*下面是 ARP 协议格式的定义*/
struct arp_header
{
    u_int16_t  arp_hardware_type;                          /*硬件类型*/
    u_int16_t  arp_protocol_type;                          /*协议类型*/
    u_int8_t   arp_hardware_length;                        /*硬件地址长度*/
    u_int8_t   arp_protocol_length;                        /*协议地址长度*/
    u_int16_t  arp_operation_code;                         /*操作码*/
    u_int8_t arp_source_ethernet_address[6];               /*源以太网地址*/
    u_int8_t arp_source_ip_address [4];                    /*源 IP 地址*/
    u_int8_t arp_destination_ethernet_address [6];         /*目的以太网地址*/
    u_int8_t arp_destination_ip_address[4];                /*目的 IP 地址*/
};
/*下面是 IP 协议格式的定义*/
struct ip_header
{
    #if defined(WORDS_BIGENDIAN)
    u_int8_t      ip_version:4,                            /*版本*/
                  ip_header_length:4;                      /*首部长度*/
    #else
    u_int8_t      ip_header_length:4,
                  ip_version:4;
    #endif
    u_int8_t      ip_tos;                                  /*服务质量*/
    u_int16_t     ip_length;                               /*长度*/
    u_int16_t     ip_id;                                   /*标识*/
    u_int16_t     ip_off;                                  /*偏移*/
    u_int8_t      ip_ttl;                                  /*生存时间*/
    u_int8_t      ip_protocol;                             /*协议类型*/
```

```
    u_int16_t      ip_checksum;                          /*校验和*/
    struct in_addr   ip_souce_address;                   /*源 IP 地址*/
    struct in_addr   ip_destination_address;             /*目的 IP 地址*/
};
/*下面是 UDP 协议格式定义*/
struct udp_header
{
    u_int16_t udp_source_port;                           /*源端口号*/
    u_int16_t udp_destination_port;                      /*目的端口号*/
    u_int16_t udp_length;                                /*长度*/
    u_int16_t udp_checksum;                              /*校验和*/
};
/*下面是 TCP 协议格式的定义*/
struct tcp_header
{
    u_int16_t tcp_source_port;                           /*源端口号*/
    u_int16_t tcp_destination_port;                      /*目的端口号*/
    u_int32_t tcp_sequence_lliiuuwweennttaaoo;           /*序列号*/
    u_int32_t tcp_acknowledgement;                       /*确认序列号*/
    #ifdef WORDS_BIGENDIAN
    u_int8_t  tcp_offset:4,                              /*偏移*/
          tcp_reserved:4;                                /*未用*/
    #else
    u_int8_t  tcp_reserved:4,                            /*未用*/
          tcp_offset:4;                                  /*偏移*/
    #endif
    u_int8_t  tcp_flags;                                 /*标记*/
    u_int16_t tcp_windows;                               /*窗口大小*/
    u_int16_t tcp_checksum;                              /*校验和*/
    u_int16_t tcp_urgent_pointer;                        /*紧急指针*/
};
/*下面是 ICMP 协议格式的定义*/
struct icmp_header
{
    u_int8_t icmp_type;                                  /*ICMP 类型*/
    u_int8_t icmp_code;                                  /*ICMP 代码*/
    u_int16_t icmp_checksum;                             /*校验和*/
    u_int16_t icmp_id;                                   /*标识符*/
    u_int16_t icmp_sequence;                             /*序列码*/
};
/*下面是分析 TCP 协议的函数,其定义方式与回调函数相同*/
void tcp_protocol_packet_callback(u_char *argument,const struct pcap_pkthdr*
packet_header,const u_char* packet_content)
```

```
{
    struct tcp_header *tcp_protocol;                          /*TCP 协议变量*/
    u_char flags;                                             /*标记*/
    int header_length;                                        /*长度*/
    u_short source_port;                                      /*源端口*/
    u_short destination_port;                                 /*目的端口*/
    u_short  windows;                                         /*窗口大小*/
    u_short urgent_pointer;                                   /*紧急指针*/
    u_int sequence;                                           /*序列号*/
    u_int acknowledgement;                                    /*确认号*/
    u_int16_t checksum;                                       /*校验和*/
    tcp_protocol = (struct tcp_header *) (packet_content+14+20);      /*获得 TCP 协议内容*/
    source_port = ntohs(tcp_protocol->tcp_source_port);               /*获得源端口*/
    destination_port = ntohs(tcp_protocol->tcp_destination_port);     /*获得目的端口*/
    header_length = tcp_protocol->tcp_offset * 4;                     /*长度*/
    sequence = ntohl(tcp_protocol->tcp_sequence_lliiuuwweennttaaoo);  /*序列码*/
    acknowledgement = ntohl(tcp_protocol->tcp_acknowledgement);       /*确认序列码*/
    windows = ntohs(tcp_protocol->tcp_windows);                       /*窗口大小*/
    urgent_pointer = ntohs(tcp_protocol->tcp_urgent_pointer);         /*紧急指针*/
    flags = tcp_protocol->tcp_flags;                                  /*标识*/
    checksum =ntohs(tcp_protocol->tcp_checksum);                      /*校验和*/
    printf("-------  TCP 协议   -------\n");
    printf("源端口号:%d\n", source_port);
    printf("目的端口号:%d\n", destination_port);
    switch (destination_port)
        {
            case 80:printf("上层协议为 HTTP 协议\n");break;
            case 21:printf("上层协议为 FTP 协议\n");break;
            case 23:printf("上层协议为 TELNET 协议\n");break;
            case 25:printf("上层协议为 SMTP 协议\n");break;
            case 110:printf("上层协议为 POP3 协议\n");break;
            default:break;
        }
    printf("序列码:%u\n", sequence);
    printf("确认号:%u\n", acknowledgement);
    printf("首部长度:%d\n", header_length);
    printf("保留:%d\n", tcp_protocol->tcp_reserved);
    printf("标记:");
    if (flags & 0x08) printf("PSH ");
    if (flags & 0x10) printf("ACK ");
    if (flags & 0x02) printf("SYN ");
    if (flags & 0x20) printf("URG ");
    if (flags & 0x01) printf("FIN ");
```

```
        if (flags & 0x04) printf("RST ");
    printf("\n");
        printf("窗口大小:%d\n", windows);
        printf("校验和:%d\n", checksum);
        printf("紧急指针:%d\n", urgent_pointer);
}
/*下面是实现UDP协议分析的函数，函数类型与回调函数相同*/
void udp_protocol_packet_callback(u_char *argument,const struct pcap_pkthdr*
packet_header,const u_char* packet_content)
{
    struct udp_header *udp_protocol;                /*UDP协议变量*/
    u_short source_port;                            /*源端口*/
    u_short  destination_port;                      /*目的端口号*/
    u_short length;
    udp_protocol = (struct udp_header *) (packet_content+14+20);      /*获得UDP协议内容*/
    source_port = ntohs(udp_protocol->udp_source_port);               /*获得源端口*/
    destination_port = ntohs(udp_protocol->udp_destination_port);     /*获得目的端口 */
    length = ntohs(udp_protocol->udp_length);                         /*获得长度*/
    printf("----------  UDP协议    ----------\n");
    printf("源端口号:%d\n", source_port);
    printf("目的端口号:%d\n", destination_port);
    switch (destination_port)
        {
            case 138: printf("上层协议为NetBIOS数据报服务\n");   break;
            case 137: printf("上层协议为NetBIOS名字服务\n");break;
            case 139: printf("上层协议为NetBIOS会话服务n");break;
            case 53:  printf("上层协议为域名服务\n");break;
            default:break;
        }
    printf("长度:%d\n", length);
    printf("校验和:%d\n", ntohs(udp_protocol->udp_checksum));
}
/*下面是实现分析ICMP协议的函数，函数类型与回调函数相同*/
void icmp_protocol_packet_callback(u_char *argument,const struct pcap_pkthdr*
packet_header,const u_char* packet_content)
{
    struct icmp_header *icmp_protocol;                        /*ICMP协议变量*/
    icmp_protocol = (struct icmp_header *) (packet_content+14+20);
        /*获得ICMP协议内容*/
    printf("----------  ICMP协议    ----------\n");
    printf("ICMP类型:%d\n", icmp_protocol->icmp_type);        /*获得ICMP类型*/
    switch (icmp_protocol->icmp_type)
    {
```

```
        case 8:
        printf("ICMP 回显请求协议\n");
        printf("ICMP 代码:%d\n", icmp_protocol->icmp_code);
        printf("标识符:%d\n", icmp_protocol->icmp_id);
        printf("序列码:%d\n", icmp_protocol->icmp_sequence);
        break;
        case 0:
        printf("ICMP 回显应答协议\n");
        printf("ICMP 代码:%d\n", icmp_protocol->icmp_code);
        printf("标识符:%d\n", icmp_protocol->icmp_id);
        printf("序列码:%d\n", icmp_protocol->icmp_sequence);
            break;
        default :break;
    }
    printf("ICMP 校验和:%d\n", ntohs(icmp_protocol->icmp_checksum));
        /*获得 ICMP 校验和*/
    return;
}
/*下面是实现 ARP 协议分析的函数，函数类型与回调函数相同*/
void arp_protocol_packet_callback(u_char *argument,const struct pcap_pkthdr*
packet_header,const u_char* packet_content)
{
    struct arp_header *arp_protocol;
    u_short protocol_type;
    u_short hardware_type;
    u_short operation_code;
    u_char *mac_string;
    struct in_addr source_ip_address;
    struct in_addr destination_ip_address;
    u_char hardware_length;
    u_char protocol_length;
    printf("--------  ARP 协议  --------\n");
    arp_protocol = (struct arp_header *) (packet_content+14);
    hardware_type = ntohs(arp_protocol->arp_hardware_type);
    protocol_type = ntohs(arp_protocol->arp_protocol_type);
    operation_code = ntohs(arp_protocol->arp_operation_code);
    hardware_length=arp_protocol->arp_hardware_length;
    protocol_length=arp_protocol->arp_protocol_length;
    printf("硬件类型:%d\n", hardware_type);
    printf("协议类型 Protocol Type:%d\n", protocol_type);
    printf("硬件地址长度:%d\n",hardware_length);
    printf("协议地址长度:%d\n",protocol_length);
    printf("ARP Operation:%d\n", operation_code);
```

```
        printf("总长度:%d\n", ntohs(ip_protocol->ip_length));
        printf("标识:%d\n", ntohs(ip_protocol->ip_id));
        printf("偏移:%d\n", (offset & 0x1fff) * 8);
        printf("生存时间:%d\n", ip_protocol->ip_ttl);
        printf("协议类型:%d\n", ip_protocol->ip_protocol);
    switch (ip_protocol->ip_protocol)
        {
            case 6:   printf("上层协议为 TCP 协议\n");break;
            case 17:  printf("上层协议为 UDP 协议\n");break;
            case 1:   printf("上层协议为 ICMP 协议 ICMP\n");break;
            default:break;
        }
    printf("校验和:%d\n", checksum);
    printf("源 IP 地址:%s\n", inet_ntoa(ip_protocol->ip_souce_address));
        /*获得源 IP 地址*/
    printf("目的 IP 地址:%s\n", inet_ntoa(ip_protocol->ip_destination_address));
        /*获得目的 IP 地址*/
    switch (ip_protocol->ip_protocol) /*根据 IP 协议判断上层协议*/
        {
            case 6:
        tcp_protocol_packet_callback(argument,
                                     packet_header,
                                     packet_content);
        break;
        /*上层协议是 TCP 协议，调用分析 TCP 协议的函数，注意参数的传递*/
        case 17:
        udp_protocol_packet_callback(argument,
                                     packet_header,
                                     packet_content);
        break;
        /*上层协议是 UDP 协议，调用分析 UDP 协议的函数，注意参数的传递*/
            case 1:
        icmp_protocol_packet_callback(argument,
                                     packet_header,
                                     packet_content);
        break;
        /*上层协议是 ICMP 协议，调用分析 ICMP 协议的函数，注意参数的传递*/
            default:break;
        }
}
/*下面是分析以太网协议的函数，也是回调函数*/
void ethernet_protocol_packet_callback(u_char *argument,const struct pcap_pkthdr*
packet_header,const u_char* packet_content)
```

```
{
    u_short ethernet_type;                          /*以太网类型*/
    struct ether_header *ethernet_protocol;         /*以太网协议变量*/
    u_char *mac_string;                             /*以太网地址*/
    static int packet_number = 1;                   /*数据包个数，静态变量*/
    printf("**************************************************\n");
    printf("捕获第%d个网络数据包\n",packet_number);
        printf("捕获时间:\n");
    printf("%s",ctime((const time_t*)&packet_header->ts.tv_sec));
        /*获得捕获数据包的时间*/
        printf("数据包长度:\n");
    printf("%d\n",packet_header->len);
        printf("--------   以太网协议   --------\n");
    ethernet_protocol = (struct ether_header *) packet_content;
        /*获得以太网协议内容*/
    printf("类型:\n");
    ethernet_type= ntohs(ethernet_protocol->ether_type);
        /*获得以太网类型*/
    printf("%04x\n",ethernet_type);
    switch(ethernet_type)        /*根据以太网类型判断*/
    {
        case 0x0800:   printf("上层协议为 IP 协议\n");break;
        case 0x0806:   printf("上层协议为 ARP 协议\n"); break;
        case 0x8035:   printf("上层协议为 RARP 协议\n");break;
        default :break;
    }
    printf("源以太网地址: \n");
    mac_string=ethernet_protocol->ether_shost;
    printf("%02x:%02x:%02x:%02x:%02x:%02x\n",*mac_string,*(mac_string+1),
                *(mac_string+2),*(mac_string+3),
                *(mac_string+4),*(mac_string+5));
        /*获得源以太网地址*/
    printf("目的以太网地址: \n");
    mac_string=ethernet_protocol->ether_dhost;
    printf("%02x:%02x:%02x:%02x:%02x:%02x\n",*mac_string,*(mac_string+1),
                *(mac_string+2),*(mac_string+3),
                *(mac_string+4),*(mac_string+5));
        /*获得目的以太网地址*/
    switch(ethernet_type)
    {
        case 0x0806:
        arp_protocol_packet_callback(argument,
                                     packet_header,
```

```
                                        packet_content);
            break;
            /*上层协议为ARP协议，调用分析ARP协议的函数，注意参数的传递*/
            case 0x0800:
            ip_protocol_packet_callback(argument,
                                        packet_header,
                                        packet_content);
            break;
            /*上层协议为IP协议，调用分析IP协议的函数，注意参数的传递*/
            default :break;
        }
        printf("*************************************************\n");
        packet_number++;
}
/*主函数*/
void  main()
{
        pcap_t* pcap_handle;                          /*WinPcap句柄*/
        char error_content[PCAP_ERRBUF_SIZE];         /*存储错误信息*/
        char *net_interface;                          /*网络接口*/
        struct bpf_program bpf_filter;                /*BPF过滤规则*/
        char bpf_filter_string[] = "";                /*过滤规则字符串*/
        bpf_u_int32 net_mask;                         /*掩码*/
        bpf_u_int32 net_ip;                           /*网络地址*/
        net_interface = pcap_lookupdev(error_content);  /*获得可用的网络接口*/
        pcap_lookupnet(net_interface,
                       &net_ip,
                       &net_mask,
                       error_content);                /*获得网络地址和掩码地址*/
        pcap_handle = pcap_open_live(net_interface,
                                     BUFSIZ,
                                     1,
                                     1,
                                     error_content);  /*打开网络接口*/
        pcap_compile(pcap_handle,
                     &bpf_filter, bpf_filter_string, 0, net_ip);      /*编译BPF过滤规则*/
        pcap_setfilter(pcap_handle, &bpf_filter);                     /*设置过滤规则*/
        if(pcap_datalink(pcap_handle)!=DLT_EN10MB)
                return;
        pcap_loop(pcap_handle,
                -1,
                ethernet_protocol_packet_callback,
                NULL);
```

```
        /*注册回调函数，循环捕获网络数据包，利用回调函数来处理每个数据包*/
        pcap_close(pcap_handle);
        /*关闭WinPcap操作*/
}
```

此程序的功能是捕获网络上的所有数据包，并对捕获的数据包进行分析，所分析的协议包括以太网协议、ARP 协议、IP 协议、ICMP 协议、TCP 协议和 UDP 协议。

编译此程序，编译过程同上，运行后的输出内容如下（只摘录了 5 个数据包的内容）：

```
************************************************
捕获第1个网络数据包
捕获时间：
Sat Apr 30 11:48:16 2005
数据包长度：
42
--------  以太网协议  --------
类型：
0806
上层协议为ARP协议
源以太网地址：
00:0a:e6:7a:41:2a
目的以太网地址：
ff:ff:ff:ff:ff:ff
--------  ARP协议  --------
硬件类型:1
协议类型 Protocol Type:2048
硬件地址长度:6
协议地址长度:4
ARP Operation:1
ARP请求协议
源以太网地址：
00:0a:e6:7a:41:2a
源IP地址:192.168.0.3
目的以太网地址：
00:00:00:00:00:00
目的IP地址:192.168.0.4
************************************************
************************************************
捕获第2个网络数据包
捕获时间：
Sat Apr 30 11:48:16 2005
数据包长度：
60
--------  以太网协议  --------
类型：
```

0806

上层协议为 ARP 协议

源以太网地址：

00:00:e8:e7:b9:46

目的以太网地址：

00:0a:e6:7a:41:2a

-------- ARP 协议 --------

硬件类型:1

协议类型 Protocol Type:2048

硬件地址长度:6

协议地址长度:4

ARP Operation:2

ARP 应答协议

源以太网地址：

00:00:e8:e7:b9:46

源 IP 地址:192.168.0.4

目的以太网地址：

00:0a:e6:7a:41:2a

目的 IP 地址:192.168.0.3

****

****

捕获第 3 个网络数据包

捕获时间：

Sat Apr 30 11:48:16 2005

数据包长度：

78

-------- 以太网协议 --------

类型：

0800

上层协议为 IP 协议

源以太网地址：

00:0a:e6:7a:41:2a

目的以太网地址：

00:00:e8:e7:b9:46

---------- IP 协议 ----------

版本号:4

首部长度:20

服务质量:0

总长度:64

标识:19255

偏移:0

生存时间:64

协议类型:6

上层协议为 TCP 协议

校验和:28201

源 IP 地址:192.168.0.3

目的 IP 地址:192.168.0.4

------- TCP 协议 -------

源端口号:2466

目的端口号:23

上层协议为 Telnet 协议

序列码:1649784010

确认号:0

首部长度:44

保留:0

标记:SYN

窗口大小:44620

校验和:58752

紧急指针:0

****

****

捕获第 4 个网络数据包

捕获时间:

Sat Apr 30 11:48:16 2005

数据包长度:

60

-------- 以太网协议 --------

类型:

0800

上层协议为 IP 协议

源以太网地址:

00:00:e8:e7:b9:46

目的以太网地址:

00:0a:e6:7a:41:2a

---------- IP 协议 ----------

版本号:4

首部长度:20

服务质量:0

总长度:40

标识:187

偏移:0

生存时间:128

协议类型:6

上层协议为 TCP 协议

校验和:47293

源 IP 地址:192.168.0.4

目的 IP 地址:192.168.0.3

------- TCP 协议 ---- --

源端口号:23

目的端口号:2466

序列码:0

确认号:1649784011

首部长度:20

保留:0

标记:ACK RST

窗口大小:0

校验和:3487

紧急指针:0

****

****

捕获第 5 个网络数据包

捕获时间:

Sat Apr 30 11:48:16 2005

数据包长度:

78

-------- 以太网协议 --------

类型:

0800

上层协议为 IP 协议

源以太网地址:

00:0a:e6:7a:41:2a

目的以太网地址:

00:00:e8:e7:b9:46

---------- IP 协议 ----------

版本号:4

首部长度:20

服务质量:0

总长度:64

标识:19256

偏移:0

生存时间:64

协议类型:6

上层协议为 TCP 协议

校验和:28200

源 IP 地址:192.168.0.3

目的 IP 地址:192.168.0.4

------- TCP 协议 -------

源端口号:2466

目的端口号:23

```
上层协议为 Telnet 协议
序列码:1649784810
确认号:0
首部长度:44
保留:0
标记:SYN
窗口大小:44620
校验和:58752
紧急指针:0
*****************************************
```

编译并运行此程序，其运行结果如图 4-11 所示。

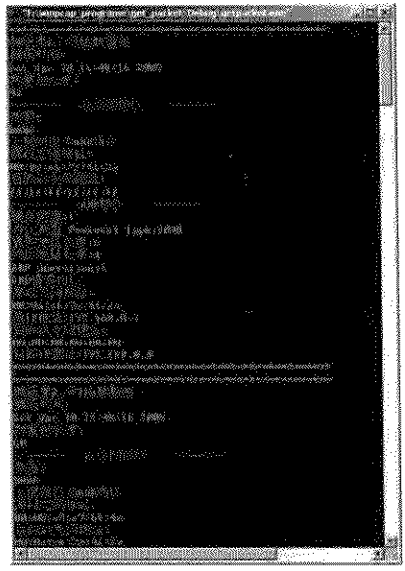

图 4-11　捕获网络数据包

此程序是前面介绍 Libpcap 时所举的综合例子的翻版。其功能是一样的，并且程序的语句也差不多。

4.5.3　发送数据包

使用 WinPcap 可以将数据包发送到网络上，这是 Libpcap 在 Linux 下无法实现的。至少目前还不行。而使用 WinPcap 可以方便地在 Windows 平台下将原始的数据包发送到网络上。下面举一个简单的例子加以说明。此程序的源代码如下：

```
#include "pcap.h"                                    /*WinPcap 头文件*/
void main()
```

```
{
    pcap_t* pcap_handle;                        /*WinPcap 句柄*/
    char *net_interface;                        /*网络接口*/
    char error_content[PCAP_ERRBUF_SIZE];       /*存储错误信息*/
    u_int8_t
    packet_content[94]="\x00\x10\x67\x00\xB1\x86\x0D\x0E\x0A\x0D\x00\x00\x08\x00\x45\x00
    \x00\x50\x00\xF2\x00\x00\x40\x06\xF8\x60\xC0\xA8\x00\x02\xC0\xA8\x00\x03\x00\x63\x00
    \x50\x01\x01\x01\x01\x02\x02\x02\x02\xA0\x02\x7F\xFF\xB6\xF4\x00\x0A\x03\x03\x0A\x01
    \x02\x04\x01\x09\x08\x0A\x3F\x3F\x3F\x3F\x00\x00\x00\x00\x00\x00\x30\x31\x32\x33\x34
    \x35\x36\x37\x38\x39\x30\x31\x32\x33\x34\x35\x36\x37\x38\x39";
    /*要发送的原始数据包*/
    net_interface = pcap_lookupdev(error_content);          /*获得网络接口*/
    pcap_handle = pcap_open_live(net_interface,             /*网络接口*/
                                 94,                        /*数据包大小*/
                                 1,                         /*混杂模式*/
                                 1,                         /*等待时间*/
                                 error_content);            /*错误信息*/
        /*打开网络接口*/
    pcap_sendpacket(pcap_handle,                            /*WinPcap 句柄*/
                    packet_content,                         /*数据包缓存内容*/
                    94);                                    /*数据包长度*/
        /*发送网络数据包*/
        pcap_close(pcap_handle);
        /*关闭 WinPcap 操作*/
}
```

该程序的功能非常简单，即发送一个原始的网络数据包，此数据包可以是任意字节的一段数据。此程序用十六进制显示。打开网络接口，然后调用发送数据包的函数 pcap_sendpacket()，就可以把网络数据包发送到网络上。

为了分析程序运行的结果，这里使用著名的捕获网络数据包的工具 Ethereal 来捕获刚才发送的网络数据包。其捕获结果如图 4-12 所示。

使用 Ethereal 的导出功能，把此数据包的详细信息导入一个文本文件之中，其数据包的详细信息如下：

```
Frame 1 (94 bytes on wire, 94 bytes captured)
    Arrival Time: Apr 23, 2005 15:48:46.406433000
    Time delta from previous packet: 0.000000000 seconds
    Time since reference or first frame: 0.000000000 seconds
    Frame Number: 1
    Packet Length: 94 bytes
    Capture Length: 94 bytes
    Protocols in frame: eth:ip:tcp:http:data
Ethernet II, Src: 0d:0e:0a:0d:00:00, Dst: 00:10:67:00:b1:86
    Destination: 00:10:67:00:b1:86 (RedbackN_00:b1:86)
```

```
    Source: 0d:0e:0a:0d:00:00 (0d:0e:0a:0d:00:00)
    Type: IP (0x0800)
Internet Protocol, Src Addr: 192.168.0.2 (192.168.0.2), Dst Addr: 192.168.0.3 (192.168.0.3)
    Version: 4
    Header length: 20 bytes
    Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
        0000 00.. = Differentiated Services Codepoint: Default (0x00)
        .... ..0. = ECN-Capable Transport (ECT): 0
        .... ...0 = ECN-CE: 0
    Total Length: 80
    Identification: 0x00f2 (242)
    Flags: 0x00
        0... = Reserved bit: Not set
        .0.. = Don't fragment: Not set
        ..0. = More fragments: Not set
    Fragment offset: 0
    Time to live: 64
    Protocol: TCP (0x06)
    Header checksum: 0xf860 (correct)
    Source: 192.168.0.2 (192.168.0.2)
    Destination: 192.168.0.3 (192.168.0.3)
Transmission Control Protocol, Src Port:99 (99), Dst Port: http (80), Seq: 0, Ack: 0, Len: 20
    Source port: 99 (99)
    Destination port: http (80)
    Sequence number: 0    (relative sequence number)
    Next sequence number: 20    (relative sequence number)
    Header length: 40 bytes
    Flags: 0x0002 (SYN)
        0... .... = Congestion Window Reduced (CWR): Not set
        .0.. .... = ECN-Echo: Not set
        ..0. .... = Urgent: Not set
        ...0 .... = Acknowledgment: Not set
        .... 0... = Push: Not set
        .... .0.. = Reset: Not set
        .... ..1. = Syn: Set
        .... ...0 = Fin: Not set
    Window size: 32767
    Checksum: 0xb6f4 (correct)
    Options: (20 bytes)
        Window scale: 10 (multiply by 1024)
        NOP
        Maximum segment size: 265 bytes
        Time stamp: tsval 1061109567, tsecr 0
```

```
        EOL
Hypertext Transfer Protocol
    Data (20 bytes)
```

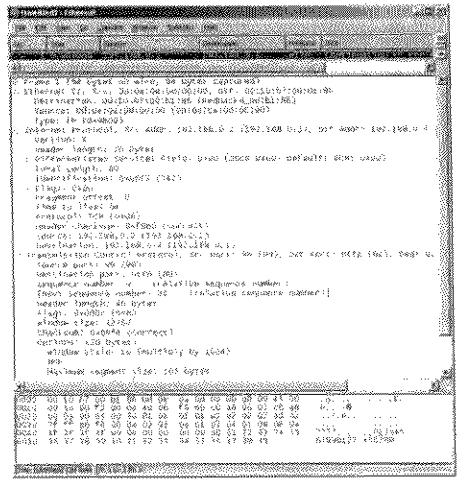

图 4-12　捕获所发送的数据包

从捕获到的数据包的详细内容可看出，此数据包的长度为 94 字节，链路层协议为以太网协议，网络层协议为 IP 协议，传输层协议为 TCP 协议，应用层协议为 HTTP 协议。但是，在这个程序中构造的数据包只是一个原始的字节流，也就是数组 packet_content 的内容，在这里是用十六进制表示的。其中，每个协议部分不能很方便地任意控制。比如，IP 协议部分的每个字段包括源 IP 地址、目的 IP 地址等内容，都是原始的字节流信息。所以，使用 WinPcap 来发送数据包是很不方便的。为了使构造和发送网络数据包的过程变得非常方便，于是产生了著名的网络数据包构造和发送开发包 Libnet，在第 5 章将重点讨论。使用 Libnet 来构造数据包，可以构造任意的网络数据包，而且所构造的数据包的内容可以自由控制，非常方便、灵活。

4.6　小结

Windows 平台下的专业网络数据包捕获开发包 WinPcap，是 Libpcap 开发包在 Windows 平台下的移植。使用 WinPcap 可以实现很多功能，它的性能优越，使用灵活，且与 Libpcap 兼容。

本章讨论了 WinPcap 的基本原理，介绍了 Windows 平台下的网络驱动程序接口规范（NDIS）。在 WinPcap 中使用了数据包的捕获机制 NPF，而 NPF 就是 NDIS 驱动类型中的协议驱动部分。从这点可以看到 NPF 和 NDIS 之间的联系。

在 WinPcap 中包括两种编程接口，一个是较低层的编程接口 packet.dll，另一个是较高层的编程接口 wpcap.dll。packet.dll 的编程接口与各个操作系统密切联系，可以直接与内核

进行通信，但是使用 packet.dll 编程接口比较烦琐，所以应用不多。而 wpcap.dll 编程接口是与操作系统无关的更高层的编程接口，它的使用方法与 Libpcap 兼容，所以使用 wpcap.dll 编程接口非常方便简单。一般情况下，都是使用 wpcap.dll 来实现 Windows 平台下数据包的捕获功能。

由于 WinPcap 与 Libpcap 兼容，所以它的使用方法与 Libpcap 几乎完全相同。它的数据结构和输出函数都跟 Libpcap 差不多，本章对它们逐一进行了简单介绍，更详细的内容可以参考前面 Libpcap 的介绍。另外，在 WinPcap 中有与 Windows 相关的一些函数，本章对它们进行了详细介绍。最后，通过举例对 WinPcap 的使用方法进行了详细的阐述，可以看到使用 WinPcap 的开发流程跟 Libpcap 一致。

第 5 章　数据包构造和发送开发包 Libnet

5.1　Libnet 介绍

Libnet 是一个专业网络数据包构造和发送开发包，它是一个高层次的 API 函数库，允许开发者构造和发送网络数据包。Libnet 提供了一个对底层网络数据包进行构造、修改和发送的高级接口。它隐藏了很多底层细节，省去了很多麻烦，如多路技术、缓冲区管理、网络数据包头信息、字节流顺序、操作系统兼容性，以及校验和计算问题等。这些问题 Libnet 都帮我们解决了，所以开发者不用再考虑这些烦琐的问题了。Libnet 主要提供了在 IP 层和链路层构造网络数据包的功能和一些非常有用的辅助功能。使用 Libnet 可以非常轻松、快捷地构造任何形式的网络数据包，开发各种各样的网络安全应用程序，以至非常复杂的程序。

Libnet 主要由 Mike D. Schiffman 开发和维护，很多其他开发人员也作出了很多贡献。Libnet 是在 BSD 许可下发布的，它是开放源代码软件，可以从网站上免费下载。其下载地址为：http://www.packetfactory.net/libnet/dist/libnet.tar.gz。

Libnet 开发包的应用非常广泛，已经在很多软件中用到。使用 Libnet 开发的比较著名的软件如下。

1. Ettercap

Ettercap 是一个交换网络环境下的网络监视、捕获和记录工具，支持多种主动或被动的协议分析，有数据插入、过滤、保持连接同步等多种功能，也有一个能支持多种嗅探模式的、强大而完整的嗅探套件，支持插件技术，能够检查网络环境是否是交换局域网，并且能使用主动或被动的操作系统指纹识别技术，让你了解当前局域网的情况。Ettercap 有 5 种监听模式。

2. Firewalk

Firewalk 使用类似 traceroute 的技术来分析 IP 包的响应，从而测定网关的访问控制列表和绘制网络图。它还使用类似于路由跟踪的 IP 数据包分析方法，来测定一个特殊的数据包是否能够从攻击者的主机传送到位于数据包过滤设备后的目标主机。Firewalk 还能测定带有各种控制信息的数据包是否能通过给定网关。

3. Snort

Snort 是一个小型的网络入侵检测系统，可以完成实时网络流量分析和对网络上的 IP 包登录进行测试等功能，能完成协议分析、内容查找和匹配，可用来探测多种攻击和嗅探，如缓冲区溢出、秘密端口扫描、CGI 攻击、SMB 嗅探、拇纹采集尝试等攻击。其中有些模块用到 Libnet 开发包。

4. Tcpreplay

Tcpreplay 是基于 Libnet 的 TCP 操作软件，它能够再现网络流量，能把由 Tcpdump 等工

具保存下来的流量监测文件在任意点进行回放，这对于研究和测试网络安全是很有帮助的。

除了上述的一些软件使用了 Libnet 开发包之外，还有很多网络安全系统使用了 Libnet 开发包，如 RING 指纹探测软件等。

Libnet 的最新稳定版本为 1.1.2.1，最新测试版本为 1.1.3。本书以 1.1.2.1 版本作为介绍对象。应该注意，不同版本的 Libnet 开发包，其函数的调用形式都不相同。

5.1.1 Libnet 的由来

在一些网络安全系统中，经常碰到要手工产生网络数据包的情况。其中包括一些常见的基本操作，并且这些操作都具有一定的可重复性。如果能够把这些重复的操作用一些函数来实现，就可以屏蔽一些比较烦琐的操作。

手工产生网络数据包的基本步骤不外乎构造数据包和发送数据包。所构造数据包的种类就要根据数据包使用的协议来决定。例如，可以构造比较常用的网络数据包，像 ARP 网络数据包、IP 数据包、ICMP 数据包、TCP 数据包、UDP 数据包等。发送数据包的过程需要与操作系统打交道，特别是网络部分，包括网卡的设定、控制等。

在网络入侵检测系统中，如果系统检测到入侵行为，就必须做出响应，此响应可以是被动的，也可以是主动的。被动的响应是直接报警，告知发生了什么类型的入侵行为，然后管理者根据入侵行为做出反应，可以关闭系统，也可以切断网络连接，等等。主动的响应是整个系统不需要人为参与，而是系统自动做出积极主动的响应。例如，当发生入侵行为时，就可以发送相应的数据包来切断网络连接，也可以主动地回应攻击者，进行反攻击，当然这要符合相应的法律规则。可以看到，在入侵检测系统中，就用到了构造数据包的过程，可以构造不同的数据包进行不同的操作。

在网络安全扫描系统中，也要用到构造数据包的过程。例如，在远程检测主机的漏洞时，可以发送不同的数据包来进行测试，并查看相应的返回结果，然后制定远程主机的漏洞是什么。当然，不只是检测漏洞的过程，还有很多的探测功能，如扫描远程主机的端口是否打开，远程主机使用的是什么样的操作系统，等等。这些功能都可以通过构造不同的网络数据包，然后检测返回结果来实现。

构造数据包的功能，在不同的操作系统（如 Linux，Windows 以及 DOS 操作系统），其实现的方式是不相同的。如何屏蔽这些操作系统的不同，也是在设计开发包时应该考虑的。在 Windows 系统下，要构造一个网络数据包，必须进行一些相关的设置和操作，并且在构造不同的数据包时，有些设置和操作必须完成，也就是说它们是重复性的，并且都比较烦琐而容易出错。所以，为了减少开发的难度，减少出错的几率，人们想到开发一个构造数据包的开发包，利用它可以使一些重复的操作不需要进行，使一些容易出错的地方不用出现。Libnet 就是在这种情况下诞生的。

1998 年，Libnet 就已出现了，但那时 Libnet 的使用不是很方便，特别是内存处理和校验和的计算非常烦琐，并且它支持的协议种类也不太多。它的缺点是在构造和发送一个数据包时，要执行的步骤太多。从 2001 年开始，Libnet 的作者 Mike Schiffman 克服了这些缺点，使 Libnet 的实用性更强。例如，在计算校验和时开发者不需要进行计算，而由系统自动计算。至此，可以说 Libnet 开发包已经非常完善了，使用 Libnet 开发包的人越来越多，很多著名的网络安全软件也都使用 Libnet。

5.2 Libnet 的组成

5.2.1 文件

Libnet 主要由很多个源文件和头文件组成，包含 18 000 多行代码。其中源文件由以下一些文件组成：

- libnet_dll.c；
- libnet_erro.c；
- libnet_build_snmp.c；
- libnet_link_none.c；
- libnet_version.c；
- libnet_build_data.c；
- libnet_build_802.1x.c；
- libnet_prand.c；
- libnet_build_802.3.c；
- libnet_link_nit.c；
- libnet_build_link.c；
- libnet_build_rip.c；
- libnet_build_igmp.c；
- libnet_build_sebek.c；
- libnet_build_mpls.c；
- libnet_build_sebek.c；
- libnet_build_isl.c；
- libnet_build_udp.c；
- libnet_build_vrrp.c；
- libnet_build_802.1q.c；
- libnet_advanced.c；
- libnet_link_snit.c；
- libnet_build_dns.c；
- libnet_build_ntp.c；
- libnet_build_rpc.c；
- libnet_build_802.2.c；
- libnet_link_pf.c；
- libnet_link_snoop.c；
- libnet_build_dhcp.c；
- libnet_build_Ethernet.c；
- libnet_build_cdp.c；
- libnet_crc.c；
- libnet_build_arp.c；
- libnet_raw.c；

- libnet_init.c；
- libnet_build_token_ring.c；
- libnet_build_fddi.c；
- libnet_build_stp.c；
- libnet_link_linux.c；
- libnet_port_list.c；
- libnet_link_bpf.c；
- libnet_cq.c；
- libnet_link_win32.c；
- libnet_build_bgp.c；
- libnet_if_addr.c；
- libnet_build_tcp.c；
- libnet_internal.c；
- libnet_checksum.c；
- libnet_resolve.c；
- libnet_asn1.c；
- libnet_write.c；
- libnet_build_gre.c；
- libnet_pblock.c；
- libnet_build_icmp.c；
- libnet_link_dlpi.c；
- libnet_build_ospf.c；
- libnet_build_ip.c。

其中头文件主要有：

- libnet-asn1.h；
- libnet-functons.h；
- libnet-headers.h；
- libnet_macros.h；
- libnet-structures.h；
- libnet-types.h；
- config.h；
- getopt.h；
- in_system.h；
- libnet.h；
- bpf.h；
- gnuc.h；
- ifaddrlist.h。

一般情况下，每个文件实现一个协议的构造功能。

5.2.2 函数构成

Libnet 开发包的输出函数有 100 多个，主要由以下几类函数构成：

（1）核心函数；

（2）地址函数；

（3）数据包构造函数；

（4）数据包发送函数；

（5）高级处理函数；

（6）Libnet 句柄队列操作函数；

（7）辅助函数。

下面列出所有函数名称，并进行简单说明，其详细的函数定义和参数说明在 5.4 节中详细介绍，它们全部在头文件 libnet_functions.h 中声明。

- libnet_init() ——对 Libnet 环境进行初始化；
- libnet_build_802_1q() ——构造协议 802.1q；
- libnet_build_802_1x() ——构造协议 802.1x；
- libnet_build_802_2() ——构造协议 802.2；
- libnet_build_802_2snap() ——构造协议 802.2snap；
- libnet_build_802_3() ——构造协议 802.3；
- libnet_build_arp() ——构造协议 ARP；
- libnet_build_bgp4_header() ——构造协议 BGP4；
- libnet_build_bgp4_notification() ——构造协议 BGP4 NOFIFICATION；
- libnet_build_bgp4_open() ——构造协议 BGP4 OPEN；
- libnet_build_bgp4_update() ——构造协议 BGP4 UPDATE；
- libnet_build_bootpv4() ——构造协议 BOOTUP；
- libnet_build_cdp() ——构造协议 CDP；
- libnet_build_data() ——构造数据块；
- libnet_build_dhcpv4() ——构造协议 DHCP；
- libnet_build_dnsv4() ——构造协议 DNSv4；
- libnet_build_egre() ——构造协议 EGRE；
- libnet_build_ethernet() ——构造协议 Ethernet；
- libnet_build_fddi() ——构造协议 FDDI；
- libnet_build_gre() ——构造协议 GRE；
- libnet_build_gre_last_sre() ——构造协议 GRE LAST SRE；
- libnet_build_gre_sre() ——构造协议 GRE SRE；
- libnet_build_icmpv4_echo() ——构造协议 ICMP ECHO；
- libnet_build_icmpv4_mask() ——构造协议 ICMP 地址掩码；
- libnet_build_icmpv4_redirect() ——构造 ICMP 重定向协议。
- libnet_build_icmpv4_timeexceed() ——构造 ICMP 超时协议；
- libnet_build_icmpv4_timestamp() ——构造 ICMP 时间戳协议；
- libnet_build_icmpv4_unreach() ——构造 ICMP 不可达协议；

- libnet_build_igmp() ——构造 IGMP 协议；
- libnet_build_ipsec_ah() ——构造 IPSEC AH 协议；
- libnet_build_ipsec_esp_ftr() ——构造 IPSEC ESP FTR 协议；
- libnet_build_ipsec_esp_hdr() ——构造 IPSEC ESP HDR 协议；
- libnet_build_ipv4() ——构造 IPv4 协议；
- libnet_build_ipv4_options() ——构造 IPv4 选项协议；
- libnet_build_ipv6() ——构造 IPv6 协议；
- libnet_build_ipv6_destopts() ——构造 IPv6 目的选项协议；
- libnet_build_ipv6_frag() ——构造 IPv6 标志协议；
- libnet_build_ipv6_hbhopts() ——构造 IPv6 hbhopts 协议；
- libnet_build_ipv6_routing() ——构造 IP6 路由协议；
- libnet_build_isl() ——构造 ISL 协议；
- libnet_build_link() ——构造链路层协议；
- libnet_build_mpls() ——构造 MPLS 协议；
- libnet_build_ntp() ——构造 NTP 协议；
- libnet_build_ospfv2() ——构造 OSPFv2 协议；
- libnet_build_ospfv2_dbd() ——构造 OSPF DBD 协议；
- libnet_build_ospfv2_hello() ——构造 OSPF HELLO 协议；
- libnet_build_ospfv2_lsa() ——构造 OSPF LSA 协议；
- libnet_build_ospfv2_lsa_as() ——构造 OSPF LSA AS 协议；
- libnet_build_ospfv2_lsa_net() ——构造 OSPF LSA NET 协议；
- libnet_build_ospfv2_lsa_rtr() ——构造 OSPF LSA RTR 协议；
- libnet_build_ospfv2_lsa_sum() ——构造 OSPF LSA SUM 协议；
- libnet_build_ospfv2_lsr() ——构造 OSPF LSR 协议；
- libnet_build_ospfv2_lsu() ——构造 OSPF LSU 协议；
- libnet_build_rip() ——构造 RIP 协议；
- libnet_build_rpc_call() ——构造 RPC CALL 协议；
- libnet_build_sebek() ——构造 SEBEK 协议；
- libnet_build_stp_conf() ——构造 STP CONF 协议；
- libnet_build_stp_tcn() ——构造 STP TCN 协议；
- libnet_build_tcp() ——构造 TCP 协议；
- libnet_build_tcp_options() ——构造 TCP 选项协议；
- libnet_build_token_ring() ——构造令牌环协议；
- libnet_build_udp() ——构造 UDP 协议；
- libnet_build_vrrp() ——构造 VRRP 协议；
- libnet_autobuild_arp() ——构造 ARP 协议；
- libnet_autobuild_ethernet() ——构造 Ethernet 协议；
- libnet_autobuild_fddi() ——构造 FDDI 协议；
- libuet_autobuild_ipv4() ——构造 IPv4 协议；
- libnet_autobuild_ipv6() ——构造 IPv6 协议；

- libnet_autobuild_link() ——构造链路层协议；
- libnet_autobuild_token_ring() ——构造令牌环协议；
- libnet_write() ——发送数据包；
- libnet_addr2name4() ——IPv4 地址转换；
- libnet_addr2name6_r() ——IPv6 地址转换；
- libnet_adv_cull_header() ——构造协议头；
- libnet_adv_cull_packet() ——构造数据块；
- libnet_adv_free_packet() ——释放数据块；
- libnet_adv_write_link() ——发送数据块；
- libnet_clear_packet() ——清空数据块；
- libnet_cq_add() ——向句柄队列中添加句柄；
- libnet_cq_destroy() ——销毁句柄队列；
- libnet_cq_find_by_label() ——查询句柄队列；
- libnet_cq_getlabel() ——获取句柄标记；
- libnet_cq_head() ——获取句柄队列首结点；
- libnet_cq_last() ——获取句柄队列最后一个结点；
- libnet_cq_next() ——向下移动一个句柄结点；
- libnet_cq_remove() ——删除一个句柄结点；
- libnet_cq_remove_by_label() ——删除一个句柄结点；
- libnet_cq_size() ——获取句柄队列的大小；
- libnet_destroy() ——销毁 Libnet；
- libnet_diag_dump_context() ——输出句柄信息；
- libnet_diag_dump_bex() ——输出数据包信息；
- libnet_diag_dump_pblock() ——输出协议块信息；
- libnet_diag_dump_pblock_type() ——输出协议块类型；
- libnet_name2addr4() ——IPv4 地址转换；
- libnet_name2addr6() ——IPv6 地址转换；
- libnet_plist_chain_dump() ——输出端口队列；
- libnet_plist_chain_dump_string() ——输出端口队列；
- libnet_plist_chain_free() ——释放端口列表；
- libnet_plist_chain_new() ——构造端口列表；
- libnet_plist_chain_next_pair() ——获取下一个端口；
- libnet_seed_prand() ——设置随机种子；
- libnet_stats() ——获取 Libnet 状态信息；
- libnet_toggle_checksum() ——计算校验和；
- libnet_get_hwaddr() ——获取 MAC 地址；
- libnet_get_ipaddr4() ——获取 IPv4 地址；
- libnet_get_ipaddr6() ——获取 IPv6 地址；
- libnet_get_prand() ——获取一个随机数；
- libnet_getdevice() ——获取设备名称；

- libnet_geterror() ——获取错误信息；
- libnet_getfd() ——获取 fd 信息；
- libnet_getgre_length() ——获取 gre 长度；
- libnet_getpacket_size() ——获取数据包大小；
- libnet_getpbuf() ——获取缓冲区内容；
- libnet_getpbuf_size() ——获取缓冲区大小；
- libnet_hex_aton() ——信息转换。

5.2.3 协议部分

Libnet 支持的协议种类非常多，几乎包括所有常用的网络协议。下面分别从应用层、传输层、网络层和链路层进行介绍。

其支持的应用层协议有：

- 边界网关协议（BGP，Border Gateway Protocol）；
- 远程过程调用（RPC，Remote Procedure Call）协议；
- 域名系统（DNS，Domain Name Systems）；
- 网络时间协议（NTP，Network Time Protocol）；
- 引导程序协议 BOOTP（Bootstrap Protocol）；
- 动态主机配置协议（DHCP，Dynamic Host Configuration Protocol）。

其支持的传输层协议主要有：

- 传输控制协议（TCP，Transmission Control Protocol）；
- 用户数据报协议（UDP）。

其支持的网络层协议主要有：

- 路由信息协议（RIP，Routing Information Protocol）；
- 开放最短路径优先（OSPF，Open Shortest Path First）协议；
- 虚拟路由器冗余协议（VRRP，Virtual Router Redundancy Protocol）；
- 信报控制协议（ICMP，Internet Control Message Protocol）；
- 因特网组管理协议（IGMP，Internet Group Management Protocol）；
- 网际协议 IPv4；
- 网际协议 IPV6；
- 通用路由封装（GRE，Generic Routing Encapsulation）；
- 封装安全有效负荷（ESP，Encapsulating Security Payloads）；
- 认证头（AH，Authentication Headers）。

其支持的链路层协议主要有：

- 思科发现协议（CDP，Cisco Discovery Protocol）；
- 光纤分布式数据接口（FDDI，Fiber Distributed Data Interface）；
- 基于端口的访问控制协议 802.1X；
- 以太网（Ethernet）协议；
- 虚拟桥接局域网 802.1Q；
- IEEE 802.3；
- 多协议标记交换（MPLS）；

- 令牌环（Token Ring）协议（IEEE 802.5 LAN protocol）；
- 地址解析协议（ARP，Address Resolution Protocol）；
- IEEE 802.2；
- 逆向地址解析协议 RARP；
- 子网访问协议（SNAP，SubNetwork Access Protocol）。

5.2.4 Libnet 的安装

在安装 Libnet 开发包之前，必须先下载，从其主页下载最新版本的开发包，这里介绍的是版本 1.1.2.1。然后根据不同的平台进行解压缩，在此主要介绍在 Linux 平台和 Windows 平台下的安装过程。

1. 在 Linux 平台下安装

以 Red Hat Linux 为安装环境。由于 Libnet 使用了 Libpcap 开发包，所以在安装 Libnet 之前首先要安装 Libpcap 开发包。

（1）安装 Libpcap，过程如下：

先解压缩 Libpcap 开发包。

```
tar -xzvf libpcap-0.9.tar.gz
```

其过程如图 5-1 所示。

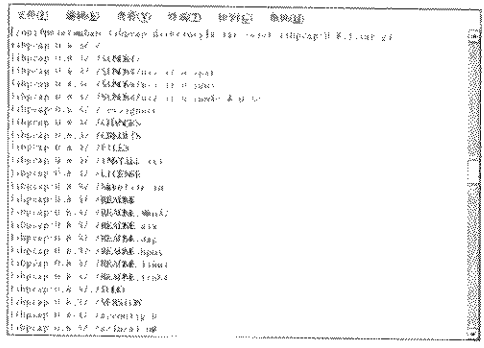

图 5-1 解压缩 Libpcap 安装包

再进入 Libpcap 目录，对 Libpcap 进行配置。

```
./configure
```

配置过程如图 5-2 所示。

执行 make 命令进行编译。

```
make
```

编译过程如图 5-3 所示。

最后执行 make install 进行安装。

```
make install
```

安装过程如图 5-4 所示。

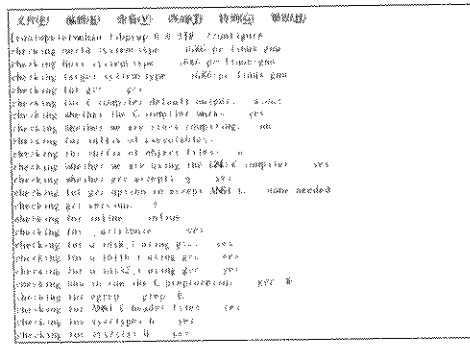

图 5-2　对 Libpcap 进行配置

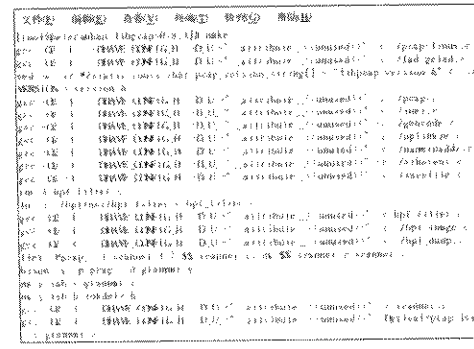

图 5-3　对 Libpcap 进行编译

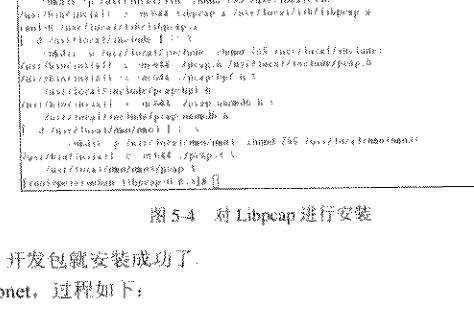

图 5-4　对 Libpcap 进行安装

此时 Libpcap 开发包就安装成功了。

（2）安装 Libnet，过程如下：

首先解压缩。

```
tar -xzvf libnet-1.1.2.1.tar.gz
```

其过程如图 5-5 所示。

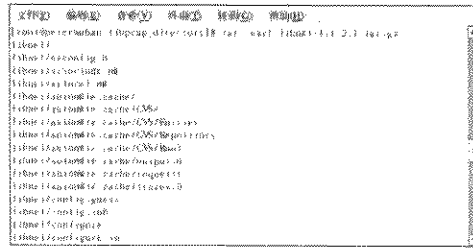

图 5-5　解压缩 Libnet 安装包

进入 Libnet 目录，执行命令 configure 进行配置。

```
./configure
```

配置过程如图 5-6 所示。

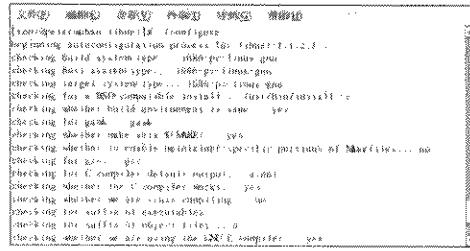

图 5-6　对 Libnet 进行配置

然后执行 make 命令进行编译。

```
make
```

编译过程如图 5-7 所示。

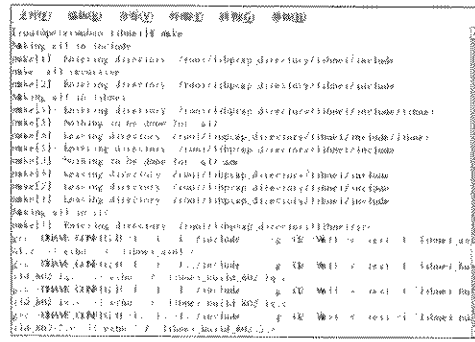

图 5-7　对 Libnet 进行编译

最后进行安装，执行命令 make install。

```
make install
```

安装过程如图 5-8 所示。

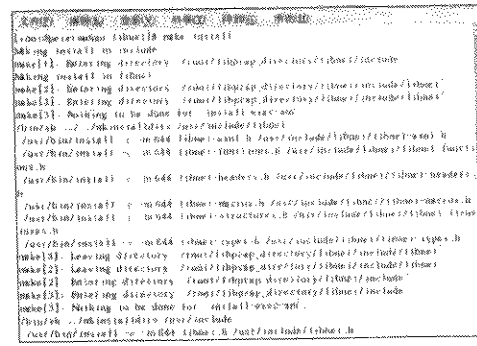

图 5-8　对 Libnet 进行安装

至此，Libnet 开发包就安装成功了。

2．在 Windows 平台下安装

在 Windows 平台下的安装，我们以 Windows 2000 SP4 为例进行介绍。其安装过程共分为三个步骤。

（1）第一步，安装 SDK。

首先下载 Windows SDK 开发平台，其下载地址为：

```
http://www.microsoft.com/msdownload/platformsdk/sdkupdate/
```

安装该开发平台，安装之后，在 Visual C++ 环境中进行设置。打开 Visual C++ 的 tools 菜单的子菜单 options，打开对话框 options，然后在 directories 页面中进行设置，在 include 路径中添加路径 SDK 的 include 路径：

```
C:\PROGRAM FILES\MICROSOFT SDK\INCLUDE
```

在 library 路径中添加 SDK 的 lib 路径：

```
C:\PROGRAM FILES\MICROSOFT SDK\LIB
```

（2）第二步，安装 WinPcap。

由于在 Libnet 中使用了 WinPcap 开发包，所以必须先安装 WinPcap 开发包。WinPcap 是 Windows 平台下的免费数据包捕获开发包。

从地址 http://winpcap.polito.it/install/bin/WinPcap_3_0.exe 直接下载。下载的文件是一个可执行的安装文件。

从地址 http://winpcap.polito.it/install/bin/wpdpack_3_0.zip 下载 WinPcap 的开发环境，其中包括头文件、lib 等。这两个部分都必须下载。

首先执行 winpcap_3_0.exe 安装文件，如图 5-9 所示。

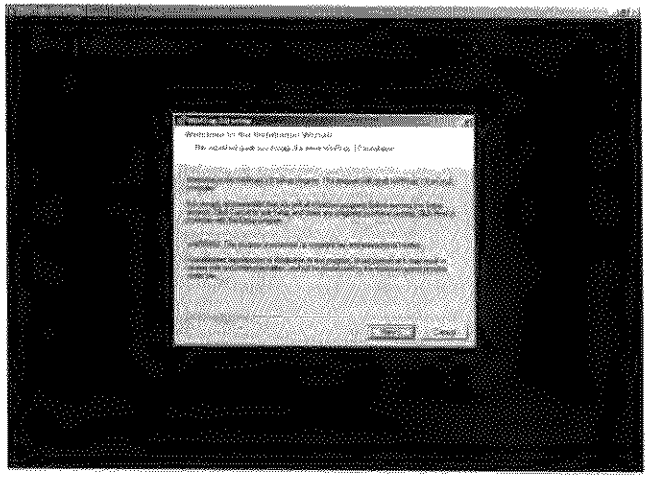

图 5-9　安装 WinPcap 步骤 1

单击“next”进行安装，如图 5-10 所示。

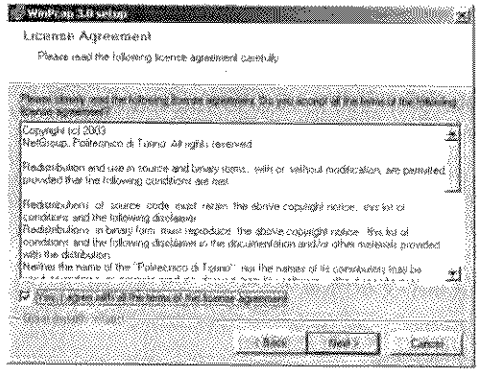

图 5-10　安装 WinPcap 步骤 2

选择复选框，再选择“next”，最后出现如图 5-11 所示的窗口，表示 WinPcap 已经安装成功了。

对 wpdpack_3_0.zip 的安装，只需解压缩到一个文件夹下就可以了，如图 5-12 所示。

此时，这个文件夹下面有一些子文件夹，最重要的两个是 include 文件夹和 lib 文件夹。include 文件夹是 WinPcap 开发包的头文件。lib 文件夹是 WinPcap 的静态库文件，也是使用 WinPcap 开发程序所必需的，其中包括 wpcap.lib 和 packet.lib。

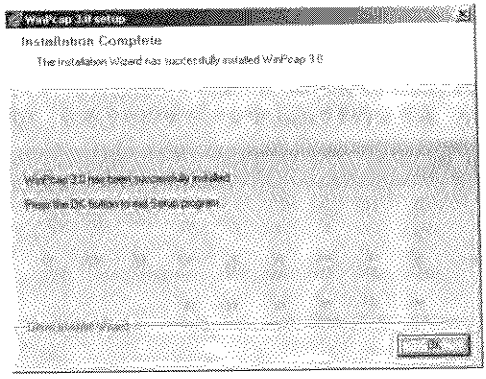

图 5-11　安装 WinPcap 成功

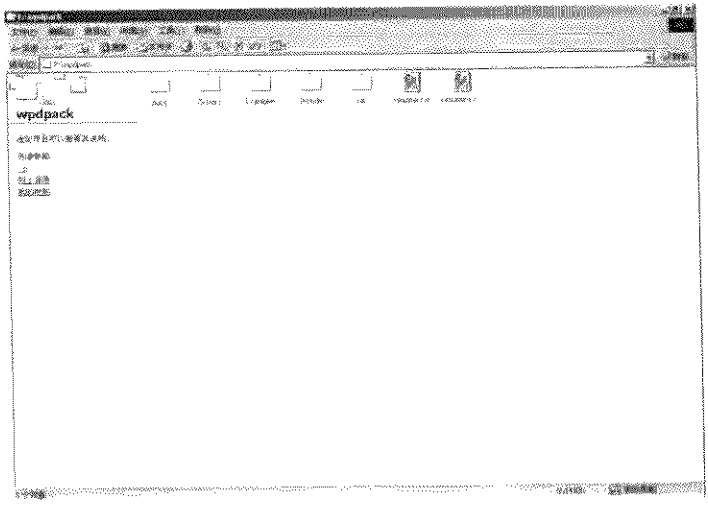

图 5-12　wpdpack_3_0.zip 的安装

在 Libnet 的头文件 libnet.h 中有以下两行语句：

```
#include "pcap.h"
#pragma comment( lib,"wpcap" )
```

所以，必须把 WinPcap 开发包的路径写进 Visual C++ 开发环境中。具体步骤是：选择 Visual C++ 菜单 tools 的子菜单 options，打开 options 对话框，然后在页面 directories 中设置 include 路径和 library 路径。在 include 的路径中添加 WinPcap 的 include 文件夹路径，在 library 中添加 WinPcap 的 lib 文件夹路径。

（3）第三步，安装 Libnet。

在 Windows 平台下安装 Libnet，必须使用 Visual C++ 开发环境，可以使用版本 6.0。其

步骤如下：

首先把下载的 Libnet 开发包进行解压缩，然后生成一个文件夹 libnet-1.1.2.1。把此文件夹所有的文件都拷贝到一个盘符下面，如“F：盘符”。可以看到，在文件夹 libnet-1.1.2.1 中有一个子文件夹 Libnet，在子文件夹 Libnet 中又有很多的子文件夹，如图 5-13 所示。

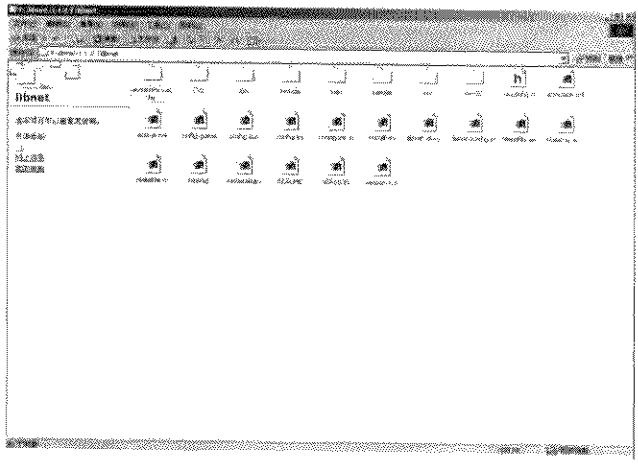

图 5-13　Libnet 开发包的文件夹分布

其中子文件夹 src 中是源代码，文件夹 sample 中是一些示例，文件夹 include 中是所包含的头文件，文件夹 win32 中是在 Windows 平台下安装的一些文件。

打开文件夹 win32，就可以看到一些 Visual C++ 相关的开发文件，如图 5-14 所示。

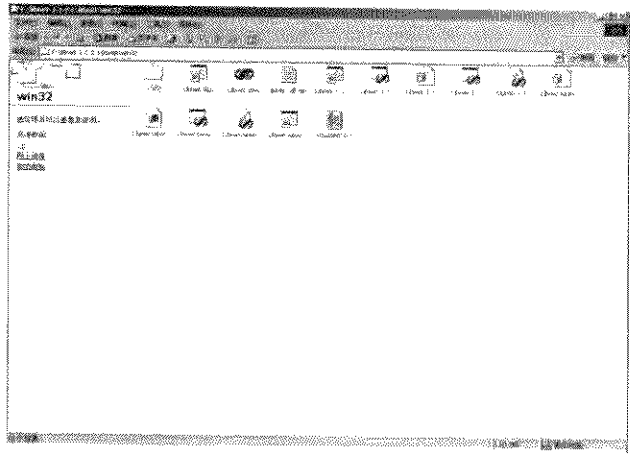

图 5-14　文件夹 win32 中的文件

然后双击文件 libnet.dsw，此时就打开了 Libnet 的 Visual C++ 工作空间，如图 5-15 所示。

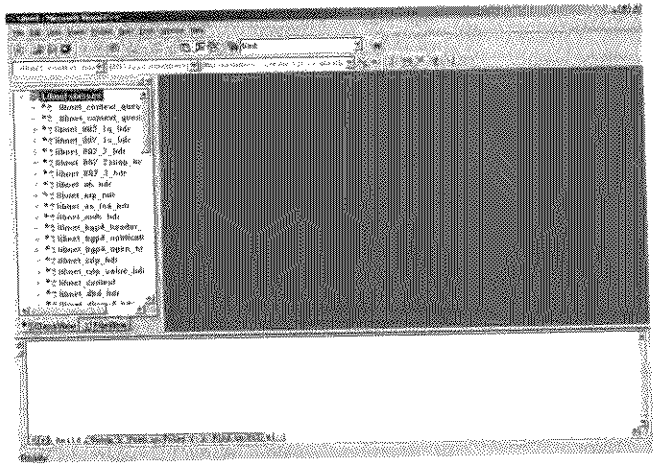

图 5-15　libent 开发包的 Visual C++ 工作空间

此时就可以对 Libnet 进行编译，但是在 Libnet 中用到 Wincrypt API 函数，所以必须在库中加上库文件 advapi32.lib。具体步骤是：在 Visual C++ 菜单 Project 的子菜单“Setting...”中进行设置，在 Project Setting 对话框的 link 页面中将 Object/Library Modules 设置为 advapi32.lib，如图 5-16 所示。

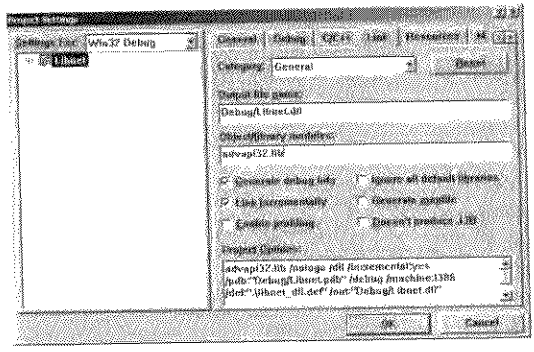

图 5-16　配置 advapi32.lib

然后单击菜单 Build 的子菜单 Build Libnet.dll，此时 Visual C++ 就对 Libnet 开发包进行编译。在输出窗口会出现如下信息：

```
--------------------Configuration: Libnet - Win32 Debug--------------------
Compiling...
```

```
libnet_advanced.c
f:\libnet-1.1.2.1\libnet\src\libnet_advanced.c(102):warning C4018:'==':signed/unsigned mismatch
libnet_asn1.c
libnet_build_802.1q.c
libnet_build_802.1x.c
libnet_build_802.2.c
libnet_build_802.3.c
libnet_build_arp.c
libnet_build_bgp.c
libnet_build_cdp.c
libnet_build_data.c
libnet_build_dhcp.c
libnet_build_dns.c
libnet_build_ethernet.c
libnet_build_fddi.c
libnet_build_gre.c
libnet_build_icmp.c
libnet_build_igmp.c
libnet_build_ip.c
libnet_build_ipsec.c
libuet_build_isl.c
Generating Code...
f:\libnet-1.1.2.1\libnet\src\libnet_build_ip.c(102):warning C4761: integral size mismatch in argument; conversion supplied
f:\libnet-1.1.2.1\libnet\src\libnet_build_ip.c(305):warning C4761: integral size mismatch in argument; conversion supplied
f:\libnet-1.1.2.1\libnet\src\libnet_build_dns.c(87):warning C4761: integral size mismatch in argument; conversion supplied.
f:\libnet-1.1.2.1\libnet\src\libnet_build_cdp.c(74):warning C4761: integral size mismatch in argument; conversion supplied
f:\libnet-1.1.2.1\libnet\src\libnet_build_cdp.c(153):warning C4761: integral size mismatch in argument; conversion supplied
f:\libnet-1.1.2.1\libnet\src\libnet_build_802.1q.c(74):warning C4761: integral size mismatch in argument; conversion supplied
Compiling...
libnet_build_link.c
libnet_build_mpls.c
libnet_build_ntp.c
libnet_build_ospf.c
libnet_build_rip.c
libnet_build_rpc.c
libnet_build_snmp.c
libnet_build_stp.c
libnet_build_tcp.c
libnet_build_token_ring.c
libnet_build_udp.c
libnet_build_vrrp.c
libnet_checksum.c
libnet_cq.c
```

```
libnet_crc.c
libnet_dll.c
libnet_error.c
libnet_if_addr.c
libnet_init.c
libnet_internal.c
Generating Code...
f:\libnet-1.1.2.1\libnet\src\libnet_checksum.c(178) : warning C4761: integral size mismatch in argument; conversion supplied
f:\libnet-1.1.2.1\libnet\src\libnet_checksum.c(196) : warning C4761: integral size mismatch in argument; conversion supplied
f:\libnet-1.1.2.1\libnet\src\libnet_checksum.c(210) : warning C4761: integral size mismatch in argument; conversion supplied
f:\libnet-1.1.2.1\libnet\src\libnet_build_udp.c(71) : warning C4761: integral size mismatch in argument; conversion supplied
f:\libnet-1.1.2.1\libnet\src\libnet_build_tcp.c(155) : warning C4761: integral size mismatch in argument; conversion supplied
f:\libnet-1.1.2.1\libnet\src\libnet_build_tcp.c(349) : warning C4761: integral size mismatch in argument; conversion supplied
f:\libnet-1.1.2.1\libnet\src\libnet_build_tcp.c(353) : warning C4761: integral size mismatch in argument; conversion supplied
f:\libnet-1.1.2.1\libnet\src\libnet_build_ospf.c(75) : warning C4761: integral size mismatch in argument; conversion supplied
f:\libnet-1.1.2.1\libnet\src\libnet_build_ospf.c(405): warning C4761: integral size mismatch in argument; conversion supplied
Compiling...
libnet_link_win32.c
libnet_pblock.c
libnet_port_list.c
f:\libnet-1.1.2.1\libnet\src\libnet_port_list.c(165) : warning C4018: '!=' : signed/unsigned mismatch
libnet_prand.c
libnet_raw.c
f:\libnet-1.1.2.1\libnet\src\libnet_raw.c(45) : warning C4142: benign redefinition of type
libnet_resolve.c
libnet_version.c
libnet_write.c
f:\libnet-1.1.2.1\libnet\src\libnet_write.c(96) : warning C4018: '==' : signed/unsigned mismatch
Generating Code...
Linking...
   Creating library Debug/Libnet.lib and object Debug/Libnet.exp
Libnet.dll - 0 error(s), 19 warning(s)
```

从编译的信息可以看出没有任何错误，证明编译通过。至此可以看到在文件夹 win32 中多了一个文件夹 debug。打开此文件夹，可以看到文件 libnet.dll 和 libnet.lib，如图 5-17 所示。

文件 libnet.dll 是动态链接库文件，它是一个非常重要的文件，是所有 libnet 应用程序在 Windows 环境下运行所必需的，如果没有 libnet.dll 文件，libent 应用程序就无法运行。文件 libnet.dll 中有 Libnet 库函数的导出定义。

可以把文件 libnet.dll 拷贝至 Windows 系统目录下。这样，当使用 Libnet 开发包开发的程序运行时就可以找到动态链接库 libnet.dll 了，不然就会出现以下错误："无法找到动态链接库 libnet.dll 于指定的路径"。

文件 libnet.lib 是库文件，用于开发程序，在后面用开发包 Libnet 写程序时要用到。

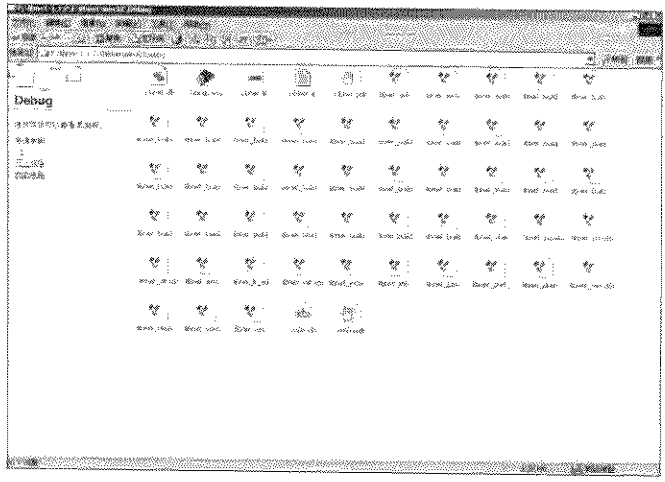

图 5-17　文件夹 debug 信息

5.3　Libnet 数据结构

Libnet 开发包中使用了很多数据结构，主要包含各种常量、各种协议数据结构，以及与 Libnet 本身密切相关的几个核心数据结构。

5.3.1　常量

Libnet 库包含了一些常量，这些常量都代表一定的意义。它们主要在头文件 libnet-macros.h 和 libnet-headers.h 中。下面对主要的常量进行详细介绍。

在头文件 libnet-macros.h 中定义的常量有：

- IN6ADDR_ERROR_INIT
- for_each_context_in_eq
- LIBNET_DONT_RESOLVE

其值为 0，表示不需要对 IP 地址进行域名解析，它用在 Linbet 的名字解析函数中，表示不需要进行 DNS 查询操作，IP 地址用十进制形式。

- LIBNET_ERRBUF_SIZE

其值为 0x100，表示 Libnet 出错信息的长度是 256 字节。

- LIBNET_MAX_PACKET

其值为 0xffff，表示最大的 IP 数据包的长度为 65 535 字节。

- LIBNET_MAXOPTION_SIZE

其值为 0x28，表示 Libnet 的 IP 和 TCP 选项最长为 40 字节。

- LIBNET_OFF

其值为 1，可以用在很多地方。

- LIBNET_ON

其值为 0，可以用在很多地方。

- LIBNET_PR2
- LIBNET_RESOLVE

其值为 1，表示需要对 IP 进行域名解析。它也用在 Libnet 的名字解析函数中，表示需要进行 DNS 查询操作，把 IP 地址变换为名字。

在头文件 libnet-headers.h 中定义的常量有：

- LIBNET_802_1Q_H

其值为 0x12 ，表示协议头 802.1Q 的长度为 18 字节。

- LIBNET_802_1X_H

其值为 0x04，表示协议头 802.1X 的长度为 4 字节。

- LIBNET_802_2_H

其值为 0x03，表示协议头 802.2LLC 的长度为 3 字节。

- LIBNET_802_2SNAP_H

其值为 0x08，表示协议头 802.2LLC/SNAP 的长度为 8 字节。

- LIBNET_802_3_H

其值为 0x0e，表示 802.3 协议头的长度为 14 字节。

- LIBNET_ARP_H

其值为 0x08，表示协议头 ARP 的长度为 8 字节。

- LIBNET_ARP_ETH_IP_H

其值为 0x1c，表示协议头 ARP 加以太网和 IP 的长度为 28 字节。

- LIBNET_BGP4_HEADER_H

其值为 0x13，表示协议头 BGP 的长度为 19 字节。

- LIBNET_BGP4_OPEN_H

其值为 0x0a，表示协议头 BGP 开放长度为 10 字节。

- LIBNET_BGP4_UPDATE_H

其值为 0x04，表示协议头 BGP 开放长度为 4 字节。

- LIBNET_BGP4_NOTIFICATION_H

其值为 0x02，表示协议头 BGPnotif 的长度为 2 字节。

- LIBNET_CDP_H

其值为 0x08，表示协议头 CDP 的长度为 8 字节。

- LIBNET_DHCPV4_H

其值为 0xf0，表示协议头 DHCP 的长度为 240 字节。

- LIBNET_UDP_DNSV4_H

其值为 0x0c，表示协议头 UDP DNS 的长度为 12 字节。

- LIBNET_TCP_DNSV4_H

其值为 0x0e，表示协议头 TCP DNS 的长度为 14 字节。

- LIBNET_ETH_H

其值为 0x0e，表示协议头 Ethernet 的长度为 14 字节。

- LIBNET_FDDI_H

其值为 0x15，表示协议头 FDDI 的长度为 12 字节。

- LIBNET_ICMPV4_H

其值为 0x04，表示协议头 ICMP 的长度为 4 字节。

- LIBNET_ICMPV4_ECHO_H

其值为 0x08，表示协议头 ICMP_ECHO 的长度为 8 字节。

- LIBNET_ICMPV4_MASK_H

其值为 0x0c，表示协议头 ICMP_MASK 的长度为 12 字节。

- LIBNET_ICMPV4_UNREACH_H

其值为 0x08，表示协议头 ICMP_UNREACH 的长度为 8 字节。

- LIBNET_ICMPV4_TIMXCEED_H

其值为 0x08，表示协议头 ICMP_TIMXCEED 的长度为 8 字节。

- LIBNET_ICMPV4_REDIRECT_H

其值为 0x08，表示协议头 ICMP_REDIRECT 的长度为 8 字节。

- LIBNET_ICMPV4_TS_H

其值为 0x14，表示协议头 ICMP_TIMESTAMP 的长度为 10 字节。

- LIBNET_ICMPV6_H

其值为 0x08，表示协议头 ICMP6 的长度为 8 字节。

- LIBNET_IGMP_H

其值为 0x08，表示协议头 IGMP 的长度为 8 字节。

- LIBNET_IPV4_H

其值为 0x14，表示协议头 IPv4 的长度为 20 字节。

- LIBNET_IPV6_H

其值为 0x28，表示协议头 IPv6 的长度为 40 字节。

- LIBNET_IPV6_FRAG_H

其值为 0x08，表示协议头 IPv6 frag 的长度为 8 字节。

- LIBNET_IPV6_ROUTING_H

其值为 0x04，表示协议头 IPv6 frag 的长度为 4 字节。

- LIBNET_IPV6_DESTOPTS_H

其值为 0x02，表示协议头 IPv6 dest opts 长度为 2 字节。

- LIBNET_IPV6_HBHOPTS_H

其值为 0x02，表示协议头 IPv6 hop/hop 长度为 2 字节。

- LIBNET_IPSEC_ESP_HDR_H

其值为 0x0c，表示协议头 IPSEC ESP 长度为 12 字节。

- LIBNET_IPSEC_ESP_FTR_H

其值为 0x02，表示协议头 IPSEC ESP footer 长度为 2 字节。

- LIBNET_IPSEC_AH_H

其值为 0x10，表示协议头 IPSEC AH 的长度为 16 字节。

- LIBNET_ISL_H

其值为 0x1a，表示协议头 ISL 的长度为 26 字节。

- LIBNET_GRE_H

其值为 0x04，表示协议头 GRE 的长度为 4 字节。

- LIBNET_GRE_SRE_H

其值为 0x04，表示协议头 GRE SRE 的长度为 4 字节。

- LIBNET_MPLS_H

其值为 0x04，表示协议头 MPLS 的长度为 4 字节。

- LIBNET_OSPF_H

其值为 0x10，表示协议头 OSPF 的长度为 16 字节。

- LIBNET_OSPF_HELLO_H

其值为 0x18，表示协议头 OSPF hello 的长度为 24 字节。

- LIBNET_OSPF_DBD_H

其值为 0x08，表示协议头 OSPF DBD 的长度为 8 字节。

- #define LIBNET_OSPF_LSR_H

其值为 0x0c，表示协议头 OSPF LSR 的长度为 12 字节。

- LIBNET_OSPF_LSU_H

其值为 0x04，表示协议头 OSPF LSU 的长度为 4 字节。

- LIBNET_OSPF_LSA_H

其值为 0x14，表示协议头 OSPF LSA 的长度为 20 字节。

- LIBNET_OSPF_AUTH_H

其值为 0x08，表示协议头 OSPF AUTH 的长度为 8 字节。

- LIBNET_OSPF_CKSUM

其值为 0x10，表示协议头 OSPF CKSUM 的长度为 16 字节。

- LIBNET_OSPF_LS_RTR_H

其值为 0x10，表示协议头 OSPF LS RTR 的长度为 16 字节。

- LIBNET_OSPF_LS_NET_H

其值为 0x08，表示协议头 OSPF LS NET 的长度为 8 字节。

- LIBNET_OSPF_LS_SUM_H

其值为 0x0c，表示协议头 OSPF LS SUM 的长度为 12 字节。

- LIBNET_OSPF_LS_AS_EXT_H

其值为 0x10，表示协议头 OSPF LS AS 的长度为 16 字节。

- LIBNET_NTP_H

其值为 0x30，表示协议头 NTP 的长度为 48 字节。

- LIBNET_RIP_H

其值为 0x18，表示协议头 RIP 的长度为 24 字节。

- LIBNET_RPC_CALL_H

其值为 0x28，表示协议头 RPC 长度为 40 字节。

- LIBNET_RPC_CALL_TCP_H

其值为 0x2c，表示协议头 RPC 的长度为 44 字节。

- LIBNET_SEBEK_H

其值为 0x30，表示协议头 SEBEK 的长度为 48 字节。

- LIBNET_STP_CONF_H

其值为 0x23，表示协议头 STP conf 的长度为 35 字节。

- LIBNET_STP_TCN_H

其值为 0x04，表示协议头 STP tcn 的长度为 4 字节。

- LIBNET_TOKEN_RING_H

其值为 0x16，表示协议头 Token Ring 的长度为 22 字节。

- LIBNET_TCP_H

其值为 0x14，表示协议头 TCP 的长度为 20 字节。

- LIBNET_UDP_H

其值为 0x08，表示协议头 UDP 长度为 8 字节。

- LIBNET_VRRP_H

其值为 0x08，表示协议头 VRRP 的长度为 8 字节。

5.3.2 协议头的数据结构

由上面可以看到，Libnet 支持的协议种类非常多。其数据结构的定义全部在文件 libnet-headers.h 中有详细说明。

在构造不同的协议时，必须对其协议格式了如指掌。协议头的数据结构定义了每个协议的协议格式，对它们的熟悉和掌握对于构造各种网络数据包是至关重要的。

下面列出 Libnet 支持的所有协议头（34 个）的数据结构。

（1）虚拟局域网络协议 IEEE 802.1Q。

```
struct libnet_802_1q_hdr
{
    u_int8_t vlan_dhost[ETHER_ADDR_LEN];                /*目的以太网地址*/
    u_int8_t vlan_shost[ETHER_ADDR_LEN];                /*源以太网地址 */
    u_int16_t vlan_tpi;                                 /*标志协议 ID */
    u_int16_t vlan_priority_c_vid;                      /*优先级 ID */
    #define LIBNET_802_1Q_PRIMASK   0x0007              /*优先级掩码 */
    #define LIBNET_802_1Q_CFIMASK   0x0001              /*CFI 掩码 */
    #define LIBNET_802_1Q_VIDMASK   0x0fff              /*vid 掩码 */
    u_int16_t vlan_len;                                 /*长度 */
};
```

（2）可扩展鉴定协议 IEEE 802.1X EAP。

```
struct libnet_802_1x_hdr
{
    u_int8_t dot1x_version;                             /*协议版本 */
    u_int8_t dot1x_type;                                /*帧类型 */
    #define LIBNET_802_1X_PACKET     0x00               /* 802.1x 包 */
    #define LIBNET_802_1X_START      0x01               /* 802.1x s 开始*/
    #define LIBNET_802_1X_LOGOFF     0x02               /* 802.1x 偏移 */
    #define LIBNET_802_1X_KEY        0x03               /* 802.1x 关键字 */
    #define LIBNET_802_1X_ENCASFAL   0x04               /*802.1x encasfal */
    u_int16_t dot1x_length;                             /*整个帧的长度*/
}
```

（3）链路层控制协议 IEEE 802.2 LLC。

```
struct libnet_802_2_hdr
{
    u_int8_t llc_dsap;              /* 目的服务连接点 */
    u_int8_t llc_ssap;              /* 源服务连接点 */
    #define LIBNET_SAP_STP              0x42
    #define LIBNET_SAP_SNAP             0xaa
    u_int8_t llc_control;           /* 控制区域 */
};
```

（4）子网附加点协议（SubNetwork Attachment Point ）IEEE 802.2 LLC/SNAP。

```
struct libnet_802_2snap_hdr
{
    u_int8_t snap_dsap;             /* 源服务连接点 */
    u_int8_t snap_ssap;             /* 源服务连接点 */
    u_int8_t snap_control;          /* 控制区域 */
    u_int8_t snap_oui[3];           /* OUI */
    u_int16_t snap_type;            /* 类型 */
};
```

（5）IEEE 以太网协议（IEEE Ethernet）802.3。

```
struct libnet_802_3_hdr
{
    u_int8_t  _802_3_dhost[ETHER_ADDR_LEN]     ;/* 目的以太网地址 */
    u_int8_t  _802_3_shost[ETHER_ADDR_LEN]     ;/* 源以太网地址 */
    u_int16_t _802_3_len;                      /* 包类型 ID */
};
```

（6）地址转换协议（ARP，Address Resolution Protocol）。

```
struct libnet_arp_hdr
{
    u_int16_t ar_hrd;                   /* 硬件地址形式 */
    #define ARPHRD_NETROM       0       /* KA9Q: NET/ROM 伪码 */
    #define ARPHRD_ETHER        1       /* 以太网 10Mbps */
    #define ARPHRD_EETHER       2       /* 扩展以太网 */
    #define ARPHRD_AX25         3       /* AX.25 Level 2 */
    #define ARPHRD_PRONET       4       /* PROnet 令牌环 */
    #define ARPHRD_CHAOS        5       /* Chaosnet 协议 */
    #define ARPHRD_IEEE802      6       /* IEEE 802.2 Ethernet/TR/TB 协议*/
    #define ARPHRD_ARCNET       7       /* ARCnet 协议*/
    #define ARPHRD_APPLETLK     8       /* APPLEtalk 协议*/
    #define ARPHRD_LANSTAR      9       /* Lanstar 协议*/
    #define ARPHRD_DLCI         15      /* 帧中继 DLCI 协议 */
    #define ARPHRD_ATM          19      /* ATM 协议*/
    #define ARPHRD_METRICOM     23      /* Metricom STRIP 协议 */
```

```
    #define ARPHRD_IPSEC        31        /* IPsec 通道*/
    u_int16_t ar_pro;                     /* 协议地址类型 */
    u_int8_t  ar_hln;                     /* 硬件地址长度 */
    u_int8_t  ar_pln;                     /* 协议地址长度 */
    u_int16_t ar_op;                      /* 操作类型 */
    #define ARPOP_REQUEST       1         /* 地址解析的请求*/
    #define ARPOP_REPLY         2         /* 请求的应答 */
    #define ARPOP_REVREQUEST    3         /* 获取给定的硬件地址的协议地址 */
    #define ARPOP_REVREPLY      4         /* 获取协议地址的应答 */
    #define ARPOP_INVREQUEST    8         /*对等鉴别的请求 */
    #define ARPOP_INVREPLY      9         /* 对等鉴别的应答 */
};
```

（7）边界网关协议 BGP4（Border Gateway Protocol 4）。

```
struct libnet_bgp4_header_hdr
{
    #define LIBNET_BGP4_MARKER_SIZE  16
    u_int8_t marker[LIBNET_BGP4_MARKER_SIZE];
    u_int16_t len;
    u_int8_t type;
    #define LIBNET_BGP4_OPEN              1
    #define LIBNET_BGP4_UPDATE            2
    #define LIBNET_BGP4_NOTIFICATION      3
    #define LIBNET_BGP4_KEEPALIVE         4
};
```

（8）边界网关协议 BGP4 open。

```
struct libnet_bgp4_open_hdr
{
    u_int8_t version;
    u_int16_t src_as;
    u_int16_t hold_time;
    u_int32_t bgp_id;
    u_int8_t opt_len;
};
```

（9）边界网关协议 BGP4 notification。

```
struct libnet_bgp4_notification_hdr
{
    #define LIBNET_BGP4_MESSAGE_HEADER_ERROR  1
    #define LIBNET_BGP4_OPEN_MESSAGE_ERROR    2
    #define LIBNET_BGP4_UPDATE_MESSAGE_ERROR  3
    #define LIBNET_BGP4_HOLD_TIMER_EXPIRED    4
    #define LIBNET_BGP4_FINITE_STATE__ERROR   5
    #define LIBNET_BGP4_CEASE                 6
```

```
    u_int8_t err_code;
    /* 消息头错误子代码 */
    #define LIBNET_BGP4_CONNECTION_NOT_SYNCHRONIZED     1
    #define LIBNET_BGP4_BAD_MESSAGE_LENGTH              2
    #define LIBNET_BGP4_BAD_MESSAGE_TYPE                3
    /* OPEN 消息错误子代码 */
    #define LIBNET_BGP4_UNSUPPORTED_VERSION_NUMBER      1
    #define LIBNET_BGP4_BAD_PEER_AS                     2
    #define LIBNET_BGP4_BAD_BGP_IDENTIFIER              3
    #define LIBNET_BGP4_UNSUPPORTED_OPTIONAL_PARAMETER  4
    #define LIBNET_BGP4_AUTHENTICATION_FAILURE          5
    #define LIBNET_BGP4_UNACCEPTABLE_HOLD_TIME          6
    /* UPDATE 消息错误子代码 */
    #define LIBNET_BGP4_MALFORMED_ATTRIBUTE_LIST
    #define LIBNET_BGP4_UNRECOGNIZED_WELL_KNOWN_ATTRIBUTE
    #define LIBNET_BGP4_MISSING_WELL_KNOWN_ATTRIBUTE
    #define LIBNET_BGP4_ATTRIBUTE_FLAGS_ERROR
    #define LIBNET_BGP4_ATTRIBUTE_LENGTH_ERROR
    #define LIBNET_BGP4_INVALID_ORIGIN_ATTRIBUTE
    #define LIBNET_BGP4_AS_ROUTING_LOOP
    #define LIBNET_BGP4_INVALID_NEXT_HOP_ATTRIBUTE
    #define LIBNET_BGP4_OPTIONAL_ATTRIBUTE_ERROR
    #define LIBNET_BGP4_INVALID_NETWORK_FIELD
    #define LIBNET_BGP4_MALFORMED_AS_PATH
    u_int8_t err_subcode;
};
```

（10）Cisco 发现协议（CDP，Cisco Discovery Protocol）。

```
#define LIBNET_PROTO_CDP    200
struct libnet_cdp_hdr
{
    u_int8_t cdp_version;                   /* 版本*/
    u_int8_t cdp_ttl;                       /* 时间 */
    u_int16_t cdp_sum;                      /* 校验和*/
    u_int16_t cdp_type;                     /* 类型*/
    #define LIBNET_CDP_DEVID     0x1        /* 设备 ID */
    #define LIBNET_CDP_ADDRESS   0x2        /* 地址 */
    #define LIBNET_CDP_PORTID    0x3        /* 端口 ID */
    #define LIBNET_CDP_CAPABIL   0x4        /*性能 */
    #define LIBNET_CDP_VERSION   0x5        /* 版本 */
    #define LIBNET_CDP_PLATFORM  0x6        /* 平台*/
    #define LIBNET_CDP_IPPREFIX  0x7        /* ip 前缀*/
    u_int16_t cdp_len;                      /* 类型加长度和值的长度 */
    /* CDP 性能 */
```

```
    #define LIBNET_CDP_CAP_L3R  0x01          /* 执行第 3 层路由 */
    #define LIBNET_CDP_CAP_L2B  0x02          /* 执行第 2 层透明网桥 */
    #define LIBNET_CDP_CAP_L2SRB 0x04         /* 执行第 2 层源路由 */
    #define LIBNET_CDP_CAP_L2S  0x08          /* 执行第 2 层交换 */
    #define LIBNET_CDP_CAP_SR   0x10          /* 发送和接收数据包 */
    #define LIBNET_CDP_CAP_NOI  0x20          /* 在非路由端口不转发 IGMP */
    #define LIBNET_CDP_CAP_L1F  0x40          /* 提供第 1 层功能 */
};
struct libnet_cdp_value_hdr
{
    u_int16_t cdp_type;
    u_int16_t cdp_len;
};
```

（11）动态主机配置协议（DHCP，Dynamic Host Configuration Protocol）。

```
struct libnet_dhcpv4_hdr
{
    u_int8_t dhcp_opcode;                 /* 操作码*/
    #define LIBNET_DHCP_REQUEST 0x1
    #define LIBNET_DHCP_REPLY   0x2
    u_int8_t dhcp_htype;                  /* 硬件地址类型*/
    u_int8_t dhcp_hlen;                   /* 硬件地址长度*/
    u_int8_t dhcp_hopcount;               /* 代理服务器使用 */
    u_int32_t dhcp_xid;                   /* 事务 ID */
    u_int16_t dhcp_secs;                  /* 第 2 次重启次数*/
    u_int16_t dhcp_flags;                 /* DHCP 标志 */
    u_int32_t dhcp_cip;                   /* 客户端 IP */
    u_int32_t dhcp_yip;                   /* 自己的 IP */
    u_int32_t dhcp_sip;                   /* 服务器的 IP */
    u_int32_t dhcp_gip;                   /* 网关 IP */
    u_int8_t dhcp_chaddr[16];             /* 客户端硬件地址 */
    u_int8_t dhcp_sname[64];              /* 服务器名字 */
    u_int8_t dhcp_file[128];              /* 启动文件名字*/
    u_int32_t dhcp_magic;                 /* BOOTP 头 */
    #define DHCP_MAGIC                    0x63825363
    #define LIBNET_BOOTP_MIN_LEN          0x12c
    #define LIBNET_DHCP_PAD               0x00
    #define LIBNET_DHCP_SUBNETMASK        0x01
    #define LIBNET_DHCP_TIMEOFFSET        0x02
    #define LIBNET_DHCP_ROUTER            0x03
    #define LIBNET_DHCP_TIMESERVER        0x04
    #define LIBNET_DHCP_NAMESERVER        0x05
    #define LIBNET_DHCP_DNS               0x06
    #define LIBNET_DHCP_LOGSERV           0x07
```

```
#define LIBNET_DHCP_COOKIESERV          0x08
#define LIBNET_DHCP_LPRSERV             0x09
#define LIBNET_DHCP_IMPSERV             0x0a
#define LIBNET_DHCP_RESSERV             0x0b
#define LIBNET_DHCP_HOSTNAME            0x0c
#define LIBNET_DHCP_BOOTFILESIZE        0x0d
#define LIBNET_DHCP_DUMPFILE            0x0e
#define LIBNET_DHCP_DOMAINNAME          0x0f
#define LIBNET_DHCP_SWAPSERV            0x10
#define LIBNET_DHCP_ROOTPATH            0x11
#define LIBNET_DHCP_EXTENPATH           0x12
#define LIBNET_DHCP_IPFORWARD           0x13
#define LIBNET_DHCP_SRCROUTE            0x14
#define LIBNET_DHCP_POLICYFILTER        0x15
#define LIBNET_DHCP_MAXASMSIZE          0x16
#define LIBNET_DHCP_IPTTL               0x17
#define LIBNET_DHCP_MTUTIMEOUT          0x18
#define LIBNET_DHCP_MTUTABLE            0x19
#define LIBNET_DHCP_MTUSIZE             0x1a
#define LIBNET_DHCP_LOCALSUBNETS        0x1b
#define LIBNET_DHCP_BROADCASTADDR       0x1c
#define LIBNET_DHCP_DOMASKDISCOV        0x1d
#define LIBNET_DHCP_MASKSUPPLY          0x1e
#define LIBNET_DHCP_DOROUTEDISC         0x1f
#define LIBNET_DHCP_ROUTERSOLICIT       0x20
#define LIBNET_DHCP_STATICROUTE         0x21
#define LIBNET_DHCP_TRAILERENCAP        0x22
#define LIBNET_DHCP_ARPTIMEOUT          0x23
#define LIBNET_DHCP_ETHERENCAP          0x24
#define LIBNET_DHCP_TCPTTL              0x25
#define LIBNET_DHCP_TCPKEEPALIVE        0x26
#define LIBNET_DHCP_TCPALIVEGARBAGE     0x27
#define LIBNET_DHCP_NISDOMAIN           0x28
#define LIBNET_DHCP_NISSERVERS          0x29
#define LIBNET_DHCP_NISTIMESERV         0x2a
#define LIBNET_DHCP_VENDSPECIFIC        0x2b
#define LIBNET_DHCP_NBNS                0x2c
#define LIBNET_DHCP_NBDD                0x2d
#define LIBNET_DHCP_NBTCPIP             0x2e
#define LIBNET_DHCP_NBTCPSCOPE          0x2f
#define LIBNET_DHCP_XFONT               0x30
#define LIBNET_DHCP_XDISPLAYMGR         0x31
#define LIBNET_DHCP_DISCOVERADDR        0x32
```

```
    #define LIBNET_DHCP_LEASETIME          0x33
    #define LIBNET_DHCP_OPTIONOVERLOAD     0x34
    #define LIBNET_DHCP_MESSAGETYPE        0x35
    #define LIBNET_DHCP_SERVIDENT          0x36
    #define LIBNET_DHCP_PARAMREQUEST       0x37
    #define LIBNET_DHCP_MESSAGE            0x38
    #define LIBNET_DHCP_MAXMSGSIZE         0x39
    #define LIBNET_DHCP_RENEWTIME          0x3a
    #define LIBNET_DHCP_REBINDTIME         0x3b
    #define LIBNET_DHCP_CLASSSID           0x3c
    #define LIBNET_DHCP_CLIENTID           0x3d
    #define LIBNET_DHCP_NISPLUSDOMAIN      0x40
    #define LIBNET_DHCP_NISPLUSSERVERS     0x41
    #define LIBNET_DHCP_MOBILEIPAGENT      0x44
    #define LIBNET_DHCP_SMTPSERVER         0x45
    #define LIBNET_DHCP_POP3SERVER         0x46
    #define LIBNET_DHCP_NNTPSERVER         0x47
    #define LIBNET_DHCP_WWWSERVER          0x48
    #define LIBNET_DHCP_FINGERSERVER       0x49
    #define LIBNET_DHCP_IRCSERVER          0x4a
    #define LIBNET_DHCP_STSERVER           0x4b
    #define LIBNET_DHCP_STDASERVER         0x4c
    #define LIBNET_DHCP_END                0xff
    #define LIBNET_DHCP_MSGDISCOVER        0x01
    #define LIBNET_DHCP_MSGOFFER           0x02
    #define LIBNET_DHCP_MSGREQUEST         0x03
    #define LIBNET_DHCP_MSGDECLINE         0x04
    #define LIBNET_DHCP_MSGACK             0x05
    #define LIBNET_DHCP_MSGNACK            0x06
    #define LIBNET_DHCP_MSGRELEASE         0x07
    #define LIBNET_DHCP_MSGINFORM          0x08
};
```

（12）域名系统（Domain Name System）DNSv4。

```
#define LIBNET_DNS_H LIBNET_UDP_DNSV4_H
struct libnet_dnsv4_hdr
{
    u_int16_t h_len;            /* 数据包长度*/
    u_int16_t id;               /* DNS 包 ID */
    u_int16_t flags;            /* DNS 标志 */
    u_int16_t num_q;            /* 问题个数 */
    u_int16_t num_answ_rr;      /* 回答资源记录个数 */
    u_int16_t num_auth_rr;      /* 确认资源记录个数 */
    u_int16_t num_addi_rr;      /* 附加资源记录个数 */
```

```
};
```

（13）以太网 Ethernet II

```
struct libnet_ethernet_hdr
{
    u_int8_t  ether_dhost[ETHER_ADDR_LEN];      /* 目的以太网地址 */
    u_int8_t  ether_shost[ETHER_ADDR_LEN];      /* 源以太网地址 */
    u_int16_t ether_type;                       /* 协议*/
};
    #ifndef ETHERTYPE_PUP
    #define ETHERTYPE_PUP  0x0200               /* PUP 协议*/
    #endif
    #ifndef ETHERTYPE_IP
    #define ETHERTYPE_IP   0x0800               /* IP 协议*/
    #endif
    #ifndef ETHERTYPE_ARP
    #define ETHERTYPE_ARP   0x0806              /* 地址解析协议 ARP */
    #endif
    #ifndef ETHERTYPE_REVARP
    #define ETHERTYPE_REVARP   0x8035           /* RARP 协议 */
    #endif
    #ifndef ETHERTYPE_VLAN
    #define ETHERTYPE_VLAN  0x8100              /* IEEE 802.1Q VLAN 协议 */
    #endif
    #ifndef ETHERTYPE_EAP
    #define ETHERTYPE_EAP   0x888e              /* IEEE 802.1X EAP 鉴定协议 */
    #endif
    #ifndef ETHERTYPE_MPLS
    #define ETHERTYPE_MPLS  0x8847              /* MPLS 协议*/
    #endif
    #ifndef ETHERTYPE_LOOPBACK
    #define ETHERTYPE_LOOPBACK  0x9000          /*回环 */
    #endif
    struct libnet_ether_addr
    {
        u_int8_t  ether_addr_octet[6];              /* 以太网地址 */
};
```

（14）光纤分布数据接口（FDDI，Fiber Distributed Data Interface）。

```
struct libnet_fddi_hdr
{
    u_int8_t  fddi_frame_control;                       /* 优先级 */
    #define LIBNET_FDDI_LLC_FRAME  0x10
    #define LIBNET_FDDI_48BIT_ADDR 0x40
    #define LIBNET_FDDI_FC_REQD    LIBNET_FDDI_LLC_FRAME | LIBNET_FDDI_48BIT_ADDR
```

```
    u_int8_t  fddi_dhost[FDDI_ADDR_LEN];                    /* 目的 FDDI 地址 */
    u_int8_t  fddi_shost[FDDI_ADDR_LEN];                    /* 源 FDDI 地址 */
    u_int8_t  fddi_llc_dsap;                                /* DSAP */
    u_int8_t  fddi_llc_ssap;                                /* SSAP */
    u_int8_t  fddi_llc_control_field;                       /* 优先级 */
    u_int8_t  fddi_llc_org_code[LIBNET_ORG_CODE_SIZE];      /* 组织代码 */
    u_int8_t  fddi_type;                                    /* 协议类型 */
    u_int8_t  fddi_type1;                                   /* 协议类型 */
    #define FDDI_TYPE_IP   0x0800                           /* IP 协议*/
    #define FDDI_TYPE_ARP   0x0806                          /* ARP 协议 */
    #define FDDI_TYPE_REVARP  0x8035                        /* RARP 协议 */
};
struct libnet_fddi_addr
{
   u_int8_t  fddi_addr_octet[6];                            /* FDDI 地址*/
};
```

（15）通用路由封装（GRE，Generic Routing Encapsulation）协议。

```
struct libnet_gre_hdr
{
    u_int16_t flags_ver;              /*版本*/
    #define GRE_CSUM                  0x8000
    #define GRE_ROUTING               0x4000
    #define GRE_KEY                   0x2000
    #define GRE_SEQ                   0x1000
    #define GRE_STRICT                0x0800
    #define GRE_REC                   0x0700
    #define GRE_ACK                   0x0080
    #define GRE_FLAGS_MASK            0x00F8
    #define GRE_VERSION_MASK          0x0007
    #define GRE_VERSION_0             0x0000
    #define GRE_VERSION_1             0x0001
    u_int16_t type;                   /*类型*/
    #define GRE_SNA                   0x0004
    #define GRE_OSI_NETWORK_LAYER     0x00FE
    #define GRE_PUP                   0x0200
    #define GRE_XNS                   0x0600
    #define GRE_IP                    0x0800
    #define GRE_CHAOS                 0x0804
    #define GRE_RFC_826_ARP           0x0806
    #define GRE_FRAME_RELAY_ARP       0x0808
    #define GRE_VINES                 0x0BAD
    #define GRE_VINES_ECHO            0x0BAE
    #define GRE_VINES_LOOPBACK        0x0BAF
```

```
    #define GRE_DECNET                          0x6003
    #define GRE_TRANSPARENT_ETHERNET_BRIDGING 0x6558
    #define GRE_RAW_FRAME_RELAY                 0x6559
    #define GRE_APOLLO_DOMAIN                   0x8019
    #define GRE_ETHERTALK                       0x809B
    #define GRE_NOVELL_IPX                      0x8137
    #define GRE_RFC_1144_TCP_IP_COMPRESSION     0x876B
    #define GRE_IP_AUTONOMOUS_SYSTEMS           0x876C
    #define GRE_SECURE_DATA                     0x876D
    #define GRE_PPP                             0x880b
    union {
            struct {
                u_int16_t sum;
                u_int16_t offset;
                u_int32_t key;
                u_int32_t seq;
            } _gre;
            struct {
                u_int16_t payload_s;
                u_int16_t callID;
                u_int32_t seq;
                u_int32_t ack;
            } _egre;
    }_data;
    #define gre_sum _data._gre.sum
    #define gre_offset _data._gre.offset
    #define gre_key _data._gre.key
    #define gre_seq _data._gre.seq
    #define egre_payload_s _data._egre.payload_s
    #define egre_callID _data._egre.callID
    #define egre_seq _data._egre.seq
    #define egre_ack _data._egre.ack
};
#ifndef IPPROTO_GRE
#define IPPROTO_GRE 47
#endif
struct libnet_gre_sre_hdr
{
    u_int16_t af;  /*地址族 */
    u_int8_t sre_offset;
    u_int8_t sre_length;
    u_int8_t *routing;
};
```

（16）IPv4 协议（Internet Protocol，version 4）。

```
struct libnet_ipv4_hdr
{
    #if (LIBNET_LIL_ENDIAN)
    u_int8_t ip_hl:4,           /* 首部长度 */
             ip_v:4;            /* 版本 */
    #endif
    #if (LIBNET_BIG_ENDIAN)
    u_int8_t ip_v:4,            /* 版本*/
             ip_hl:4;           /* 首部长度 */
    #endif
    u_int8_t ip_tos;            /* 服务类型 */
    #ifndef IPTOS_LOWDELAY
    #define IPTOS_LOWDELAY      0x10
    #endif
    #ifndef IPTOS_THROUGHPUT
    #define IPTOS_THROUGHPUT    0x08
    #endif
    #ifndef IPTOS_RELIABILITY
    #define IPTOS_RELIABILITY   0x04
    #endif
    #ifndef IPTOS_LOWCOST
    #define IPTOS_LOWCOST       0x02
    #endif
    u_int16_t ip_len;           /* 总长度 */
    u_int16_t ip_id;            /* 标志 */
    u_int16_t ip_off;
    #ifndef IP_RF
    #define IP_RF 0x8000        /*保留标志 */
    #endif
    #ifndef IP_DF
    #define IP_DF 0x4000
    #endif
    #ifndef IP_MF
    #define IP_MF 0x2000
    #endif
    #ifndef IP_OFFMASK
    #define IP_OFFMASK 0x1fff
    #endif
    u_int8_t ip_ttl;                    /* 生存时间 */
    u_int8_t ip_p;                      /* 协议类型 */
    u_int16_t ip_sum;                   /* 校验和 */
    struct in_addr ip_src, ip_dst;      /* 源地址和目的地址 */
```

```
};
/*IP选项*/
#ifndef IPOPT_EOL
#define IPOPT_EOL       0               /* 选项列表结尾 */
#endif
#ifndef IPOPT_NOP
#define IPOPT_NOP       1               /* 不操作 */
#endif
#ifndef IPOPT_RR
#define IPOPT_RR        7               /* 记录数据包路由 */
#endif
#ifndef IPOPT_TS
#define IPOPT_TS        68              /* 时间戳*/
#endif
#ifndef IPOPT_SECURITY
#define IPOPT_SECURITY  130             /* 提供 s,c,h,tcc */
#endif
#ifndef IPOPT_LSRR
#define IPOPT_LSRR      131             /* 自由源路由 */
#endif
#ifndef IPOPT_SATID
#define IPOPT_SATID     136
#endif
#ifndef IPOPT_SSRR
#define IPOPT_SSRR      137             /* 严格源路由*/
#endif
```

（17）IPv6 协议（Internet Protocol, version 6）。

```
struct libnet_in6_addr
{
    union
    {
      u_int8_t   __u6_addr8[16];
      u_int16_t  __u6_addr16[8];
      u_int32_t  __u6_addr32[4];
    } __u6_addr;                        /* 128 位的 IP6 地址 */
};
#define libnet_s6_addr __u6_addr.__u6_addr8
struct libnet_ipv6_hdr
{
    u_int8_t ip_flags[4];               /* 版本，类型，流标记 */
    u_int16_t ip_len;                   /* 总长度*/
    u_int8_t ip_nh;                     /* 下一个头 */
    u_int8_t ip_hl;                     /* 跳跃限制*/
```

```
    struct libnet_in6_addr ip_src, ip_dst;     /* 源和目的地址 */
};
#define LIBNET_IPV6_NH_FRAGMENT 44
struct libnet_ipv6_frag_hdr
{
    u_int8_t ip_nh;                    /* 下一个头*/
    u_int8_t ip_reserved;              /* 保留 */
    u_int16_t ip_frag;                 /* 碎片填充 */
    u_int32_t ip_id;                   /* 标志*/
};
#define LIBNET_IPV6_NH_ROUTING  43
struct libnet_ipv6_routing_hdr
{
    u_int8_t ip_nh;                    /* 下一个头 */
    u_int8_t ip_len;                   /* 长度*/
    u_int8_t ip_rtype;                 /* 路由类型 */
    u_int8_t ip_segments;              /* 段 */
};
/*IPv6 目的选项*/
#define LIBNET_IPV6_NH_DESTOPTS 60
struct libnet_ipv6_destopts_hdr
{
    u_int8_t ip_nh;                    /* 下一个头*/
    u_int8_t ip_len;                   /*长度*/
};
/*IPv6 跳跃选项*/
#define LIBNET_IPV6_NH_HBH      0
struct libnet_ipv6_hbhopts_hdr
{
    u_int8_t ip_nh;                    /* 下一个头 */
    u_int8_t ip_len;                   /* 长度*/
};
```

（18）ICMP6 协议（Internet Control Message Protocol v6）。

```
#ifndef IPPROTO_ICMP6
#define IPPROTO_ICMP6   0x3a
#endif
struct libnet_icmpv6_hdr
{
    u_int8_t icmp_type;                /* ICMP 类型*/
    #ifndef ICMP6_ECHO
    #define ICMP6_ECHO          128
    #endif
    #ifndef ICMP6_ECHOREPLY
```

```
    #define ICMP6_ECHOREPLY     129
    #endif
    #ifndef ICMP6_UNREACH
    #define ICMP6_UNREACH        1
    #endif
    #ifndef ICMP6_PKTTOOBIG
    #define ICMP6_PKTTOOBIG      2
    #endif
    #ifndef ICMP6_TIMXCEED
    #define ICMP6_TIMXCEED       3
    #endif
    #ifndef ICMP6_PARAMPROB
    #define ICMP6_PARAMPROB      4
    #endif
    u_int8_t icmp_code;          /* ICMP 代码*/
    u_int16_t icmp_sum;          /* ICMP 校验和*/
    u_int16_t id;                /* ICMP 标志 */
    u_int16_t seq;               /* ICMP 序列号 */
};
```

（19）ICMP 协议（Internet Control Message Protocol）。

```
struct libnet_icmpv4_hdr
{
    u_int8_t icmp_type;                     /* ICMP 类型 */
    #ifndef    ICMP_ECHOREPLY
    #define    ICMP_ECHOREPLY          0    /*回显应答*/
    #endif
    #ifndef    ICMP_UNREACH
    #define    ICMP_UNREACH            3    /*目的不可达*/
    #endif
    #ifndef    ICMP_SOURCEQUENCH
    #define    ICMP_SOURCEQUENCH       4    /*源端被关闭*/
    #endif
    #ifndef    ICMP_REDIRECT
    #define    ICMP_REDIRECT           5    /*重定向*/
    #endif
    #ifndef    ICMP_ECHO
    #define    ICMP_ECHO               8    /*请求回显*/
    #endif
    #ifndef    ICMP_ROUTERADVERT
    #define    ICMP_ROUTERADVERT       9    /*路由器通告*/
    #endif
    #ifndef    ICMP_ROUTERSOLICIT
    #define    ICMP_ROUTERSOLICIT     10    /*路由器请求*/
```

```
#endif
#ifndef    ICMP_TIMXCEED
#define    ICMP_TIMXCEED          11   /*超时*/
#endif
#ifndef    ICMP_PARAMPROB
#define    ICMP_PARAMPROB         12   /*参数问题*/
#endif
#ifndef    ICMP_TSTAMP
#define    ICMP_TSTAMP            13   /*时间戳请求*/
#endif
#ifndef    ICMP_TSTAMPREPLY
#define    ICMP_TSTAMPREPLY       14   /*时间戳应答*/
#endif
#ifndef    ICMP_IREQ
#define    ICMP_IREQ              15   /*信息请求*/
#endif
#ifndef    ICMP_IREQREPLY
#define    ICMP_IREQREPLY         16   /*信息应答*/
#endif
#ifndef    ICMP_MASKREQ
#define    ICMP_MASKREQ           17   /*地址掩码请求*/
#endif
#ifndef    ICMP_MASKREPLY
#define    ICMP_MASKREPLY         18   /*地址掩码应答*/
#endif
u_int8_t icmp_code;                     /* ICMP 代码 */
#ifndef    ICMP_UNREACH_NET
#define    ICMP_UNREACH_NET            0    /*网络不可达*/
#endif
#ifndef    ICMP_UNREACH_HOST
#define    ICMP_UNREACH_HOST           1    /*主机不可达*/
#endif
#ifndet    ICMP_UNREACH_PROTOCOL
#define    ICMP_UNREACH_PROTOCOL       2    /*协议不可达*/
#endif
#ifndef    ICMP_UNREACH_PORT
#define    ICMP_UNREACH_PORT           3    /*端口不可达*/
#endif
#ifndef    ICMP_UNREACH_NEEDFRAG
#define    ICMP_UNREACH_NEEDFRAG       4    /*需要进行分片*/
#endif
#ifndef    ICMP_UNREACH_SRCFATL
#define    ICMP_UNREACH_SRCFAIL        5    /*源站选路失败*/
```

```
#endif
#ifndef    ICMP_UNREACH_NET_UNKNOWN
#define    ICMP_UNREACH_NET_UNKNOWN        6    /*目的网络不可达*/
#endif
#ifndef    ICMP_UNREACH_HOST_UNKNOWN
#define    ICMP_UNREACH_HOST_UNKNOWN       7    /*目的主机不可达*/
#endif
#ifndef    ICMP_UNREACH_ISOLATED
#define    ICMP_UNREACH_ISOLATED           8    /*源主机被隔离*/
#endif
#ifndef    ICMP_UNREACH_NET_PROHIB
#define    ICMP_UNREACH_NET_PROHIB         9    /*目的网络被强制禁止*/
#endif
#ifndef    ICMP_UNREACH_HOST_PROHIB
#define    ICMP_UNREACH_HOST_PROHIB        10   /*目的主机被强制禁止*/
#endif
#ifndef    ICMP_UNREACH_TOSNET
#define    ICMP_UNREACH_TOSNET             11   /*由于 Tos，网络不可达*/
#endif
#ifndef    ICMP_UNREACH_TOSHOST
#define    ICMP_UNREACH_TOSHOST            12   /*由于 Tos，主机不可达*/
#endif
#ifndef    ICMP_UNREACH_FILTER_PROHIB
#define    ICMP_UNREACH_FILTER_PROHIB      13   /*通信被强制禁止*/
#endif
#ifndef    ICMP_UNREACH_HOST_PRECEDENCE
#define    ICMP_UNREACH_HOST_PRECEDENCE    14   /*主机越权*/
#endif
#ifndef    ICMP_UNREACH_PRECEDENCE_CUTOFF
#define    ICMP_UNREACH_PRECEDENCE_CUTOFF  15   /*优先权中止生效*/
#endif
#ifndef    ICMP_REDIRECT_NET
#define    ICMP_REDIRECT_NET               0    /*网络重定向*/
#endif
#ifndef    ICMP_REDIRECT_HOST
#define    ICMP_REDIRECT_HOST              1    /*主机重定向*/
#endif
#ifndef    ICMP_REDIRECT_TOSNET
#define    ICMP_REDIRECT_TOSNET            2    /*Tos 和网络重定向*/
#endif
#ifndef    ICMP_REDIRECT_TOSHOST
#define    ICMP_REDIRECT_TOSHOST           3    /*Tos 和主机重定向*/
#endif
```

```
#ifndef     ICMP_TIMXCEED_INTRANS
#define     ICMP_TIMXCEED_INTRANS          0    /*传输时 TTL 为 0 */
#endif
#ifndef     ICMP_TIMXCEED_REASS
#define     ICMP_TIMXCEED_REASS            1    /*封装时 TTL 为 0*/
#endif
#ifndef     ICMP_PARAMPROB_OPTABSENT
#define     ICMP_PARAMPROB_OPTABSENT       1    /*缺少选项*/
#endif
u_int16_t icmp_sum;          /* ICMP 校验和*/
union
{
  struct
  {
         u_int16_t id;       /* ICMP 标志*/
         u_int16_t seq;      /* ICMP 序列 */
  } echo;
  #undef icmp_id
  #undef icmp_seq
  #define icmp_id     hun.echo.id
  #define icmp_seq    hun.echo.seq
   u_int32_t gateway;        /* 网关主机 */
  struct
  {
      u_int16_t pad;         /* 填充*/
      u_int16_t mtu;         /* MTU 大小 */
  } frag;
} hun;
union
{
   struct
   {
      n_time its_otime;
      n_time its_rtime;
      n_time its_ttime;
   } ts;
   struct
   {
      struct libnet_ipv4_hdr idi_ip;
   } ip;
   u_int32_t mask;
   int8_t data[1];
   #undef icmp_mask
```

```
        #define icmp_mask    dun.mask
        #undef icmp_data
        #define icmp_data    dun.data
        #undef icmp_otime
        #define icmp_otime   dun.ts.its_otime
        #undef icmp_rtime
        #define icmp_rtime   dun.ts.its_rtime
        #undef icmp_ttime
        #define icmp_ttime   dun.ts.its_ttime
    }dun;
};
```

（20）IGMP 协议（Internet Group Message Protocol）。

```
struct libnet_igmp_hdr
{
    u_int8_t igmp_type;                          /* IGMP 类型 */
    #ifndef IGMP_MEMBERSHIP_QUERY
    #define IGMP_MEMBERSHIP_QUERY          0x11  /*成员查询 */
    #endif
    #ifndef IGMP_V1_MEMBERSHIP_REPORT
    #define IGMP_V1_MEMBERSHIP_REPORT      0x12  /* 版本 1 成员查询 */
    #endif
    #ifndef IGMP_V2_MEMBERSHIP_REPORT
    #define IGMP_V2_MEMBERSHIP_REPORT      0x16  /* 版本 2 成员查询 */
    #endif
    #ifndef IGMP_LEAVE_GROUP
    #define IGMP_LEAVE_GROUP               0x17  /* Leave-group 消息*/
    #endif
    u_int8_t igmp_code;          /* IGMP 代码 */
    u_int16_t igmp_sum;          /* IGMP 校验和*/
    struct in_addr igmp_group;   /* IGMP 主机 IP*/
};
```

（21）网际协议安全协议 IPSec（Internet Protocol Security Protocol）。

```
#ifndef IPPROTO_ESP
#define IPPROTO_ESP 50
#endif
struct libnet_esp_hdr
{
  u_int32_t esp_spi;             /* 安全参数索引 */
  u_int32_t esp_seq;             /* ESP 序列 */
  u_int32_t esp_iv;              /* 初始向量 */
};
struct libnet_esp_ftr
{
```

```
    u_int8_t esp_pad_len;          /*填充长度*/
    u_int8_t esp_nh;               /* 下一个头 */
    int8_t *esp_auth;              /* 鉴定数据 */
};
#ifndef IPPROTO_AH
#define IPPROTO_AH 51
#endif
struct libnet_ah_hdr
{
    u_int8_t ah_nh;                /*下一个头 */
    u_int8_t ah_len;               /* 负载长度 */
    u_int16_t ah_res;              /* 保留*/
    u_int32_t ah_spi;              /* 安全参数索引 */
    u_int32_t ah_seq;              /* AH序列码/
    u_int32_t ah_auth;             /*鉴定数据*/
};
```

（22）ISL 协议（Cisco Inter-Switch Link）。

```
#define LIBNET_PROTO_ISL    201
struct libnet_isl_hdr
{
    u_int8_t isl_dhost[5];         /* 目的地址 */
    #if (LIBNET_LIL_ENDIAN)
    u_int8_t isl_type:4,           /* 帧类型 */
             isl_user:4;           /* 用户定义 */
    #endif
    #if (LIBNET_BIG_ENDIAN)
    u_int8_t isl_user:4,           /* 用户定义 */
             isl_type:4;           /* 帧类型 */
    #endif
    u_int8_t isl_shost[6];         /* 源地址 */
    u_int16_t isl_len;             /* 数据包总长度 */
    u_int8_t isl_snap[6];          /* 向量代码 */
    u_int16_t isl_vid;             /* 15 位VLAN ID, 1 位 BPDU / CDP 指示器*/
    u_int16_t isl_index;           /* 端口索引*/
    u_int16_t isl_reserved;        /* FDDI 和令牌环使用 */
};
#ifndef IPPROTO_OSPF
#define IPPROTO_OSPF        89
#endif
#define IPPROTO_OSPF_LSA    890
#define LIBNET_MODX         4102
#define LIBNET_OPT_EBIT     0x02
#define LIBNET_OPT_MCBIT    0x04
```

```
#define LIBNET_OPT_NPBIT     0x08
#define LIBNET_OPT_EABIT     0x10
#define LIBNET_OPT_DCBIT     0x20
```

（23）MPLS 协议（Multi-Protocol Label Switching）。

```
struct libnet_mpls_hdr
{
   u_int32_t mpls_les;          /* 20 位标记, 3 位 exp, 1 位 bos, ttl */
   #define LIBNET_MPLS_BOS_ON       1
   #define LIBNET_MPLS_BOS_OFF      0
};
```

（24）NTP 协议（Network Time Protocol）。

```
struct libnet_ntp_hdr_l_fp
{
   u_int32_t integer;
   u_int32_t fraction;
};
struct libnet_ntp_hdr_s_fp
{
   u_int16_t integer;
   u_int16_t fraction;
};
struct libnet_ntp_hdr
{
   u_int8_t ntp_li_vn_mode;                    /*跳跃指示器, 版本, 模式*/
   #define LIBNET_NTP_LI_NW    0x0             /*无警告*/
   #define LIBNET_NTP_LI_AS    0x1             /*后一分钟有 61 秒*/
   #define LIBNET_NTP_LI_DS    0x2             /*后一分钟有 59 秒*/
   #define LIBNET_NTP_LI_AC    0x3             /* 报警条件 */
   #define LIBNET_NTP_VN_2     0x2             /* 版本 2*/
   #define LIBNET_NTP_VN_3     0x3             /*版本 3 */
   #define LIBNET_NTP_VN_4     0x4             /* 版本 4 */
   #define LIBNET_NTP_MODE_R   0x0             /* 保留 */
   #define LIBNET_NTP_MODE_A   0x1             /*对称主动 */
   #define LIBNET_NTP_MODE_P   0x2             /*对称被动 */
   #define LIBNET_NTP_MODE_C   0x3             /*客户端*/
   #define LIBNET_NTP_MODE_S   0x4             /*服务器*/
   #define LIBNET_NTP_MODE_B   0x5             /* 广播*/
   #define LIBNET_NTP_MODE_RC  0x6             /* 保留*/
   #define LIBNET_NTP_MODE_RP  0x7             /* 保留 */
   u_int8_t ntp_stratum;                       /* stratum */
   #define LIBNET_NTP_STRATUM_UNAVAIL  0x0     /* 未指明或不可用 */
   #define LIBNET_NTP_STRATUM_PRIMARY  0x1     /* 原始引用*/
   u_int8_t ntp_poll;                          /* 选择间隔 */
```

```
    u_int8_t ntp_precision;                          /* 本地时钟精度*/
    struct libnet_ntp_hdr_s_fp ntp_delay;            /* roundtrip 时延 */
    struct libnet_ntp_hdr_s_fp ntp_dispersion;       /* nominal 错误*/
    u_int32_t ntp_reference_id;                      /* 参考源 id */
    #define LIBNET_NTP_REF_LOCAL    0x4c4f434c       /* 非标准本地时钟 */
    #define LIBNET_NTP_REF_PPS      0x50505300       /*原子脉冲时钟 */
    #define LIBNET_NTP_REF_ACTS     0x41435453       /* NIST 拨号调制解调器*/
    #define LIBNET_NTP_REF_USNO     0x55534e4f       /* USNO 调制解调器服务*/
    #define LIBNET_NTP_REF_PTB      0x50544200       /* PTB (德国) 调制解调器服务*/
    #define LIBNET_NTP_REF_TDF      0x54444600       /* Allouis (法语)无线 */
    #define LIBNET_NTP_REF_DCF      0x44434600       /* Mainflingen (德语) 无线 */
    #define LIBNET_NTP_REF_MSF      0x4d534600       /* Rugby (UK) 无线*/
    #define LIBNET_NTP_REF_WWV      0x57575600       /* Ft Collins (US) 无线*/
    #define LIBNET_NTP_REF_WWVB     0x57575642       /* Boulder (US) 无线*/
    #define LIBNET_NTP_REF_WWVH     0x57575648       /* Kaui Hawaii (US) 无线*/
    #define LIBNET_NTP_REF_CHU      0x43485500       /* Ottaha (Canada) 无线*/
    #define LIBNET_NTP_REF_LORC     0x4c4f5243       /* LORAN-C 无线电导航 */
    #define LIBNET_NTP_REF_OMEG     0x4f4d4547       /* OMEGA 无线电导航*/
    #define LIBNET_NTP_REF_GPS      0x47505300       /* 全球定位系统 */
    #define LIBNET_NTP_REF_GOES     0x474f4553       /*固定轨道卫星 */
    struct libnet_ntp_hdr_l_fp ntp_ref_ts;           /*参考时间戳*/
    struct libnet_ntp_hdr_l_fp ntp_orig_ts;          /* 起源时间戳*/
    struct libnet_ntp_hdr_l_fp ntp_rec_ts;           /* 接收时间戳*/
    struct libnet_ntp_hdr_l_fp ntp_xmt_ts;           /*发送时间戳*/
};
```

（25）OSPFv2 协议。

```
struct libnet_ospf_hdr
{
    u_int8_t ospf_v;                        /* 版本 */
    #define OSPFVERSION           2
    u_int8_t ospf_type;                     /*类型*/
    #define  LIBNET_OSPF_UMD      0        /* UMD 监视数据包 */
    #define  LIBNET_OSPF_HELLO    1        /* HELLO 数据包*/
    #define  LIBNET_OSPF_DBD      2        /*  数据库描述包*/
    #define  LIBNET_OSPF_LSR      3        /* 链路状态询问包*/
    #define  LIBNET_OSPF_LSU      4        /* 链路状态更新包*/
    #define  LIBNET_OSPF_LSA      5        /* 链路状态确认包 */
    u_int16_t   ospf_len;                   /* 长度*/
    struct in_addr ospf_rtr_id;             /* 源路由 ID*/
    struct in_addr ospf_area_id;            /*漫游 ID */
    u_int16_t ospf_sum;                     /* 校验和*/
    u_int16_t ospf_auth_type;               /* 鉴定类型 */
    #define LIBNET_OSPF_AUTH_NULL     0     /*空密码*/
```

```
   #define LIBNET_OSPF_AUTH_SIMPLE    1    /* 简单密码 */
   #define LIBNET_OSPF_AUTH_MD5       2    /* MD5 */
};
struct libnet_auth_hdr
{
   u_int16_t ospf_auth_null;          /* 空 */
   u_int8_t ospf_auth_keyid;          /*鉴定关键字 ID*/
   u_int8_t ospf_auth_len;            /*鉴定数据长度*/
   u_int ospf_auth_seq;               /* 密码序列号 */
};
struct libnet_ospf_hello_hdr
{
   struct in_addr hello_nmask;        /* 接口掩码 */
   u_int16_t hello_intrvl;            /*下一个路由包的秒数 */
   u_int8_t hello_opts;               /* 选项 */
   u_int8_t hello_rtr_pri;            /*路由优先级 */
   u_int hello_dead_intvl;
   struct in_addr hello_des_rtr;      /* 目的路由器*/
   struct in_addr hello_bkup_rtr;     /* 备份路由器 */
   struct in_addr hello_nbr;          /* 相邻路由器 */
};
struct libnet_dbd_hdr
{
   u_int16_t dbd_mtu_len;             /* IP 数据包的最大长度 */
   u_int8_t dbd_opts;                 /* DBD 数据包选项 */
   u_int8_t dbd_type;                 /* 交换事件类型*/
   #define LIBNET_DBD_IBI      0x01
   #define LIBNET_DBD_MBIT     0x02   /*更多的 DBD 数据包*/
   #define LIBNET_DBD_MSBIT    0x04
   u_int  dbd_seq;
/* DBD 序列码 */
};
```

（26）LSA 数据格式。

```
#define LIBNET_LS_TYPE_RTR       1        /* 路由-LSA */
#define LIBNET_LS_TYPE_NET       2        /* 网络-LSA */
#define LIBNET_LS_TYPE_IP        3        /* 摘要-LSA (IP Network) */
#define LIBNET_LS_TYPE_ASBR      4        /* 摘要-LSA (ASBR) */
#define LIBNET_LS_TYPE_ASEXT     5        /* AS-外部-LSA */
struct libnet_lsr_hdr
{
   u_int lsr_type;                        /* 被查询的 LS 类型 */
   u_int lsr_lsid;                        /*链路状态 ID */
   struct in_addr lsr_adrtr;              /* 广告路由 */
```

```
};
struct libnet_lsu_hdr
{
    u_int lsu_num;                          /* 被广播的LSA个数 */
};
struct libnet_lsa_hdr
{
    u_int16_t lsa_age;                      /* 自LSA被创建的秒数 */
    u_int8_t lsa_opts;                      /* OPTS*/
    u_int8_t lsa_type;
    u_int lsa_id;                           /* 链路状态ID */
    struct in_addr lsa_adv;                 /* 确认路由器ID */
    u_int lsa_seq;                          /* LSA序列 */
    u_int16_t lsa_sum;                      /* 校验和 */
    u_int16_t lsa_len;                      /* 长度 */
};
struct libnet_rtr_lsa_hdr
{
    u_int16_t rtr_flags;                    /* 标记 */
    #define LIBNET_RTR_FLAGS_W  0x0100      /* W 位*/
    #define LIBNET_RTR_FLAGS_E  0x0200      /* E 位*/
    #define LIBNET_RTR_FLAGS_B  0x0400      /* B位 */
    u_int16_t rtr_num;                      /* 数目 */
    u_int rtr_link_id;                      /* ID */
    #define LIBNET_LINK_ID_NBR_ID   1       /* 邻居路由器ID*/
    #define LIBNET_LINK_ID_IP_DES   2       /* 目标路由器IP地址 */
    #define LIBNET_LINK_ID_SUB      3       /* IP 子网数目*/
    u_int rtr_link_data;                    /* 链路数据*/
    u_int8_t rtr_type;                      /* 路由器链路类型*/
    #define LIBNET_RTR_TYPE_PTP     1       /* 点对点 */
    #define LIBNET_RTR_TYPE_TRANS   2       /* 连接公网 */
    #define LIBNET_RTR_TYPE_STUB    3       /* 连接子网 */
    #define RTR_TYPE_VRTL           4       /* 连接虚拟网络 */
    u_int8_t rtr_tos_num;                   /* 不同TOS metrics 数目*/
    u_int16_t rtr_metric;
};
struct libnet_net_lsa_hdr
{
    struct in_addr net_nmask;               /* 网络掩码 */
    u_int  net_rtr_id;                      /* 路由器ID*/
};
struct libnet_sum_lsa_hdr
{
```

```
    struct in_addr sum_nmask;                /* 目的 IP 地址掩码*/
    u_int  sum_metric;
    u_int  sum_tos_metric;
};
struct libnet_as_lsa_hdr
{
    struct in_addr as_nmask;                 /* 广告目的网络掩码 */
    u_int  as_metric;
    #define LIBNET_AS_E_BIT_ON 0x80000000
    struct in_addr as_fwd_addr;              /* 转发地址 */
    u_int  as_rte_tag;                       /* 外部路由标记 */
};
```

（27）RIP 协议。

```
struct libnet_rip_hdr
{
    u_int8_t rip_cmd;                        /* RIP 命令 */
    #define RIPCMD_REQUEST     1             /* 查询信息 */
    #define RIPCMD_RESPONSE    2             /* 查询的应答 */
    #define RIPCMD_TRACEON     3             /* 打开 */
    #define RIPCMD_TRACEOFF    4             /* 关闭 */
    #define RIPCMD_POLL        5             /* 查询 */
    #define RIPCMD_POLLENTRY   6             /* 查询 */
    #define RIPCMD_MAX         7
    u_int8_t rip_ver;                        /* RIP 版本 */
    #define RIPVER_0           0
    #define RIPVER_1           1
    #define RIPVER_2           2
    u_int16_t rip_rd;                        /* 0 (v1) or 路由域 (v2) */
    u_int16_t rip_af;                        /* 地址簇 */
    u_int16_t rip_rt;                        /* 0 (v1) or 路由标记 (v2) */
    u_int32_t rip_addr;                      /* IP 地址 */
    u_int32_t rip_mask;                      /* 0 (v1) or 子网掩码(v2) */
    u_int32_t rip_next_hop;                  /* 0 (v1) or 下一跳 IP 地址(v2) */
    u_int32_t rip_metric;                    /* Metric */
};
```

（28）RPC 协议。

```
#define LIBNET_RPC_CALL      0
#define LIBNET_RPC_REPLY     1
#define LIBNET_RPC_VERS      2
#define LIBNET_RPC_LAST_FRAG 0x80000000
#define LIBNET_PMAP_PROGRAM          100000
#define LIBNET_PMAP_PROC_NULL        0
#define LIBNET_PMAP_PROC_SET         1
```

```
#define LIBNET_PMAP_PROC_UNSET        2
#define LIBNET_PMAP_PROC_GETADDR      3
#define LIBNET_PMAP_PROC_DUMP         4
#define LIBNET_PMAP_PROC_CALLIT       5
#define LIBNET_PMAP_PROC_BCAST        5
#define LIBNET_PMAP_PROC_GETTIME      6
#define LIBNET_PMAP_PROC_UADDR2TADDR  7
#define LIBNET_PMAP_PROC_TADDR2UADDR  8
#define LIBNET_PMAP_PROC_GETVERSADDR  9
#define LIBNET_PMAP_PROC_INDIRECT     10
#define LIBNET_PMAP_PROC_GETADDRLIST  11
#define LIBNET_PMAP_PROC_GETSTAT      12
struct libnet_rpc_opaque_auth
{
   u_int32_t rpc_auth_flavor;
   u_int32_t rpc_auth_length;
};
struct libnet_rpc_call
{
   u_int32_t rpc_rpcvers;              /* 版本 */
   u_int32_t rpc_prognum;              /* 程序代码 */
   u_int32_t rpc_vers;                 /* 程序版本*/
   u_int32_t rpc_procedure;            /* RPC 过程 */
   struct libnet_rpc_opaque_auth rpc_credentials;
   struct libnet_rpc_opaque_auth rpc_verifier;
};
struct libnet_rpc_call_hdr
{
   u_int32_t rpc_xid;
   u_int32_t rpc_type;
   struct libnet_rpc_call  rpc_call;
};
struct libnet_rpc_call_tcp_hdr
{
   u_int32_t rpc_record_marking;
   struct libnet_rpc_call_hdr rpc_common;
};
```

（29）STP 协议（Spanning Tree Protocol）。

```
struct libnet_stp_conf_hdr
{
   u_int16_t stp_id;                   /* 协议 ID*/
   u_int8_t stp_version;               /* 协议版本*/
   u_int8_t stp_bpdu_type;             /* 桥协议数据单元类型 */
```

```
    u_int8_t stp_flags;                 /* 控制标记 */
    u_int8_t stp_rootid[8];             /* 根 ID */
    u_int32_t stp_rootpc;               /* 根目录 ID */
    u_int8_t stp_bridgeid[8];           /* 桥 ID */
    u_int16_t stp_portid;               /* 端口 ID */
    u_int16_t stp_mage;                 /* 消息*/
    u_int16_t stp_maxage;               /* 最大时间 */
    u_int16_t stp_hellot;               /* 通知时间 */
    u_int16_t stp_fdelay;               /* 转发延迟 */
};
struct libnet_stp_tcn_hdr
{
    u_int16_t stp_id;                   /* 协议 ID*/
    u_int8_t stp_version;               /* 协议版本 */
    u_int8_t stp_bpdu_type;             /* 桥协议数据单元类型*/
};
```

（30）传输控制协议（TCP，Transmission Control Protocol）。

```
struct libnet_tcp_hdr
{
    u_int16_t th_sport;                 /* 源端口 */
    u_int16_t th_dport;                 /* 目的端口 */
    u_int32_t th_seq;                   /* 序列号 */
    u_int32_t th_ack;                   /* 确认序列号 */
    #if (LIBNET_LIL_ENDIAN)
    u_int8_t th_x2:4,                   /* 未用*/
             th_off:4;                  /* 数据偏移 */
    #endif
    #if (LIBNET_BIG_ENDIAN)
    u_int8_t th_off:4,                  /* 数据偏移*/
          th_x2:4;                      /* 未用*/
    #endif
    u_int8_t  th_flags;                 /* 控制标记 */
    #ifndef TH_FIN
    #define TH_FIN    0x01              /* 发送数据结束 */
    #endif
    #ifndef TH_SYN
    #define TH_SYN    0x02              /* 同步序列号 */
    #endif
    #ifndef TH_RST
    #define TH_RST    0x04              /* 重置连接*/
    #endif
    #ifndef TH_PUSH
    #define TH_PUSH   0x08              /*把数据给应用层 */
```

```
    #endif
    #ifndef TH_ACK
    #define TH_ACK    0x10              /* 确认*/
    #endif
    #ifndef TH_URG
    #define TH_URG    0x20              /* 紧急*/
    #endif
    #ifndef TH_ECE
    #define TH_ECE    0x40
    #endif
    #ifndef TH_CWR
    #define TH_CWR    0x80
    #endif
    u_int16_t th_win;                   /* 窗口 */
    u_int16_t th_sum;                   /* 校验和*/
    u_int16_t th_urp;                   /* 紧急指针*/
};
```

（31）令牌环（Token Ring）。

```
struct libnet_token_ring_hdr
{
    u_int8_t  token_ring_access_control;
    #define LIBNET_TOKEN_RING_FRAME  0x10
    u_int8_t  token_ring_frame_control;
    #define LIBNET_TOKEN_RING_LLC_FRAME  0x40
    u_int8_t  token_ring_dhost[TOKEN_RING_ADDR_LEN];
    u_int8_t  token_ring_shost[TOKEN_RING_ADDR_LEN];
    u_int8_t  token_ring_llc_dsap;
    u_int8_t  token_ring_llc_ssap;
    u_int8_t  token_ring_llc_control_field;
    u_int8_t  token_ring_llc_org_code[LIBNET_ORG_CODE_SIZE];
    u_int16_t token_ring_type;
    #define TOKEN_RING_TYPE_IP              0x0800  /* IP 协议 */
    #define TOKEN_RING_TYPE_ARP             0x0806  /* ARP 协议 */
    #define TOKEN_RING_TYPE_REVARP          0x8035  /*RARP 协议 */
};
struct libnet_token_ring_addr
{
    u_int8_t  token_ring_addr_octet[6];     /*令牌环地址 */
};
```

（32）用户数据报协议（UDP，User Datagram Protocol）。

```
struct libnet_udp_hdr
{
    u_int16_t uh_sport;            /* 源端口 */
```

```
    u_int16_t uh_dport;            /* 目的端口 */
    u_int16_t uh_ulen;             /* 长度/
    u_int16_t uh_sum;              /* 校验和*/
};
```

（33）Sebek 头。

```
struct libnet_sebek_hdr
{
    u_int32_t magic;               /*确定应该被隐藏的数据包 */
    u_int16_t version;             /* 协议版本*/
    #define SEBEK_PROTO_VERSION 1
    u_int16_t type;                /* 记录的类型*/
    #define SEBEK_TYPE_READ        0
    #define SEBEK_TYPE_WRITE       1
    u_int32_t counter;             /*  PDU 计数器用来确定丢失包 */
    u_int32_t time_sec;            /* EPOCH 秒数 */
    u_int32_t time_usec;           /* 剩余的微秒数 */
    u_int32_t pid;                 /* PID */
    u_int32_t uid;                 /* UID */
    u_int32_t fd;                  /* FD */
    #define SEBEK_CMD_LENGTH   12
    u_int8_t cmd[SEBEK_CMD_LENGTH];    /*命令的前 12 个字符*/
    u_int32_t length;              /* PDU 主体的字节长度*/
};
```

（34）VRRP 协议（Virtual Router Redundancy Protocol）。

```
#ifndef IPPROTO_VRRP
#define IPPROTO_VRRP 112
#endif
struct libnet_vrrp_hdr
{
    #if (LIBNET_LIL_ENDIAN)
    u_int8_t vrrp_v:4,                 /* 协议版本*/
             vrrp_t:4;                 /* 数据包类型*/
    #endif
    #if (LIBNET_BIG_ENDIAN)
    u_int8_t vrrp_t:4,                 /*数据包类型*/
             vrrp_v:4;                 /*协议版本*/
    #endif
    #define LIBNET_VRRP_VERSION_01  0x1
    #define LIBNET_VRRP_VERSION_02  0x2
    #define LIBNET_VRRP_TYPE_ADVERT 0x1
    u_int8_t vrrp_vrouter_id;          /* 虚拟路由 ID*/
    u_int8_t vrrp_priority;            /* 优先级*/
    u_int8_t vrrp_ip_count;            /*IP 地址数目*/
```

```
    u_int8_t vrrp_auth_type;            /* 许可类型 */
    #define LIBNET_VRRP_AUTH_NONE       0x1
    #define LIBNET_VRRP_AUTH_PASSWD     0x2
    #define LIBNET_VRRP_AUTH_IPAH       0x3
    u_int8_t vrrp_advert_int;           /* 广告间距*/
    u_int16_t vrrp_sum;                 /* 校验和 */
};
```

5.3.3 核心数据结构

与 Libnet 开发包密切相关的一些核心数据结构全部定义在 libnet-structures.h 文件中。它们在开发程序中占据重要的地位，Libnet 的很多函数都要用到这些数据结构。可以说，这些数据结构存放了 Libnet 库函数的很多信息。下面分别对它们进行详细解释。

1. libnet_plist_t

libnet_plist_t 表示端口链表数据结构，其定义如下：

```
typedef struct libnet_port_list_chain libnet_plist_t;
struct libnet_port_list_chain
{
    u_int16_t node;                     /* 链表结点个数 */
    u_int16_t bport;                    /* 开始的端口*/
    u_int16_t eport;                    /* 终止的端口 */
    u_int8_t  id;                       /* 全局排列偏移量*/
    libnet_plist_t *next;               /* 链表的下一个结点 */
};
```

端口链表数据结构在构造 UDP 和 TCP 数据包时非常有用。在 UDP 和 TCP 协议格式中具有源端口号和目的端口号，如果想构造一系列的 UDP 或 TCP 数据包，但是要求它们的端口号的分布有一定的规律，此时就可以使用此端口链表数据结构。它使发送多个不同端口号的 TCP 或 UDP 数据包变得非常容易。这在 5.6 节有详细介绍。

2. libnet_stats

libnet_stats 表示 Libnet 的统计分析的数据结构，其定义如下：

```
struct libnet_stats
{
#if (!defined(__WIN32__) || (__CYGWIN__))
    u_int64_t packets_sent;             /*发送的数据包*/
    u_int64_t packet_errors;            /*出错的数据包*/
    u_int64_t bytes_written;            /*已经写的字节数目 */
#else
    __int64 packets_sent;               /*发送的数据包 */
    __int64 packet_errors;              /*出错的数据包 */
    __int64 bytes_written;              /* 已写的字节数目*/
#endif
};
```

此统计分析数据结构主要是对当前的 Libnet 环境进行状态统计。其中包括发送数据包个数的、出错数据包个数以及发送字节数等信息。在任意时刻都可以调用函数 libnet_stats() 来获取此数据结构的内容。

3. libnet_ptag_t

libnet_ptag_t 表示一个 Libnet 协议标记，用来在链表中来识别一个特定的协议块，其定义如下：

```
typedef int32_t libnet_ptag_t;
#define LIBNET_PTAG_INITIALIZER        0
```

此协议标记其实是一个 32 位的整数，它用来标识一个协议块。不同的协议块用一个不同的整数来表示，既简单又方便。这样，对协议块的管理变得非常容易。

4. libnet_pblock_t

libnet_pblock_t 表示 Libnet 的协议块的数据结构，其定义如下：

```
struct libnet_protocol_block
{
    u_int8_t *buf;                      /* 协议的缓冲区*/
    u_int32_t b_len;                    /* 缓冲区的长度 */
    u_int16_t h_len;                    /* 首部长度 */
    u_int32_t ip_offset;                /* IP 首部的偏移量*/
    u_int32_t copied;                   /* 拷贝的字节数目 */
    u_int8_t type;                      /* 协议块的类型*/
    #define LIBNET_PBLOCK_ARP_H             0x01      /* ARP 协议头 */
    #define LIBNET_PBLOCK_DHCPV4_H          0x02      /* DHCP v4 协议头 */
    #define LIBNET_PBLOCK_DNSV4_H           0x03      /* DNS v4 协议头*/
    #define LIBNET_PBLOCK_ETH_H             0x04      /* Ethernet 协议头*/
    #define LIBNET_PBLOCK_ICMPV4_H          0x05      /* ICMP v4 base 协议头*/
    #define LIBNET_PBLOCK_ICMPV4_ECHO_H     0x06      /* ICMP v4 echo 协议头*/
    #define LIBNET_PBLOCK_ICMPV4_MASK_H     0x07      /* ICMP v4 mask 协议头*/
    #define LIBNET_PBLOCK_ICMPV4_UNREACH_H   0x08     /* ICMP v4 unreach 协议头*/
    #define LIBNET_PBLOCK_ICMPV4_TIMXCEED_H  0x09     /* ICMP v4 exceed 协议头*/
    #define LIBNET_PBLOCK_ICMPV4_REDIRECT_H  0x0a     /* ICMP v4 redirect 协议头*/
    #define LIBNET_PBLOCK_ICMPV4_TS_H       0x0b      /* ICMP v4 timestamp 协议头*/
    #define LIBNET_PBLOCK_IGMP_H            0x0c      /* IGMP 协议头*/
    #define LIBNET_PBLOCK_IPV4_H            0x0d      /* IP v4 协议头*/
    #define LIBNET_PBLOCK_IPO_H             0x0e      /* IP v4 options */
    #define LIBNET_PBLOCK_IPDATA            0x0f      /* IP data */
    #define LIBNET_PBLOCK_OSPF_H            0x10      /* OSPF base 协议头*/
    #define LIBNET_PBLOCK_OSPF_HELLO_H      0x11      /* OSPF hello 协议头*/
    #define LIBNET_PBLOCK_OSPF_DBD_H        0x12      /* OSPF dbd 协议头*/
    #define LIBNET_PBLOCK_OSPF_LSR_H        0x13      /* OSPF lsr 协议头*/
    #define LIBNET_PBLOCK_OSPF_LSU_H        0x14      /* OSPF lsu 协议头*/
    #define LIBNET_PBLOCK_OSPF_LSA_H        0x15      /* OSPF lsa 协议头*/
```

```
#define LIBNET_PBLOCK_OSPF_AUTH_H        0x16     /* OSPF auth 协议头*/
#define LIBNET_PBLOCK_OSPF_CKSUM         0x17     /* OSPF checksum 协议头*/
#define LIBNET_PBLOCK_LS_RTR_H           0x18     /* linkstate rtr 协议头*/
#define LIBNET_PBLOCK_LS_NET_H           0x19     /* linkstate net 协议头*/
#define LIBNET_PBLOCK_LS_SUM_H           0x1a     /* linkstate as sum 协议头*/
#define LIBNET_PBLOCK_LS_AS_EXT_H        0x1b     /* linkstate as ext 协议头*/
#define LIBNET_PBLOCK_NTP_H              0x1c     /* NTP 协议头*/
#define LIBNET_PBLOCK_RIP_H              0x1d     /* RIP 协议头*/
#define LIBNET_PBLOCK_TCP_H              0x1e     /* TCP 协议头*/
#define LIBNET_PBLOCK_TCPO_H             0x1f     /* TCP options */
#define LIBNET_PBLOCK_TCPDATA            0x20     /* TCP data */
#define LIBNET_PBLOCK_UDP_H              0x21     /* UDP 协议头*/
#define LIBNET_PBLOCK_VRRP_H             0x22     /* VRRP 协议头*/
#define LIBNET_PBLOCK_DATA_H             0x23     /* generic data */
#define LIBNET_PBLOCK_CDP_H              0x24     /* CDP 协议头*/
#define LIBNET_PBLOCK_IPSEC_ESP_HDK_H    0x25     /* IPSEC ESP 协议头*/
#define LIBNET_PBLOCK_IPSEC_ESP_FTR_H    0x26     /* IPSEC ESP footer */
#define LIBNET_PBLOCK_IPSEC_AH_H         0x27     /* IPSEC AH 协议头*/
#define LIBNET_PBLOCK_802_1Q_H           0x28     /* 802.1q 协议头*/
#define LIBNET_PBLOCK_802_2_H            0x29     /* 802.2 协议头*/
#define LIBNET_PBLOCK_802_2SNAP_H        0x2a     /* 802.2 SNAP 协议头*/
#define LIBNET_PBLOCK_802_3_H            0x2b     /* 802.3 协议头*/
#define LIBNET_PBLOCK_STP_CONF_H         0x2c     /* STP configuration 协议头*/
#define LIBNET_PBLOCK_STP_TCN_H          0x2d     /* STP TCN 协议头*/
#define LIBNET_PBLOCK_ISL_H              0x2e     /* ISL 协议头*/
#define LIBNET_PBLOCK_IPV6_H             0x2f     /* IP v6 协议头*/
#define LIBNET_PBLOCK_802_1X_H           0x30     /* 802.1x 协议头*/
#define LIBNET_PBLOCK_RPC_CALL_H         0x31     /* RPC Call 协议头*/
#define LIBNET_PBLOCK_MPLS_H             0x32     /* MPLS 协议头*/
#define LIBNET_PBLOCK_FDDI_H             0x33     /* FDDI 协议头*/
#define LIBNET_PBLOCK_TOKEN_RING_H       0x34     /* TOKEN RING 协议头*/
#define LIBNET_PBLOCK_BGP4_HEADER_H      0x35     /* BGP4 协议头*/
#define LIBNET_PBLOCK_BGP4_OPEN_H        0x36     /* BGP4 open 协议头*/
#define LIBNET_PBLOCK_BGP4_UPDATE_H      0x37     /* BGP4 update 协议头*/
#define LIBNET_PBLOCK_BGP4_NOTIFICATION_H 0x38    /* BGP4 notification 协议头*/
#define LIBNET_PBLOCK_GRE_H              0x39     /* GRE 协议头*/
#define LIBNET_PBLOCK_GRE_SRE_H          0x3a     /* GRE SRE 协议头*/
#define LIBNET_PBLOCK_IPV6_FRAG_H        0x3b     /* IPv6 frag 协议头*/
#define LIBNET_PBLOCK_IPV6_ROUTING_H     0x3c     /* IPv6 routing 协议头*/
#define LIBNET_PBLOCK_IPV6_DESTOPTS_H    0x3d     /* IPv6 dest opts 协议头*/
#define LIBNET_PBLOCK_IPV6_HBHOPTS_H     0x3e     /* IPv6 hop/hop opts 协议头*/
#define LIBNET_PBLOCK_SEBEK_H            0x3f     /* Sebek 协议头*/
u_int8_t flags;                                   /* 控制标记 */
```

```
    #define LIBNET_PBLOCK_DO_CHECKSUM       0x01        /* 校验和 */
    libnet_ptag_t ptag;                                 /* 协议块标记*/
    struct libnet_protocol_block *next;                 /* 下一个 pblock */
    struct libnet_protocol_block *prev;                 /* 前一个 pblock */
};
typedef struct libnet_protocol_block libnet_pblock_t;
```

此数据结构是 Libnet 的一个核心数据结构，它表示一个特定的协议块。关于此协议块的一些重要信息如下：

（1）u_int8_t *buf; ——此成员表示此协议块的协议数据，它指向一段内存，此内存存储的内容就是此协议块的实际协议数据。例如 IP 协议块，它存放的是 IP 协议块的协议数据。

（2）u_int32_t b_len; ——此成员表示由 buf 存储的协议数据的长度。

（3）u_int8_t type; ——此成员表示此协议块的类型。Libnet 共定义了 63 种协议块类型，它们在此数据结构中都有定义。

（4）libnet_ptag_t ptag; ——此成员表示此协议块的协议标记，用一个 32 位整数来表示。

（5）struct libnet_protocol_block *next; ——此成员表示指向下一个协议块的指针。

（6）struct libnet_protocol_block *prev; ——此成员表示指向上一个协议块的指针。由 prev 指针和 next 指针构成协议块链表。

下面举一个例子，如图 5-18 所示，它表示一个 IP 协议块。

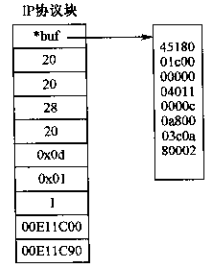

图 5-18 一个 IP 协议块的例子

5. libnet_t

libnet_t 表示 Libnet 句柄的数据结构。此数据结构是 Libnet 开发包中最重要的一个数据结构，它在 Libnet 的开发包中具有重要的意义。对此数据结构的理解，关系到对 Libnet 开发包的掌握程度，如果能够深刻理解并熟练掌握此数据结构，对 Libnet 开发包的使用就会变得得心应手。

libnet_t 数据结构的内部成员不需要程序员手工初始化，其定义如下：

```
struct libnet_context
{
    #if ((__WIN32__) && !(__CYGWIN__))
    SOCKET fd;
    LPADAPTER  lpAdapter;
    #else
    int fd;                                     /*数据包设备的文件描述符 */
    #endif
    int injection_type;  /* 类型，例如原始套接字 ipv4 或 ipv6 或链路层协议*/
    #define LIBNET_LINK     0x00                /* 链路层接口*/
    #define LIBNET_RAW4     0x01                /*原始套接字接口 (ipv4) */
```

（5）u_int32_t n_pblocks; ——此成员表示协议块的个数。

（6）char *device; ——此成员表示网络设备的名字。

（7）struct libnet_stats stats; ——此成员表示此 Libnet 的状态信息。

（8）u_int32_t total_size; ——此成员表示数据包的整体大小。

举一个例子，此例子表示了一个 ICMP 端口不可达数据包，如图 5-19 所示。掌握了此数据结构的机理，可以说 Libnet 开发包的精华基本上就掌握了。关于 ICMP 数据包的构造在 5.6.5 节有详细介绍。

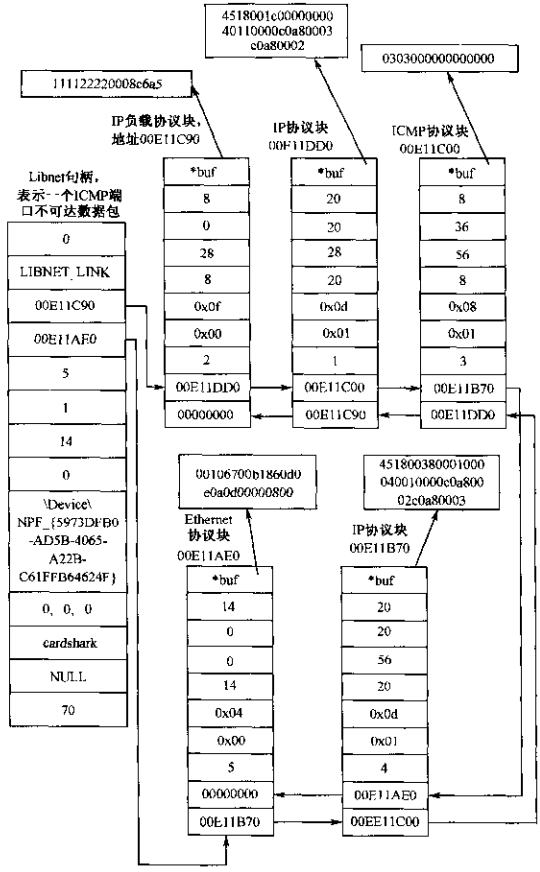

图 5-19　一个 ICMP 不可达数据包的 Libnet 数据结构

图 5-19 中的数据包有 5 个协议块。

- 第 1 个协议块其实是一个数据块，它是 IP 协议负载数据。在 Libnet 中一个数据块

与一个协议块具有同样的存储结构，所以都可以用协议块来统一表示，只是要知道它也可以表示一般意义上的数据块。在 Libnet 句柄结构中 protocol_blocks 成员就指向此协议块，所以 protocol_blocks 的数值就是此协议块的首地址，此协议块的 next 成员就指向后一个协议块，prev 成员指向空。

- 第 2 个协议块是 IP 协议块。其 next 成员指向后一个协议块，prev 成员指向第 1 个协议块。
- 第 3 个协议块是 ICMP 协议块。其 next 成员指向后一个协议块，prev 成员指向第 2 个协议块。
- 第 4 个协议块是 IP 协议块。其 next 成员指向后一个协议块，prev 成员指向第 3 个协议块。
- 第 5 个协议块是 Ethernet 协议块。其 next 成员指向空，prev 成员指向第 4 个协议块。

在 Libnet 句柄结构中，pblock_end 成员指向第 5 个协议块。这样，Libnet 句柄就包含一个完整的数据包，此数据包包含 5 个协议块，这些协议块依次进行构造。

6. libnet_cqd_t

libnet_cqd_t 表示 Libnet 句柄队列数据结构。同样，它的数据成员也不需要程序员来手工初始化。其定义如下：

```
typedef struct _libnet_context_queue libnet_cq_t;
struct _libnet_context_queue
{
    libnet_t *context;                  /*指向 Libnet 句柄的指针*/
    libnet_cq_t *next;                  /*队列的下一个结点*/
    libnet_cq_t *prev;                  /*队列的前一个结点 */
};
struct _libnet_context_queue_descriptor
{
    u_int32_t node;                     /*队列的结点个数 */
    u_int32_t cq_lock;                  /*锁的状态*/
    libnet_cq_t *current;               /*当前句柄*/
};
typedef struct _libnet_context_queue_descriptor libnet_cqd_t;
```

此数据结构表示一个 Libnet 句柄队列，它主要用在构造多个不同的数据包时。因为一个 Libnet 句柄就表示一个数据包，多个 Libnet 句柄就表示多个不同的数据包。所以，此数据结构就构造了一个 Libnet 句柄队列。

此 Libnet 句柄队列其实是一个双向链表。链表的每个结点共有 3 个域：context 表示 Libnet 句柄，next 表示下一个句柄结点，prev 表示上一个句柄结点。

整个 Libnet 句柄队列结构由 libnet_cqd_t 结构体表示，它有 3 个成员：结点个数 node，当前句柄 current 以及锁状态 cq_lock。锁状态表示此 Libnet 句柄是否上锁，如果上锁，就不能改写。

对此数据结构的操作由句柄队列操作函数来完成，其详细内容见 5.4.6 节。

5.4 Libnet 函数

5.4.1 核心函数

核心函数主要包括 Libnet 初始化函数和销毁函数以及一些必须的设置函数。Libnet 初始化函数主要完成了对 Libnet 的初始化功能，其中包括内存分配、网络接口设置和 Libnet 的类型设置等。内存分配主要实现的是在开始构造数据包之前分配足够的内存空间，在不需要内存的时候再销毁内存空间。主要的核心函数如下：

（1）libnet_t *libnet_init (int injection_type, char *device, char *err_buf)

函数返回值：如果成功，就返回一个 Libnet 的句柄，此句柄在后面的构造和发送的过程中都要用到；如果失败，就返回 NULL。

参数描述：参数 injection_type 表示要构造的类型，有 LIBNET_LINK, LIBNET_LINK_ADV, LIBNET_RAW4, LIBNET_RAW4_ADV, LIBNET_RAW6 和 LIBNET_RAW6_ADV。LIBNET_RAW4 表示原始套接字 IPv4 类型，LIBNET_RAW6 表示原始套接字 IPv6 类型，LIBNET_RAW4_ADV 表示高级模式原始套接字 IPv4 类型，LIBNET_RAW6_ADV 表示高级模式原始套接字 IPv6 类型，LIBNET_LINK 表示链路层类型，LIBNET_LINK_ADV 表示高级模式链路层类型。类型不同，libnet_init() 函数执行的结果也就不同。参数 device 表示网络接口，可以是 NULL，也可由 Libnet 来进行选择。参数 err_buf 存放出错的信息。

此函数是最重要的核心函数之一，任何函数的调用都是在该函数调用之后调用，所以说它是入口函数。此函数的主要功能是对开发环境进行初始化，它返回一个 Libnet 句柄，在后面的函数调用中要用到。如果构造的类型是 LIBNET_LINK 或 LIBNET_LINK_ADV，此函数就对链路层的数据包进行构造。这样，应用程序就能够发送从数据链路层开始构造的各种网络数据包了。如果 Libnet 使用链路层而且 device 参数是非空的，此函数就试图使用特定的网络设备来发送数据包。设备的名字可以是标准的字符串，以指定特定的设备类型。例如，在 Linux 中用“eth0”代表 100 Mbps 的以太网接口；在 OpsenBSD 中使用“fxp0”代表 100Mbps 以太网接口；在 Windows 环境下是一个字符串，如“\Device\NPF_{5973DFB0-AD5B-4065-A22B-C61FFB64624F}”。如果 device 是 NULL，Libnet 就试图寻找一个合适的网络接口来使用。如果 injectin_type 是 LIBNET_RAW4 或 LIBNET_RAW4_ADV，此函数就初始化 IPv4 原始套接字接口。最后一个参数 err_buf 是大小为 LIBNET_ERRBUF_SIZE 的一段内存，当函数调用出错时，它用来存放一个错误信息。

此函数的调用需要超级权限，如果没有超级权限就无法调用。如果调用成功，此函数就返回一个有效的 Libnet 句柄，此句柄在后面的函数中都要用到。如果调用失败，此函数就返回 NULL 值。举例如下：

```
libnet_t *l;
char errbuf[LIBNET_ERRBUF_SIZE];
l = libnet_init(LIBNET_RAW4, NULL, err);
```

（2）void libnet_destroy(libnet_t *l)

函数返回值：无。

参数描述：参数 l 是一个 Libnet 句柄指针。

此函数销毁有参数 l 指向的 Libnet 会话。它包括关闭网络接口和释放所有的有参数 l 创

建的内部内存结构。

此函数的调用一般在程序的末尾，它是与函数 libnet_init() 相对应的。当完成了所有的 Libnet 函数库的调用之后，就可以调用此函数来进行事后处理。

（3）void libnet_clear_packet(libnet_t *l)

函数返回值：无。

参数描述：参数 l 是指向 Libnet 句柄的指针。

此函数清除当前构造的数据包，并清空所有的 pblocks。此函数在当程序员想发送一个完全不同类型的数据包时被调用。

5.4.2 地址函数

地址函数主要是对 IP 地址进行操作的一些函数。

（1）char *libnet_addr2name4(u_int32_t in, u_int8_t use_name)

函数返回值：返回 IPv4 地址的文字描述字符串形式。

参数描述：参数 in 表示网络字节顺序的 IPv4 地址。参数 use_name 表示是否需要进行名字解析，它只有两个值，要么是 LIBNET_RESOLVE，要么是 LIBNET_DON'T_RESOLVE。

此函数实现了 IPv4 地址的转换功能。返回一个网络字节顺序的 IPv4 地址的 DNS 名字或者带点的十进制形式。如果参数 use_name 是 LIBNET_RESOLVE，就返回一个 DNS 名字；如果参数 use_name 是 LIBNET_DON'T_RESOLVE，就返回一个带点的十进制形式；如果是 LIBNET_RESOLVE，但是无法进行 DNS 查询获得名字，就返回带点的十进制形式的地址。

（2）u_int32_t libnet_name2addr4(libnet_t *l, char *host_name, u_int8_t use_name)

函数返回值：返回网络字节顺序的 IPv4 地址。

参数描述：参数 l 表示 Libnet 句柄；参数 host_name 表示主机的名字；参数 use_name 表示是否进行域名解析，有 LIBNET_RESOLVE 和 LIBNET_DON'T_RESOLVE 两种值。

此函数返回一个网络字节顺序的 IPv4 地址。如果 use_name 使用 LIBNET_RESOLVE，就进行域名解析；如果 use_name 使用 LIBNET_DON'T_RESOLVE，就不进行域名解析。

（3）struct libnet_in6_addr libnet_name2addr6(libnet_t *l, char *host_name, u_int8_t use_name)

函数返回值：返回一个网路字节顺序的 IPv6 地址。

参数描述：参数 l 表示一个 Libnet 句柄；参数 host_name 表示主机的名字；参数 use_name 表示是否进行域名解析，有 LIBNET_RESOLVE 和 LIBNET_DON'T_RESOLVE 两个值。

此函数返回一个网络字节顺序的 IPv6 地址。参数 host_name 表示一个带点的十进制 IP 地址或一个 DNS 名字。如果 use_name 使用 LIBNET_RESOLVE，就进行域名解析；如果 use_name 使用 LIBNET_DON'T_RESOLVE，就不进行域名解析。

（4）void libnet_addr2name6_r(struct libnet_in6_addr addr, u_int8_t use_uame,

char *host_name, int host_name_len)

函数返回值：无。

参数描述：参数 addr 表示一个 IPv6 地址；参数 use_name 表示是否进行域名解析，有网个值 LILBNET_RESOLVE 和 LIBNET_DON'T_RESOLVE；参数 host_name 表示主机名字；参数 host_name_len 表示主机名字的长度。

此函数把一个 IPv6 地址转化为一个字符串名字。

（5）u_int32_t libnet_get_ipaddr4(libnet_t *l)

函数返回值：返回一个 IP 地址。如果失败，就返回－1。

参数描述：参数 l 表示一个 Libnet 句柄。

此函数返回初始化设备的 IP 地址。如果 Libnet 在初始化时没有指定设备的名字，而且其模式为原始套接字模式，此函数就试图寻找一个设备名字。如果此函数操作失败，就返回－1，出错的原因可以通过函数 libnet_geterror()来获得；如果成功，就返回一个 IP 地址。

（6）struct libnet_in6_addr libnet_get_ipaddr6(libnet_t *l);

函数返回值：返回一个 IPv6 地址。

参数描述：参数 l 表示一个 Libnet 句柄。

此函数返回一个 IPv6 地址。

（7）struct libnet_ether_addr * libnet_get_hwaddr(libnet_t *l)

函数返回值：返回一个 MAC 地址。如果失败，就返回 NULL。

参数描述：参数 l 表示一个 Libnet 句柄。

此函数返回一个在 Libnet 初始化时设备的 MAC 地址。如果 Libnet 在初始化时没有指定设备，此函数就试图寻找一个。如果函数调用失败，就返回 NULL，可以使用 libnet_geterror()进行查看。

（8）u_int8_t *libnet_hex_aton(int8_t *s, int *len)

函数返回值：返回一个字符串。如果失败，就返回 NULL。

参数描述：参数 s 表示一个用冒号隔开的十六进制地址，参数 len 表示返回字符串的长度。

5.4.3 数据包构造函数

数据包处理函数包括数据包构造函数和数据包发送函数，其中数据包构造函数完成了所有的协议数据包构造的功能。由于 Libnet 可以构造很多协议格式的数据包，每种协议的数据包部分都用某一个函数来实现，这些函数的模式都是一样的，并且这些函数都是平台独立的。

每个协议构造函数都遵循同一模式进行调用。每个协议构造函数一般都包括下面四部分：与协议相关的一些选项；与协议相关的负载和它的长度；一个 Libnet 句柄；一个协议标记。

Libnet 使用一个协议标记来表示一部分协议数据。当使用协议构造函数构造一部分协议数据时，就产生一个新的协议标记来表示这一部分协议数据。这些协议标记必须存储起来，以便以后修改这些协议标记所代表的协议数据。如果构造函数操作失败，就返回－1，可以通过调用 libnet_geterror()函数来查看。

几乎每个协议构造函数都有一个参数 ptag，用来表示一个协议标记。如果其值是 0，就表示创建一个新的协议数据；如果其值不是 0，就修改由 ptag 表示的协议数据。

每个协议构造函数都返回一个协议标记，用来表示此函数创建的协议标记或修改后的协议标记。

可以看出每个创建的协议数据都用惟一的协议标记来代表。此协议标记其实是一个 32 位的整数，用一个简单的整数就代表一个协议数据，这是 Libnet 最优秀的地方。它把复杂的

协议数据只用一个整数来代表，其他操作都由 Libnet 内部自动完成，这使开发者使用起来非常方便。

一个完整的数据包通常由几部分协议构造函数来构造，其构造的顺序不能颠倒，必须按照从高到低的顺序进行。先构造应用层协议数据，再构造传输层协议数据，再构造网络层协议数据，最后构造链路层协议数据。

例如，要构造一个 DNS 数据包，就必须按照下面的调用顺序：

```
libnet_build_dnsv4()
libnet_build_udp()
libnet_build_ipv4()
libnet_build_ethernet()
```

下面对每个协议数据包的构造函数进行详细说明。

（1）libnet_ptag_t libnet_build_802_1q(u_int8_t *dst, u_int8_t *src, u_int16_t tpi,u_int8_t priority, u_int8_t cfi, u_int16_t vlan_id, u_int16_t len_proto,u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 dst 表示目的以太网地址。参数 src 表示源以太网地址。参数 tpi 表示协议类型。参数 priority 表示优先级。参数 cfi 表示规范形式指示器。参数 vlan_id 表示 vlan 标记。参数 len_proto 表示长度或协议，在 802.3 中表示长度，在 Ethernet II 中表示协议。参数 payload 表示负载，也可以是 NULL。参数 payload_s 表示负载的长度，可以是 0。参数 l 表示 Libnet 句柄。参数 ptag 表示协议标记，可以是已经存在的标记，也可以是 0。如果是已经存在的标记，此函数就修改此标记表示的协议数据；如果是 0，就表示此函数新建一个协议数据。

此函数构造一个 IEEE 802.1q VLAN 协议头。

（2）libnet_ptag_t libnet_build_802_1x(u_int8_t eap_ver, u_int8_t eap_type, u_int16_t length, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 eap_ver 表示 EAP 版本号；参数 eap_type 表示 EAP 类型；参数 length 表示帧的长度；参数 payload 表示负载，可以是 NULL；参数 payload_s 表示负载的长度，可以是 0；参数 l 表示 Libnet 句柄；参数 ptag 表示协议标记。

此函数用来构造一个 IEEE 802.1x 协议头。

（3）libnet_ptag_t libnet_build_802_2(u_int8_t dsap, u_int8_t ssap, u_int8_t control, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 dsap 表示目的服务连接点，参数 ssap 表示源服务连接点，参数 control 表示控制区域，参数 payload 表示负载，参数 payload_s 表示负载的长度，参数 l 表示 Libnet 句柄，参数 ptag 表示一个协议标记。

此函数创建一个 IEEE 802.2 LLC 协议头。

（4）libnet_ptag_t libnet_build_802_2snap(u_int8_t dsap, u_int8_t ssap, u_int8_t control, u_int8_t *oui, u_int16_t type, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

此函数构造一个 IEEE 802.2 LLC SNAP 协议头。

参数描述：参数 dsap 表示目的服务连接点，参数 ssap 表示源服务连接点，参数 control 表示控制区域，参数 oui 表示组织惟一标记，参数 type 表示上层协议类型，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

（5）libnet_ptag_t libnet_build_802_3(u_int8_t *dst, u_int8_t *src, u_int16_t len, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 dst 表示目的以太网地址，参数 src 表示源以太网地址，参数 len 表示帧长度，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个 IEEE 802.3 协议头。调用此函数之前，必须在 libnet_init()函数调用中使用 LIBNET_LINK 类型。

（6）libnet_ptag_t libnet_build_ethernet(u_int8_t *dst, u_int8_t *src, u_int16_t type, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 dst 表示目的以太网地址，参数 src 表示源以太网地址，参数 type 表示上层协议类型，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个以太网头。注意在 libnet_init（）函数中使用 LIBNET_LINK 接口类型。

（7）libnet_ptag_t libnet_autobuild_ethernet(u_int8_t *dst, u_int16_t type, libnet_t *l)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 dst 表示目的以太网地址，参数 type 表示上层协议类型，参数 l 表示 Libnet 句柄。

此函数自动创建一个以太网协议头。注意此函数是自动创建的，以 auto 来标记，协议的其他数据都是由 Libnet 自动添加的，这样就使构造数据包更快捷、更容易，但其灵活性减小。

（8）libnet_ptag_t libnet_build_fddi(u_int8_t fc, u_int8_t *dst, u_int8_t *src, u_int8_t dsap,u_int8_t ssap, u_int8_t cf, u_int8_t *oui, u_int16_t type, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 cf 表示种类版本和优先级，参数 dst 表示目的 FDDI 地址，参数 src 表示源 FDDI 地址，参数 dsap 表示目的服务连接点，参数 ssap 表示源服务连接点，参数 cf 表示 cf，参数 oui 表示 IEEE 组织代码，参数 type 表示上层协议类型，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个 FDDI 协议头。

（9）libnet_ptag_t libnet_autobuild_fddi(u_int8_t fc, u_int8_t *dst, u_int8_t dsap, u_int8_t ssap, u_int8_t cf, u_int8_t *oui, u_int16_t type, libnet_t *l)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 fc 表示种类形式和优先级，参数 dst 表示目的 FDDI 地址，参数 ssap 表示源服务连接点，参数 cf 表示 cf，参数 oui 表示 IEEE 组织代码，参数 type 表示上层协议类型，参数 l 表示 Libnet 句柄。

此函数自动构造一个 FDDI 协议头。

（10）libnet_ptag_t libnet_build_arp(u_int16_t hrd, u_int16_t pro, u_int8_t hln, u_int8_t pln, u_int16_t op, u_int8_t *sha, u_int8_t *spa, u_int8_t *tha, u_int8_t *tpa, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 hrd 表示硬件地址形式，参数 pro 表示协议地址形式，参数 hln 表示硬件地址长度，参数 pln 表示协议地址长度，参数 op 表示 ARP 操作类型，参数 sha 表示发送方的硬件地址，参数 spa 表示发送方的协议地址，参数 tha 表示目标硬件地址，参数 tpa 表示目标协议地址，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个 ARP 协议头。

（11）libnet_ptag_t libnet_autobuild_arp(u_int16_t op, u_int8_t *sha, u_int8_t *spa, u_int8_t *tha, u_int8_t *tpa, libnet_t *l)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 op 表示 ARP 操作类型，参数 sha 表示发送方的硬件地址，参数 spa 表示发送方的协议地址，参数 tha 表示目标硬件地址，参数 tpa 表示目标协议地址，参数 l 表示 Libnet 句柄。

此函数自动构造一个 ARP 协议头。

（12）libnet_ptag_t libnet_build_tcp(u_int16_t sp, u_int16_t dp, u_int32_t seq, u_int32_t ack, u_int8_t control, u_int16_t win, u_int16_t sum, u_int16_t urg, u_int16_t len, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 sp 表示源端口，参数 dp 表示目的端口，参数 seq 表示序列码，参数 ack 表示确认码，参数 control 表示控制标记，参数 win 表示窗口大小，参数 sum 表示校验和，参数 urg 表示紧急指针，参数 len 表示整个数据包的长度，参数 payload 表示负载，参

数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个 TCP 协议头。

（13）libnet_ptag_t libnet_build_tcp_options(u_int8_t *options, u_int32_t options_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 options 表示 TCP 选项字符串，参数 options 表示选项字符串的长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造 TCP 协议选项数据。

（14）libnet_ptag_t libnet_build_udp(u_int16_t sp, u_int16_t dp, u_int16_t len, u_int16_t sum, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 sp 表示源端口，参数 dp 表示目的端口，参数 len 表示整个数据包的长度，参数 sum 表示校验和，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个 UDP 协议头。

（15）libnet_ptag_t libnet_build_cdp(u_int8_t version, u_int8_t ttl, u_int16_t sum, u_int16_t type, u_int16_t len, u_int8_t *value, u_int8_t *payload, u_int32_t payload_s,libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 version 表示版本，参数 ttl 表示生存时间，参数 sum 表示校验和，参数 type 表示参数 value 数据类型，参数 len 表示参数 value 的长度，参数 value 表示 CDP 信息字符串，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个 CDP 协议头。

（16）libnet_ptag_t libnet_build_icmpv4_echo(u_int8_t type, u_int8_t code, u_int16_t sum, u_int16_t id, u_int16_t seq, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 type 表示 ICMP 数据包类型，这里应该是 ICMP_ECHOREPLY 或 ICMP_ECHO；参数 code 表示 ICMP 数据包代码，这里应该是 0；参数 sum 表示校验和；参数 id 表示 ID 号；参数 seq 表示序列码；参数 payload 表示负载；参数 payload_s 表示负载长度；参数 l 表示 Libnet 句柄；参数 ptag 表示协议标记。

此函数构造一个 ICMPv4 查询或应答的协议头。

（17）libnet_ptag_t libnet_build_icmpv4_mask(u_int8_t type, u_int8_t code, u_int16_t sum, u_int16_t id, u_int16_t seq, u_int32_t mask, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果

失败，就返回－1。

参数描述：参数 type 表示 ICMP 数据包类型，在这里应该是 ICMP_MASKREQ 或 ICMP_MASKREPLY；参数 code 表示 ICMP 数据包代码，在这里应该是 0；参数 sum 表示校验和；参数 id 表示 ID 号；参数 seq 表示序列码；参数 mask 表示 32 位子网掩码；参数 payload 表示负载；参数 payload_s 表示负载长度；参数 l 表示 Libnet 句柄；参数 ptag 表示协议标记。

此函数构造一个 ICMPv4 地址掩码请求或应答协议头。

（18）libnet_ptag_t libnet_build_icmpv4_unreach(u_int8_t type, u_int8_t code, u_int16_t sum, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 type 表示 ICMP 数据包类型，在这里应该是 ICMP_UNREACH；参数 code 表示 ICMP 数据包代码，在这里应该是 0～15 中的一个数，0 代表网络不可达，1 代表主机不可达，2 代表协议不可达，3 代表端口不可达，4 代表需要进行分片但设置了不分片比特，5 代表源站选路失败，6 代表目的网络不认识，7 代表目的主机不认识，8 代表源主机被隔离，9 代表目的网络被强制禁止，10 代表目的主机被强制禁止，11 代表由于服务类型（TOS）而使网络不可达，12 代表由于服务类型 TOS 而使主机不可达，13 代表由于过滤而使通信被强制禁止，14 代表主机越权，15 代表优先权中止生效；参数 sum 表示校验和；参数 payload 表示负载；参数 payload_s 表示负载长度；参数 l 表示 Libnet 句柄；参数 ptag 表示协议标记。

此函数构造一个 ICMPv4 不可达协议头。

（19）libnet_ptag_t libnet_build_icmpv4_redirect(u_int8_t type, u_int8_t code, u_int16_t sum, u_int32_t gateway, u_int8_t *payload, u_int32_t payload_s, libnet_t *l,libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 type 表示 ICMP 数据包类型，在这里应该是 ICMP_REDIRECT，其值为 5，在前面的数据结构描述中进行了定义；参数 code 表示 ICMP 协议代码，在这里其值应该为从 0～3 中的一个数，0 代表对网络重定向，1 代表对主机重定向，2 代表对服务类型和网络重定向，3 代表对服务类型和主机重定向；参数 sum 表示校验和；参数 gateway 表示网关；参数 payload 表示负载；参数 payload_s 表示负载长度；参数 l 表示 Libnet 句柄；参数 ptag 表示协议标记。

此函数构造一个 ICMP 重定向协议头。

（20）libnet_ptag_t libnet_build_icmpv4_timeexceed(u_int8_t type, u_int8_t code, u_int16_t sum, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 type 表示 ICMP 数据包类型，在这里应该是 ICMP_TIMXCEED，其值是 11，在上面的数据结构的说明部分中有定义；参数 code 表示 ICMP 数据包代码，在这里应该是 0 或 1，0 代表传输期间生存时间为 0，1 代表在数据报组装期间生存时间为 0；参数 sum 表示校验和；参数 payload 表示负载；参数 payload_s 表示负载长度；参数 l 表示 Libnet

句柄；参数 ptag 表示协议标记。

此函数构造一个 ICMPv4 超时协议头。

（21）libnet_ptag_t libnet_build_icmpv4_timestamp(u_int8_t type, u_int8_t code, u_int16_t sum, u_int16_t id, u_int16_t seq, n_time otime, n_time rtime, n_time ttime,u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 type 表示 ICMP 数据包类型，在这里应该是 ICMP_TSTAMP 或 ICMP_TSTAMPREPLY，ICMP_TSTAMP 的值为 13，代表时间戳请求，ICMP_TSTAMPREPLY 的值为 14，代表时间戳应答；参数 code 表示 ICMP 数据包代码，在这里应该是 0；参数 sum 表示校验和；参数 id 表示 ID 号；参数 seq 表示序列号；参数 otime 表示发起时间戳；参数 rtime 表示接收时间戳；参数 ttime 表示传输时间戳；参数参数 payload 表示负载；参数 payload_s 表示负载长度；参数 l 表示 Libnet 句柄；参数 ptag 表示协议标记。

此函数构造一个 ICMPv4 时间戳查询或应答协议头。

（22）libnet_ptag_t libnet_build_igmp(u_int8_t type, u_int8_t code, u_int16_t sum, u_int32_t ip, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 type 表示数据包类型，参数 code 表示数据包代码，参数 sum 表示校验和，参数 ip 表示 IP 地址，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个 IGMP 协议头。

（23）libnet_ptag_t libnet_build_ipv4(u_int16_t len, u_int8_t tos, u_int16_t id, u_int16_t frag,u_int8_t ttl, u_int8_t prot, u_int16_t sum, u_int32_t src, u_int32_t dst,u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 len 表示整个 IP 数据包的长度，参数 tos 表示服务类型，参数 id 表示 IP 标识号，参数 frag 表示偏移，参数 ttl 表示生存时间，参数 prot 表示上层协议类型，参数 sum 表示校验和，参数 src 表示源 IP 地址，参数 dst 表示目的 IP 地址，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个 IPv4 协议头。

（24）libnet_ptag_t libnet_build_ipv4_options(u_int8_t *options, u_int32_t options_s, libnet_t *l,libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 options 表示 IP 选项的字符串，参数 options_s 表示 IP 选项字符串的长度。参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个 IPv4 协议选项头。

（25）libnet_ptag_t libnet_autobuild_ipv4(u_int16_t len, u_int8_t prot, u_int32_t dst, libnet_t *l)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 len 表示整个数据包的长度，参数 prot 表示上层协议类型，参数 dst 表示目的 IP 地址，参数 l 表示 Libnet 句柄。

此函数快速自动构造一个 IPv4 协议头。

（26）libnet_ptag_t libnet_build_ipv6(u_int8_t tc, u_int32_t fl, u_int16_t len, u_int8_t nh,u_int8_t hl, struct libnet_in6_addr src, struct libnet_in6_addr dst, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 tc 表示事务类型，参数 fl 表示流标记，参数 len 表示整个 IP 数据包的长度，参数 nh 表示下一个头，参数 hl 表示跳跃限制，参数 src 表示源 IPv6 地址，参数 dst 表示目的 IPv6 地址，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个 IPv6 协议头。

（27）libnet_ptag_t libnet_build_ipv6_frag(u_int8_t nh, u_int8_t reserved, u_int16_t frag, u_int32_t id, u_int8_t *payload, u_int32_t payload_s, libnet_t *l,libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 nh 表示下一个协议头，参数 reserved 表示保留未用，参数 frag 表示碎片位数，参数 id 表示 ID 号，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个 IPv6 碎片协议头。

（28）libnet_ptag_t libnet_build_ipv6_routing(u_int8_t nh, u_int8_t len, u_int8_t rtype, u_int8_t segments, u_int8_t *payload, u_int32_t payload_s, libnet_t *l,libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 nh 表示下一个协议头，参数 len 表示长度，参数 rtype 表示路由协议头的类型，参数 segments 表示路由段的数目，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个 IPv6 路由协议头。

（29）libnet_ptag_t libnet_build_ipv6_destopts(u_int8_t nh, u_int8_t len, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 nh 表示下一个协议头，参数 len 表示长度，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个 IPv6 目的选项协议头。

（30）libnet_ptag_t libnet_build_ipv6_hbhopts(u_int8_t nh, u_int8_t len, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 nh 表示下一个协议头，参数 len 表示长度。参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个 IPv6 跳跃选项协议头。

（31）libnet_ptag_t libnet_autobuild_ipv6(u_int16_t len, u_int8_t nh, struct libnet_in6_addr dst, libnet_t *l)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 len 表示长度，参数 nh 表示下一个协议头，参数 dst 表示目的 IPv6 地址，参数 l 表示 Libnet 句柄。

此函数自动构造一个 IPv6 协议头。注意此函数在此版本中还没有实现，只是有声明。

（32）libnet_ptag_t libnet_build_isl(u_int8_t *dhost, u_int8_t type, u_int8_t user, u_int8_t *shost, u_int16_t len, u_iut8_t *snap, u_int16_t vid, u_int16_t index, u_int16_t reserved, u_int8_t *payload, u_int32_t payload_s, libnet_t *l,libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 dhost 表示目的地址，参数 type 表示帧类型，参数 user 表示用户定义的数据，参数 shost 表示源 MAC 地址，参数 len 表示总的长度，参数 snap 表示 SNAP 信息，参数 vid 表示 15 位 VLAN ID 和 1 位 BPDU 或 CDP 指示器，参数 index 表示端口索引号，参数 reserved 表示用在 FDDI 和令牌环中，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个 ISL 协议头。

（33）libnet_ptag_t libnet_build_ipsec_esp_hdr(u_int32_t spi, u_int32_t seq, u_int32_t iv, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 spi 表示安全参数索引，参数 seq 表示 ESP 序列码，参数 iv 表示初始向量，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个 IPSEC ESP 协议头。

（34）libnet_ptag_t libnet_build_ipsec_esp_ftr(u_int8_t len, u_int8_t nh, int8_t *auth,u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 len 表示填充的长度，参数 nh 表示下一个协议头，参数 auth 表示验证数据，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个 IPSEC ESP 页脚协议头。

（35）libnet_ptag_t libnet_build_ipsec_ah(u_int8_t nh, u_int8_t len, u_int16_t res,u_int32_t

spi, u_int32_t seq, u_int32_t auth, u_int8_t *payload,u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 nh 表示下一个协议头，参数 len 表示负载长度，参数 res 表示保留未用，参数 spi 表示安全参数索引，参数 seq 表示序列码，参数 auth 表示验证数据，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个 IPSEC 验证协议头。

（36）libnet_ptag_t libnet_build_dnsv4(u_int16_t h_len, u_int16_t id, u_int16_t flags, u_int16_t num_q, u_int16_t num_anws_rr, u_int16_t num_auth_rr,u_int16_t num_addi_rr, u_int8_t *payload, u_int32_t payload_s, libnet_t *l,libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 h_len 表示首部长度，参数 id 表示数据包 ID，参数 flags 表示控制标记，参数 num_q 表示问题数目，参数 num_anws 表示回答资源记录的个数，参数 num_auth 表示授权资源记录个数，参数 num_addi 表示附加资源记录个数，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个 DNSv4 协议头。

（37）libnet_ptag_tlibnet_build_rip(u_int8_t cmd, u_int8_t version, u_int16_t rd, u_int16_t af,u_int16_t rt, u_int32_t addr, u_int32_t mask, u_int32_t next_hop,u_int32_t metric, u_int8_t *payload, u_int32_t payload_s, libnet_t *l,libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 cmd 表示命令；参数 version 表示版本；参数 rd 在版本 1 中表示 0，在版本 2 中表示路由区域；参数 af 表示地址簇；参数 rt 在版本 1 中表示 0，在版本 2 中表示路由标记；参数 addr 表示 IPv4 地址；参数 mask 在版本 1 中表示 0，在版本 2 中表示子网掩码；参数 next_hop 在版本 1 中表示 0，在版本 2 中表示下一个跳地址；参数 metric 表示路由量度；参数 payload 表示负载；参数 payload_s 表示负载长度；参数 l 表示 Libnet 句柄；参数 ptag 表示协议标记。

此函数构造一个 RIP 协议头。

（38）libnet_ptag_t libnet_build_rpc_call(u_int32_t rm, u_int32_t xid, u_int32_t prog_num, u_int32_t prog_vers, u_int32_t procedure, u_int32_t cflavor, u_int32_t clength,u_int8_t *cdata, u_int32_t vflavor, u_int32_t vlength, u_int8_t *vdata,u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 rm 表示用来在流中标记位置的记录，参数 xid 表示用来连接呼叫和应答的事务标记，参数 prog_num 表示远程程序规格，参数 prog_vers 表示远程程序版本规格，参数 procedure 表示远程程序执行的过程，参数 cflavor 表示证明信任类型，参数 clength 表

示信任长度，参数 cdata 表示不透明的信任数据，参数 vflavor 表示证明者类型，参数 vlength 表示证明者长度，参数 vdata 表示不透明证明者数据，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个 RPCv2 呼叫信息协议头。

（39）libnet_ptag_t libnet_build_stp_conf(u_int16_t id, u_int8_t version, u_int8_t bpdu_type, u_int8_t flags, u_int8_t *root_id, u_int32_t root_pc, u_int8_t *bridge_id,u_int16_t port_id, u_int16_t message_age, u_int16_t max_age, u_int16_t hello_time, u_int16_t f_delay, u_int8_t *payload,u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 id 表示协议 ID，参数 version 表示协议版本号，参数 bpdu_type 表示桥协议数据单元类型，参数 flags 表示标记，参数 root_id 表示根 ID，参数 root_pc 表示根路径成本，参数 bridge_id 表示桥 ID，参数 port_id 表示端口 ID，参数 message_age 表示信息年龄，参数 max_age 表示最大年龄，参数 hello_time 表示询问时间，参数 f_delay 表示转发延迟，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个 IEEE 802.1d STP 配置协议头。

（40）libnet_ptag_t libnet_build_stp_tcn(u_int16_t id, u_int8_t version, u_int8_t bpdu_type, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 id 表示协议 ID，参数 version 表示协议版本号，参数 bpdu_type 表示桥协议数据单元类型，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个 IEEE 802.1d STP 拓扑变化通知协议头。

（41）libnet_ptag_t libnet_build_token_ring(u_int8_t ac, u_int8_t fc, u_int8_t *dst, u_int8_t *src,u_iut8_t dsap, u_int8_t ssap, u_int8_t cf, u_int8_t *oui, u_int16_t type,u_int8_t *payload, u_int32_t payload_s, libuet_t *l, libnet_ptag_t ptag);

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 ac 表示访问控制字段；参数 fc 表示帧控制字段，说明帧是包含数据信息还是控制信息；参数 dst 表示目标站地址；参数 src 表示源站地址；参数 dsap 表示目标服务访问点；参数 ssap 表示源服务访问点；参数 cf 表示控制区域；参数 oui 表示组织唯一标记；参数 type 表示上层协议类型；参数 payload 表示负载；参数 payload_s 表示负载长度；参数 l 表示 Libnet 句柄；参数 ptag 表示协议标记。

此函数构造一个令牌环协议头。

（42）libnet_ptag_t libnet_autobuild_token_ring(u_int8_t ac, u_int8_t fc, u_int8_t *dst, u_int8_t dsap, u_int8_t ssap, u_int8_t cf, u_iut8_t *oui, u_int16_t type,libnet_t *l)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 ac 表示访问控制，参数 fc 表示帧控制，参数 dst 表示目标地址，参数 dsap 表示目标服务访问点，参数 ssap 表示源服务访问点，参数 cf 表示控制域，参数 oui 表示组织惟一标记，参数 type 表示上层协议类型，参数 l 表示 Libnet 句柄。

此函数自动构造一个令牌环协议头。

（43）libnet_ptag_t libnet_build_vrrp(u_int8_t version, u_int8_t type, u_int8_t vrouter_id, u_int8_t priority, u_int8_t ip_count, u_int8_t auth_type, u_int8_t advert_int,u_int16_t sum, u_int8_t *payload, u_int32_t payload_s, libnet_t *l,libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 version 表示版本号；参数 type 表示该 VRRP 数据包的类型，在该版本的协议中数据包类型只有一种：1 ADVERTISEMENT；参数 vrouter_id 表示虚拟路由标识号，此字段根据数据包报告状况识别虚拟路由器；参数 priority 表示优先级，规定虚拟路由器发送 VRRP 路由器的优先级；虚拟路由器在分配 VRRP 路由器时必须使用优先级值：1 到 254 之间；参数 ip_count 表示 IP 地址数目；参数 auth_type 表示认证类型；参数 advert_int 表示广告间的时间间隔；参数 sum 表示校验和，16 位，用于检测 VRRP 信息中的数据破坏情况；参数 payload 表示负载；参数 payload_s 表示负载长度；参数 l 表示 Libnet 句柄；参数 ptag 表示协议标记。

此函数构造一个虚拟路由器冗余协议 VRRP 协议头。

（44）libnet_ptag_t libnet_build_mpls(u_int32_t label, u_int8_t experimental, u_int8_t bos, u_int8_t ttl, u_int8_t *payload, u_int32_t payload_s, libnet_t *l,libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 label 表示标记值，参数 experimental 表示试用、保留，参数 bos 表示栈底标记，参数 ttl 表示生存时间，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个多协议标签交换（MPLS）协议头。

（45）libnet_ptag_tlibnet_build_ntp(u_int8_t leap_indicator, u_int8_t version, u_int8_t mode, u_int8_t stratum, u_int8_t poll, u_int8_t precision, u_int16_t delay_int,u_int16_t delay_frac, u_int16_t dispersion_int, u_int16_t dispersion_frac,u_int32_t reference_id, u_int32_t ref_ts_int, u_int32_t ref_ts_frac,u_int32_t orig_ts_int, u_int32_t orig_ts_frac, u_int32_t rec_ts_int,u_int32_t rec_ts_frac, u_int32_t xmt_ts_int, n_int32_t xmt_ts_frac,u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 leap_indicator 表示跳跃指示器，参数 version 表示版本号，参数 mode 表示模式，参数 stratum 表示对本地时钟级别的整体识别，参数 poll 表示连续信息间的最大间隔，参数 precision 表示本地时钟精确度，参数 delay_int 表示延迟间隔，参数 delay_frac 表示延迟的片断，参数 dispersion_int 表示传播的间隔，参数 dispersion_frac 表示传播的片断，参数 reference_id 表示参照 ID，参数 ref_ts_int 表示参考时间戳间隔，参数 ref_ts 表示参考时间戳片断，参数 orig_ts_int 表示发起时间戳间隔，参数 orig_ts_trac 表示发起时间戳片断，参

数 rec_ts_int 表示接收时间戳间隔，参数 rec_ts_frac 表示接收时间戳片断，参数 xmt_ts_int 表示传输时间戳间隔，参数 xmt_ts_frac 表示传输时间戳片断，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个网络时间协议（NTP）协议头。

（46）libnet_ptag_t libnet_build_ospfv2(u_int16_t len, u_int8_t type, u_int32_t rtr_id, u_int32_t area_id, u_int16_t sum, u_int16_t autype, u_int8_t *payload,u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 len 表示长度，参数 type 表示类型，参数 rtr_id 表示路由器 ID，参数 area_id 表示识别数据包归属区域，参数 sum 表示校验和，参数 autype 表示认证模式，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个开放最短路径优先（OSPF）协议头。

（47）libnet_ptag_t libnet_build_ospfv2_hello(u_int32_t netmask, u_int16_t interval, u_int8_t opts,u_int8_t priority, u_int dead_int, u_int32_t des_rtr, u_int32_t bkup_rtr,u_int32_t neighbor, u_int8_t *payload, u_int32_t payload_s, libnet_t *l,libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 netmask 表示子网掩码，参数 interval 表示间隔，参数 opts 表示选项，参数 priority 表示优先级，参数 dead_int 表示路由器消失间隔，参数 dst_rtr 表示指定的路由器，参数 bkup_rtr 表示备份路由器，参数 neighbour 表示邻居，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个开放最短路径优先（OSPF）查询协议头。

（48）libnet_ptag_t libnet_build_ospfv2_dbd(u_int16_t dgram_len, u_int8_t opts, u_int8_t type, u_int seqnum, u_int8_t *payload, u_int32_t payload_s, libnet_t *l,libnet_ptag_t ptag);

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 dgram_len 表示报文长度，参数 opts 表示选项，参数 type 表示类型，参数 seqnum 表示序列码。

此函数构造一个 OSPF dbd 协议头。

（49）libnet_ptag_t libnet_build_ospfv2_lsr(u_int type, u_int lsid, u_int32_t advrtr,u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag);

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 type 表示类型，参数 lsid 表示链路状态 ID，参数 advrtr 表示广告路由器，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个 OSPF 链路状态查询协议头。

（50）libnet_ptag_t libnet_build_ospfv2_lsu(u_int num, u_int8_t *payload, u_int32_t payload_s,

libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 num 表示个数，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个 OSPF 链路状态更新协议头。

（51）libnet_ptag_t libnet_build_ospfv2_lsa(u_int16_t age, u_int8_t opts, u_int8_t type,u_int lsid, u_int32_t advrtr, u_int seqnum, u_int16_t sum, u_int16_t len,u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag);

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 age 表示 LS 年龄，参数 opts 表示选项，参数 type 表示 LSA 类型，参数 lsid 表示连接状态 ID 号，参数 advrtr 表示广告路由器，参数 seqnum 表示链路状态序列码，参数 sum 表示校验和，参数 len 表示长度，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个 OSPF 链路状态确认协议头。

（52）libnet_ptag_t libnet_build_ospfv2_lsa_rtr(u_int16_t flags, u_int16_t num, u_int id,u_int data, u_int8_t type, u_int8_t tos, u_int16_t metric, u_int8_t *payload,u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag);

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 flags 表示标志，参数 num 表示数目，参数 id 表示 ID 号，参数 type 表示类型，参数 tos 表示服务质量，参数 metric 表示量度，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libuet 句柄，参数 ptag 表示协议标记。

此函数构造一个 OSPF 链路状态确认路由器协议头，是第 1 类 LSA。

（53）libnet_ptag_t libnet_build_ospfv2_lsa_net(u_int32_t nmask, u_int rtrid, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag);

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 nmask 表示网络掩码，参数 rtrid 表示路由器 ID，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个 OSPF 链路状态确认网络协议头，是第 2 类 LSA。

（54）libnet_ptag_t libnet_build_ospfv2_lsa_sum(u_int32_t nmask, u_int metric, u_int tos, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag);

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 nmask 表示网络掩码，参数 metric 表示量度，参数 tos 表示服务质量，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个 OSPF 链路状态确认摘要协议头，是第 3 类和第 4 类 LSA。

（55）libnet_ptag_t libnet_build_ospfv2_lsa_as(u_int32_t nmask, u_int metric, u_int32_t fwdaddr, u_int tag, u_int8_t *payload, u_int32_t payload_s, libnet_t *l,libnet_ptag_t ptag);

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 nmask 表示网络掩码，参数 metric 表示量度，参数 fwdaddr 表示转发地址，参数 tag 表示外部路由标记，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄。参数 ptag 表示协议标记。

此函数构造一个 OSPF AS-external-LSAs 协议头，是第 5 类 LSA。

（56）libnet_ptag_t libnet_build_data(u_int8_t *payload, u_int32_t payload_s, libuet_t *l,libnet_ptag_t ptag);

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造任意一个数据包。

（57）libnet_ptag_t libnet_build_dhcpv4(u_int8_t opcode, u_int8_t htype, u_int8_t hlen, u_int8_t hopcount, u_int32_t xid, u_int16_t secs, u_int16_t flags,u_int32_t cip, u_int32_t yip, u_int32_t sip, u_int32_t gip, u_int8_t *chaddr,u_int8_t *sname, u_int8_t *file, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 opcode 表示消息操作代码；参数 htype 表示硬件地址类型；参数 hlen 表示硬件地址长度；参数 hopcount 表示跳跃数；参数 xid 表示处理 ID；参数 secs 表示客户机地址获取，进程恢复消耗的时刻；参数 flags 表示标记；参数 cip 表示客户机 IP 地址；参数 yip 表示本机 IP 地址；参数 sip 表示服务器地址；参数 gip 表示代理 IP 地址；参数 chaddr 表示客户机硬件；参数 sname 表示服务器主机名字；参数 file 表示文件名；参数 payload 表示负载；参数 payload_s 表示负载长度；参数 l 表示 Libnet 句柄；参数 ptag 表示协议标记。

此函数构造一个动态主机配置协议 DHCPv4 的协议头。

（58）libnet_ptag_t libnet_build_bootpv4(u_int8_t opcode, u_int8_t htype, u_int8_t hlen, u_int8_t hopcount, u_int32_t xid, u_int16_t secs, u_int16_t flags,u_int32_t cip, u_int32_t yip, u_int32_t sip, u_int32_t gip, u_int8_t *cbaddr,u_int8_t *sname, u_int8_t *file, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag);

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 opcode 表示操作码；参数 htype 表示硬件地址类型；参数 hleu 表示硬件地址长度；参数 hopcount 表示跳跃数目；参数 xid 表示传输 ID；参数 secs 表示客户机地址获取，进程恢复消耗的时刻；参数 flags 表示标记；参数 cip 表示客户机 IP 地址；参数 yip 表示本机 IP 地址；参数 sip 表示服务器 IP 地址；参数 gip 表示网关 IP 地址；参数 chaddr 表示客户机硬件；参数 sname 表示服务器主机名字；参数 file 表示文件名字；参数 payload 表示负载；参数 payload_s 表示负载长度；参数 l 表示 Libnet 句柄；参数 ptag 表示协议标记。

此函数构造一个引导协议 BOOTPv4 的协议头。

（59）inline u_int32_t libnet_getgre_length(u_int16_t fv);

函数返回值：返回 GRE 协议长度。

参数描述：参数 fv 表示参数 fv 表示 16 位信息，其中 0～7 位表示校验和、序列号等，8～12 位表示标记，13～15 位表示版本号。

此函数返回 GRE 协议长度。

（60）libnet_ptag_t libnet_build_gre(u_int16_t fv, u_int16_t type, u_int16_t sum,u_int16_t offset, u_int32_t key, u_int32_t seq, u_int16_t len, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag);

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 fv 表示 16 位信息，其中 0～7 位表示校验和、序列号等，8～12 位表示标记，13～15 位表示版本号；参数 type 表示协议类型；参数 sum 表示校验和；参数 offset 表示偏移；摧残数 key 表示关键码；参数 seq 表示序列号；参数 len 表示长度；参数 payload 表示负载；参数 payload_s 表示负载长度；参数 l 表示 Libnet 句柄；参数 ptag 表示协议标记。

此函数构造一个通用路由封装 GRE 协议头。

（61）libnet_ptag_t libnet_build_egre(u_int16_t fv, u_int16_t type, u_int16_t sum,u_int16_t offset, u_int32_t key, u_int32_t seq, u_int16_t len,u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 fv 表示 16 位信息，其中 0～7 位表示校验和，序列号等，8～12 位表示标记，13～15 位表示版本号；参数 type 表示类型；参数 sum 表示校验和；参数 offset 表示偏移；参数 key 表示关键码；参数 seq 表示序列号；参数 len 表示长度；参数 payload 表示负载；参数 payload_s 表示负载长度；参数 l 表示 Libnet 句柄；参数 ptag 表示协议标记。

此函数构造一个 EGRE 协议头。

（62）libnet_ptag_t libnet_build_gre_sre(u_int16_t af, u_int8_t offset, u_int8_t length,u_int8_t *routing, u_int8_t *payload, u_int32_t payload_s, libnet_t *l,libnet_ptag_t ptag);

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 af 表示地址族，参数 offset 表示偏移，参数 length 表示长度，参数 routing 表示路由信息，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个通用路由封装源路由条目 GRE SRE 协议头。

（63）libnet_ptag_t libnet_build_gre_last_sre(libnet_t *l, libnet_ptag_t ptag);

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个 GRE LAST SRE 协议头。

（64）libnet_ptag_t

libnet_build_bgp4_header(u_int8_t marker[LIBNET_BGP4_MARKER_SIZE], u_int16_t len, u_int8_t type, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag);

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 marker 表示包含信息接收端可预测值；参数 len 表示信息长度；参数 type 表示信息类型，在这里可以是 OPEN, UPDATE, NOTIFICATION 或 KEEPALIVE。参数 payload 表示负载；参数 payload_s 表示负载长度；参数 l 表示 Libnet 句柄；参数 ptag 表示协议标记。

此函数构造一个边界网关协议 BGP4 的协议头。

（65）libnet_ptag_t libnet_build_bgp4_open(u_int8_t version, u_int16_t src_as, u_int16_t hold_time, u_int32_t bgp_id, u_int8_t opt_len, u_int8_t *payload, u_int32_t payload_s,libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 version 表示版本，参数 src_as 发送方的自治系统，参数 hold_time 表示持续时间，参数 bgp_id 表示 BGP 标识符，参数 opt_len 表示选项参数的长度，表示参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个边界网关协议 BGP OPEN 协议头。

（66）libnet_ptag_t libnet_build_bgp4_update(u_int16_t unfeasible_rt_len,u_int8_t *withdrawn_rt, u_int16_t total_path_attr_len, u_int8_t *path_attributes, u_int16_t info_len,u_int8_t *reachability_info, u_int8_t *payload, u_int32_t payload_s,libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 unfeasible_rt_len 表示不可行的路由长度，参数 withdrawn_rt 表示偏僻路由，参数 total_path_attr_len 表示整个路径属性长度，参数 path_attributes 表示路径属性，参数 info_len 表示网络层可达性信息长度，参数 reachability_info 表示网络层可达性信息，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个边界网关协议 BGP4 UPDATE 的协议头。

（67）libnet_ptag_t libnet_build_bgp4_notification(u_int8_t err_code, u_int8_t err_subcode, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag);

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回－1。

参数描述：参数 err_code 表示错误代码，参数 err_subcode 表示错误子代码，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个边界网关协议 BGP4 的通知协议头。

（68）libnet_ptag_t libnet_build_sebek(u_int32_t magic, u_int16_t version, u_int16_t type,

u_int32_t counter, u_int32_t time_sec, u_int32_t time_usec, u_int32_t pid,u_int32_t uid, u_int32_t fd, u_int8_t cmd[SEBEK_CMD_LENGTH], u_int32_t length, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag)

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 magic 表示识别隐藏数据包；参数 version 表示版本号；参数 type 表示记录类型，读数据是类型 0，写数据是类型 1；参数 counter 表示用来识别丢失数据包的 PDU 个数；参数 time_sec 表示 EPOCH 后的秒数；参数 time_usec 表示剩余微秒数；参数 pid 表示 PID；参数 uid 表示 UID；参数 fd 表示 FD；参数 cmd 表示命令；参数 length 表示 PDU 主体的长度；参数 payload 表示负载；参数 payload_s 表示负载长度；参数 l 表示 Libnet 句柄；参数 ptag 表示协议标记。

此函数构造一个 SEBEK 协议头。

（69）libnet_ptag_t libnet_build_link(u_int8_t *dst, u_int8_t *src, u_int8_t *oui, u_int16_t type, u_int8_t *payload, u_int32_t payload_s, libnet_t *l, libnet_ptag_t ptag);

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 dst 表示目的 MAC 地址，参数 src 表示源 MAC 地址，参数 oui 表示组织唯一标识，参数 type 表示上层协议类型，参数 payload 表示负载，参数 payload_s 表示负载长度，参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数构造一个链路层协议头，支持以太网和令牌环类型。

（70）libnet_ptag_t libnet_autobuild_link(u_int8_t *dst, u_int8_t *oui, u_int16_t type,libnet_t *l);

函数返回值：返回一个协议标记，此协议标记用来标识用此函数创建的协议数据。如果失败，就返回-1。

参数描述：参数 dst 表示目的 MAC 地址，参数 oui 表示组织惟一标识，参数 type 表示上层协议类型，参数 l 表示 Libnet 句柄。

此函数自动构造一个链路层协议头，支持以太网和令牌环协议类型。

5.4.4 数据包发送函数

上面介绍了各种数据包的构造函数。在完成数据包构造函数的调用之后，就可以使用数据包发送函数来发送败据包。发送数据包的函数只有一个，所以构造的任何协议数据包都使用同一个数据包发送函数，它们的发送操作都是一样的。

数据包发送函数的定义如下：

```
int libnet_write(libnet_t *l);
```

函数返回值：返回一个整型数值。

参数描述：参数 l 表示 Libnet 句柄。

此函数发送一个数据包，此数据包由 Libnet 句柄 l 来指示。

5.4.5 高级处理函数

高级处理函数是 Libnet 中用 adv 标记的函数。

函数会调用失败。

（4）int8_t * libnet_cq_getlabel(libnet_t *l)

函数返回值：返回一个 Libnet 句柄标识。

参数描述：参数 l 表示 Libnet 句柄。

此函数根据参数 l 返回一个 Libnet 句柄标识。

（5）libnet_t * libnet_cq_find_by_label(char *label)

函数返回值：返回一个 Libnet 句柄。如果失败，就返回 NULL。

参数描述：参数 label 表示需要查找的 Libnet 句柄标识。

此函数根据 label 表示的 Libnet 句柄标识从 Libnet 句柄队列中进行查找。

（6）void libnet_cq_destroy()

函数返回值：无。

参数描述：无。

此函数撤销整个 Libnet 句柄队列。在这个函数中会调用 libnet_destroy()函数来撤销所有的 Libnet 句柄。

（7）libnet_t * libnet_cq_head()

函数返回值：返回一个 Libnet 句柄。

参数描述：无。

此函数从 Libnet 队列中返回第一个 Libnet 句柄。

（8）int libnet_cq_last()

函数返回值：如果到达 Libnet 句柄的末尾，就返回 1；否则，返回 0。

参数描述：无

此函数判断是否在 Libnet 句柄队列的末尾。如果是，就返回 1；否则，返回 0。

（9）libnet_t * libnet_cq_next()

函数返回值：返回一个 Libnet 句柄。

参数描述：无。

此函数返回在 Libnet 句柄队列中的下一个 Libnet 句柄。

（10）u_int32_t libnet_cq_size()

函数返回值：返回 Libnet 句柄的个数。

参数描述：无。

此函数求得整个 Libnet 句柄队列中所有 Libnet 句柄的个数。

5.4.7 辅助函数

（1）void libnet_stats(libnet_t *l, struct libnet_stats *ls);

函数返回值：无。

参数描述：参数 l 表示 Libnet 句柄，参数 ls 表示 Libnet 状态数据结构。

此函数是获得 Lihnet 状态的信息。

（2）int libnet_getfd(libnet_t *l);

函数返回值：返回文件句柄。

参数描述：参数 l 表示 Libnet 句柄。

此函数获得 Libnet 的文件句柄。

（3）int8_t * libnet_getdevice(libnet_t *l);

函数返回值：返回设备的规范名字。

参数描述：参数 l 表示 Libnet 句柄。

此函数获得 Libnet 设备的正式名字。

（4）u_int8_t * libnet_getpbuf(libnet_t *l, libnet_ptag_t ptag);

函数返回值：返回协议块缓冲内容。如果失败，就返回 NULL。

参数描述：参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数返回由 l 和 ptag 指示的协议块缓冲区内容。

（5）u_int32_t libnet_getpbuf_size(libnet_t *l, libnet_ptag_t ptag);

函数返回值：返回一个协议块缓冲区的大小。

参数描述：参数 l 表示 Libnet 句柄，参数 ptag 表示协议标记。

此函数返回由 l 和 ptag 指定的协议块缓冲区的大小。

（6）char * libnet_geterror(libnet_t *l);

函数返回值：返回错误信息。如果无错误，就返回 NULL。

参数描述：参数 l 表示 Libnet 句柄。

此函数返回错误信息。

（7）u_int32_t libnet_getpacket_size(libnet_t *l);

函数返回值：返回所有协议块的长度之和。

参数描述：参数 l 表示 Libnet 句柄。

此函数返回由 l 指定的所有协议块的长度之和。

（8）int libnet_seed_prand(libnet_t *l);

函数返回值：成功就返回 1，失败就返回－1。

参数描述：参数 l 表示 Libnet 句柄。

此函数构造一个伪随机数生成器。

（9）u_int32_t libnet_get_prand(int mod);

函数返回值：返回一个伪随机数。

参数描述：参数 mod 表示产成伪随机数的模式。如果是 LIBNET_PR2，就表示 0 至 1；如果是 LIBNET_PR8，就表示 0 至 255；如果是 LIBNET_PR16，就表示 0 至 32767；如果是 LIBNET_Pru16，就表示 0 至 65535；如果是 LIBNET_PR32，就表示 0 至 2147483647；如果是 LIBNET_Pru32，就表示 0 至 4294967295。

此函数根据 mod 来构造一个伪随机数。

（10）int libnet_toggle_checksum(libnet_t *l, libnet_ptag_t ptag, int mode);

函数返回值：成功就返回 1，失败就返回－1。

参数描述：参数 l 表示 Libnet 句柄；参数 ptag 表示协议句柄；参数 mode 表示模式，可以是 LIBNET_ON 和 LIBNET_OFF。

此函数用来说明是否进行校验和计算。如果参数 mode 是 LIBNET_ON，在发送数据包之前就必须进行校验和计算；如果参数 mode 是 LIBNET_OFF，在发送数据包之前不用计算校验和；如果 Libnet 在初始化时使用的是 LIBNET_RAW4 模式，那么此函数不起作用，因为此时校验和都会在内核中计算，然后再把数据包发送出去。

（11）int libnet_plist_chain_new(libnet_t *l, libnet_plist_t **plist, cbar *token_list);

函数返回值：成功就返回 1，失败就返回－1。

参数描述：参数 l 表示 Libnet 句柄。参数 plist 表示端口列表，可以是 NULL；参数 token_list 表示原始端口列表字符串。

此函数构造一个新的端口列表。

（12）int libnet_plist_chain_next_pair(libnet_plist_t *plist, u_int16_t *bport, u_int16_t *eport);

函数返回值：成功就返回 1，失败就返回－1。

参数描述：参数 plist 表示先前构造的端口列表，参数 bport 表示开始的端口号或者 0，参数 eport 表示最后的端口号或者 0。

此函数返回下一个端口列表链。

（13）int libnet_plist_chain_dump(libnet_plist_t *plist);

函数返回值：成功就返回 1，失败就返回－1。

参数描述：参数 plist 表示先前产生的端口列表。

此函数输出所有端口列表中的所有端口号。

（14）char * libnet_plist_chain_dump_string(libnet_plist_t *plist);

函数返回值：返回一个端口列表的字符串。若失败，就返回 NULL。

参数描述：参数 plist 表示先前构造的端口列表。

此函数返回一个端口列表的字符串。

（15）int libnet_plist_chain_free(libnet_plist_t *plist);

函数返回值：成功就返回 1，失败就返回－1。

参数描述：参数 plist 表示先前构造的端口列表。

此函数释放所有端口列表的内存。

（16）void libnet_diag_dump_context(libnet_t *l);

函数返回值：无。

参数描述：参数 l 表示 Libnet 句柄。

此函数输出 Libnet 句柄 l 的内容。

（17）void libnet_diag_dump_pblock(libnet_t *l);

函数返回值：无。

参数描述：参数 l 表示 Libnet 句柄。

此函数输出所有协议块的内容。

（18）char * libnet_diag_dump_pblock_type(u_int8_t type);

函数返回值：返回一个协议块 pblock 类型名字的字符串。

参数描述：参数 type 表示协议块 pblock 类型

此函数返回由 type 指定的协议块的名字。

（19）void libnet_diag_dump_hex(u_int8_t *packet, u_int32_t len, int swap, FILE *stream);

函数返回值：无。

参数描述：参数 packet 表示数据包；参数 len 表示数据包的长度；参数 stream 表示流指针，如果是 1 就交换字节顺序，如果是 0 就不用交换字节顺序。

此函数输出参数 stream 流中的数据包的内容。

5.5 使用 Libnet 的开发流程

使用 Libnet 来开发程序都是遵循一定的步骤，一个完整的 Libnet 程序主要由以下几个步骤构成：

- 步骤一，对内存进行初始化；
- 步骤二，对网络进行初始化；
- 步骤三，构造各种网络数据包；
- 步骤四，对数据包进行合法性检验；
- 步骤五，发送网络数据包。

所以使用 Libnet 来开发程序是非常容易的，因为每个程序都是遵循这个步骤，只要掌握了它的开发规律，就很容易构造不同的应用程序。以构造并发送一个 TCP 包为例，Libnet 程序的开发流程如图 5-20 所示。

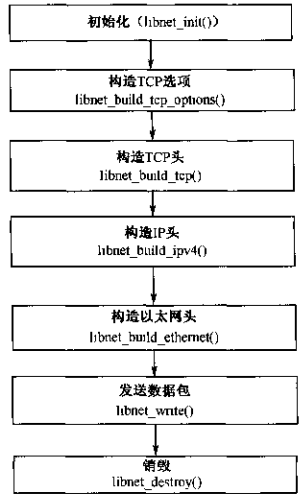

图 5-20 Libnet 程序的开发流程

5.5.1 初始化

一个 Libnet 的应用程序在开始时必须进行初始化操作，此功能由函数 libnet_init() 来实现。该函数不仅对内存进行初始化，也对网络进行初始化。

Libnet 初始化时完成以下任务。

1. 确定 Libnet 类型

在 Libnet 开发包中有 6 种 Libnet 类型，分别为 LIBNET_LINK，LIBNET_LINK_ADV，LIBNET_RAW4，LIBNET_RAW4_ADV，LIBNET_RAW6 和 LIBNET_RAW6_ADV。不同类型的 Libnet，其后面的操作也不同。

带有标记_ADV 的类型表示高级类型，在后面的操作中，必须使用高级处理函数来进行操作。例如：

（1）int libnet_adv_cull_packet(libnet_t *l, u_int8_t **packet, u_int32_t *packet_s)

此高级函数创建一个数据包。

（2）int libnet_adv_cull_header(libnet_t *l, libnet_ptag_t ptag, u_int8_t **header,u_int32_t *header_s)

此高级函数构造一个协议头。

（3）int libnet_adv_write_link(libnet_t *l, u_int8_t *packet, u_int32_t packet_s)

此函数在链路层向网络写一个数据包。

（4）void libnet_adv_free_packet(libnet_t *l, u_int8_t *packet)

此函数释放由函数 libnet_adv_cull_packet()开辟的内存。

类型为 LIBNET_LINK 的，表示构造的数据包是链路层开始的数据包，它构造的数据包的类型就比从 IP 层构造的数据包的类型更丰富、更灵活一些；类型为 LIBNET_RAW4 的，表示构造 IPv4 原始套接字接口类型的数据包；类型为 LIBNET_RAW6 的，表示构造 IPv6 原始套接字接口类型的数据包。

2. 确定网络接口

在此，网络接口是指数据包发送的接口。

指定网络接口有两种方式。一种方式是自己指定，可以是网络接口的名字，如以太网接口“eth0”等，也可以是 IP 地址；另一种方式就是由 Linbet 自动查询搜索，只要在函数 libnet_init()的参数 device 赋值 NULL 即可，此时 Libnet 会从系统中搜索可用的网络接口。

3. 获取 Libnet 句柄

在函数 libnet_init() 执行完后会返回一个 Libnet 句柄。Libnet 句柄的数据结构中包含了重要的信息，这些信息是后面很多函数都要用到的。Libnet 句柄的信息不需要开发者手工修改，它的内容都是在调用函数时被自动修改的，它与开发者不直接打交道。

Libnet 句柄的数据结构前面有说明，其定义如下：

```
struct libnet_context
{
#if ((__WIN32__) && !(__CYGWIN__))
   SOCKET fd;
   LPADAPTER  lpAdapter;
#else
   int fd;                                  /*数据包设备的文件描述符 */
#endif
   int injection_type;  /* 类型，例如原始套接字 IPv4 或 IPv6 或链路层协议*/
#define LIBNET_LINK          0x00          /* 链路层接口*/
#define LIBNET_RAW4          0x01          /*原始套接字接口 (IPv4) */
#define LIBNET_RAW6          0x02          /* 原始套接字接口 (IPv6) */
#define LIBNET_LINK_ADV      0x08          /*高级模式链路层 */
#define LIBNET_RAW4_ADV      0x09          /* 高级模式原始套接字 (IPv4) */
#define LIBNET_RAW6_ADV      0x0a          /*高级模式原始套接字(IPv6) */
```

```
#define LIBNET_ADV_MASK      0x08            /*检测是否是高级模式的掩码*/
    libnet_pblock_t *protocol_blocks;        /* 协议块首部或数据*/
    libnet_pblock_t *pblock_end;             /*链表的最后一个结点*/
    u_int32_t n_pblocks;                     /* 协议块的数目 */
    int link_type;                           /* 链路层类型*/
    int link offset;                         /* 链路层首部偏移*/
    int aligner;                             /*用来排列数据包*/
    char *device;                            /* 设备名字 */
    struct libnet_stats stats;               /* 统计分析 */
    libnet_ptag_t ptag_state;                /*pblock 标记状态 */
    char label[LIBNET_LABEL_SIZE];           /*句柄队列的文本标记 */
    char err_buf[LIBNET_ERRBUF_SIZE];        /* 错误信息 */
    u_int32_t total_size;                    /* 整体大小*/
};
typedef struct libnet_context libnet_t;
```

这里要注意的是怎样使用和理解 Libnet 句柄。Libnet 句柄可以说是最重要的一个数据结构，它惟一地描述了一个数据包的整个信息，通过它就可以了解所构造的数据包的信息。在每一个数据包构造函数中都有一个 Libnet 句柄参数，此参数就是存放数据包信息的。当要对数据包的信息进行修改时，可以通过函数对它进行修改。

参数 protocol_blocks 表示的是整个数据包的首地址。整个数据包通常是由几个协议块组成的，一个协议块是一个字节，由几个协议块组成一个双向的链表，而 protocol_blocks 就指向这个链表的第一个结点。

参数 pblock_end 表示这项协议块链表的最后一个结点。协议块链表是动态变化的，根据协议层来进行操作。首先插入的第一个结点，必须是最上层协议的结点。然后依次在从上到下进行协议块的构造和插入，最后一个结点必须是数据包的最下层协议。

5.5.2 构造数据包

构造一个数据包，就是构造一系列的协议块。一个数据包是由一个协议块构成的。

例如，如果想利用 Libnet 开发包构造一个 TCP 数据包，那么首先构造一个 TCP 协议块，然后插入协议块链表，再构造一个 IP 协议块，插入到 TCP 协议块的后面，最后构造一个以太网协议块，插入到 IP 协议块的后面。

假如一个数据包由三个协议块构成，那么此数据包在 Libnet 中的存储结构如图 5-21 所示。此存储结构是 Libnet 开发包中的核心内容。

一个 Libnet 句柄就代表一个数据包。其中，指针变量 protocol_blocks 指向协议块链表中的第一个协议块，第一个协议块也是最先构造的协议块，也就是此数据包中最上层协议的协议块；指针变量 pblock_end 指向协议块链表中的最后一个协议块，此协议块是最后构造的，并且是此数据包中底层协议的协议块。

例如，要从链路层构造一个 TCP 数据包，那么在图 5-21 中，第一个协议块就应该是 UDP 协议块，第二个协议块是 IP 协议块，第三个协议块是以太网协议块。先构造 TCP 协议块，然后再构造 IP 协议块，最后构造以太网协议块。

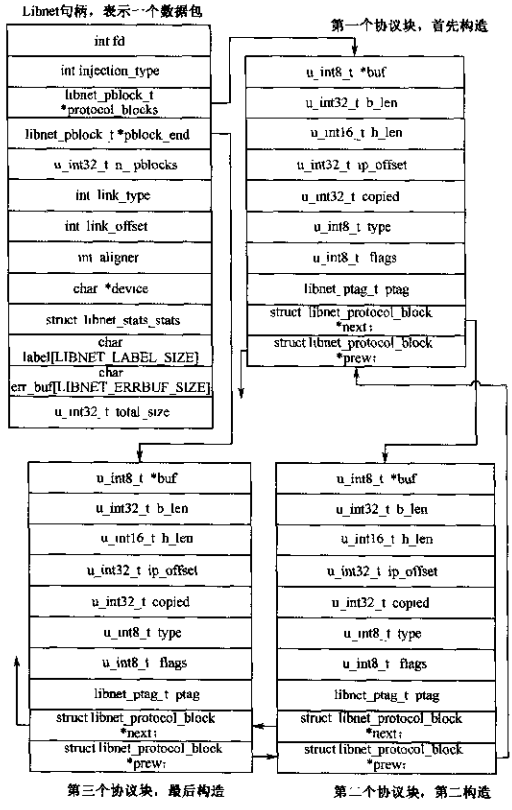

图 5-21　Libnet 中一个数据包的存储结构

5.5.3　数据包检测和发送

数据包的检测最主要的是计算校验和。在 Libnet 中，计算校验和的实现非常方便。

在构造数据包时，如果不想自己来计算校验和，只需要将校验和的参数赋值为 0 即可，Libnet 就会自动计算校验和。

发送数据包操作也非常简单，其功能主要由函数 libnet_write()函数完成。

在用一系列构造数据包的函数构造完一个数据包之后，就可以使用数据包发送函数 libnet_write()来完成发送的功能，此函数的参数就是一个数据包 Libnet 句柄。在 Libnet 中，无论构造什么样的数据包，其发送的过程都是一样的，都使用同一个函数。

5.6　使用 Libnet 开发包

前面对 Libnet 的数据结构和函数都进行了详细的说明，本节着重讨论怎样使用 Libnet

开发包来编写应用程序，这也是这一章的重点。这里主要通过一些典型的程序来进行讲解，然后对一些重要的议题进行专项论述。

本节所举的程序，其原则是尽量简单，只要能说明问题即可，与讨论问题无关的东西尽量缩减。例如，这些程序都没有考虑错误处理，也没有考虑程序的参数化问题，所使用的都是固定不变的值，不能通过程序的参数来改变。这些都是为了使程序尽可能简单，容易理解，因为在这里，举这些程序的宗旨只是为了清楚地说明某一个问题，让读者尽可能快地掌握 Libnet 的开发规律。当然，在实际的项目应用中应该考虑得更全面一些，如错误处理、灵活性、易用性、可移植性等。

在 Libnet 开发包中包含了很多有用的程序。在 Libnet 开发包发布时，其中自带了很多实验程序，这些程序说明了 Libnet 的用法。这些程序大部分是由 Libnet 的负责人 Mike D. Schiffman 写的，也有一些程序是由其他人写的。他们都对 Libnet 的结构了如指掌，所以研究这些程序是学习 Libnet 的非常好的一个途径，本章也对一部分程序进行了详细的讨论，使读者更容易看懂这些程序，包括其他在本书没讨论的程序。

5.6.1 构造 ARP 数据包

构造一个 ARP 数据包，首先必须了解 ARP 数据包的协议格式。ARP 协议格式在这里不详细说明，可以查阅相关资料。

构造 ARP 数据包的步骤如下：

（1）利用函数 libnet_init()进行初始化操作。因为 ARP 协议在网络层，所以必须先构造 ARP 协议块，也就是利用函数 libnet_build_arp() 来构造 ARP 协议块；

（2）利用函数 libnet_autobuild_Ethernet() 或 libnet_build_Ethernet() 来构造以太网协议块；

（3）使用函数 libnet_write() 发送数据包。

构造 ARP 数据包的流程如图 5-22 所示。

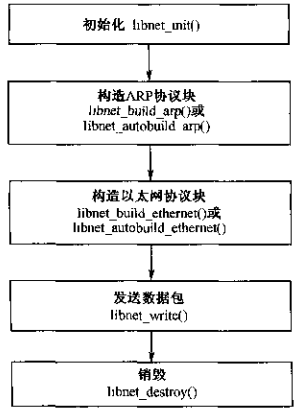

图 5-22 构造 ARP 数据包的流程

为了更能够清晰地说明怎样使用 Libnet 开发包来写程序，在这里先不考虑错误处理，写一个简单的构造 ARP 数据包的程序，尽量把程序写得简单一些。

首先讲一下在 Windows 平台下怎样开发一个使用 Libnet 开发包来构造和发送一个 ARP 数据包，其源程序如下：

```
#include <win32/libnet.h>
/*在 Windows 平台下必须包含此文件，文件 libnet.h 是 Libnet 开发包的头文件*/
void  main()
{
    int packet_size;                    /*存放数据包长度的变量*/
    libnet_t *l;                        /*Libnet 句柄*/
    libnet_ptag_t protocol_tag;         /*协议块标记*/
    char *device = NULL;                /*设备名字，此时为 NULL*/
    char error_information[LIBNET_ERRBUF_SIZE];     /*用来存放错误信息*/
    char *destination_ip_str = "192.168.0.2";
    /*目的 IP 地址字符串变量，可以指定任意一个合法的 IP 地址*/
    char *source_ip_str = "192.168.0.3";
    /*源 IP 地址字符串变量，可以指定任意一个合法的 IP 地址*/
    u_char hardware_source[6] = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06};
    /*源 MAC 地址，可以是任意指定*/
    u_char hardware_destination[6] = {0x06, 0x05, 0x04, 0x03, 0x02, 0x01};
    /*目的 MAC 地址，可以是任意指定*/
    u_long   destination_ip;                /*目的 IP 地址*/
    u_long  source_ip;                      /*源 IP 地址*/
    destination_ip=libnet_name2addr4(l,destination_ip_str,LIBNET_RESOLVE);
    /*把目的 IP 地址字符串形式转化成网络顺序字节形式的数据*/
    source_ip=libnet_name2addr4(l,source_ip_str,LIBNET_RESOLVE);
    /*把源 IP 地址字符串形式转化成网络顺序字节形式的数据*/
    l = libnet_init(                        /*初始化 Libnet*/
        LIBNET_LINK_ADV,                    /*Libnet 类型 */
        device,                             /*网络设备*/
        error_information);                 /*错误信息*/
    protocol_tag = libnet_build_arp(
        /*构造 ARP 协议块，函数的返回值是代表新生成的 ARP 协议块的一个协议块标记*/
            ARPHRD_ETHER,                   /*硬件地址类型，在这里是以太网*/
            ETHERTYPE_IP,                   /*协议地址类型，在这里是 IP 协议*/
            6,                              /*硬件地址长度，MAC 地址的长度为 6*/
            4,                              /*协议地址长度，IP 地址的长度为 4*/
            ARPOP_REPLY,                    /*操作类型，在这里是 ARP 应答类型*/
            hardware_source,                /*源硬件地址*/
            (u_int8_t *)&source_ip,         /*源 IP 地址*/
                hardware_destination,       /*目标硬件地址 */
            (u_int8_t *)&destination_ip,    /*目标协议地址*/
                NULL,                       /*负载，此时为 NULL*/
```

```
            0,                          /*负载的长度，此时为 0*/
            l,                          /*Libnet 句柄，此句柄由 libnet_init()函数生成*/
            0    /*协议块标记，此时为 0，表示构造一个新的 ARP 协议块，而不是修改已经存在的协议块*/
            );
            protocol_tag = libnet_autobuild_ethernet(
            /*构造一个以太网协议块，返回一个指向此协议块的标记*/
            hardware_destination,       /*目的硬件地址*/
            ETHERTYPE_ARP,              /*以太网上层协议类型，此时为 ARP 类型*/
            L                           /*Libnet 句柄*/
            );
    packet_size= libnet_write(l);
        /*发送已经构造的 ARP 数据包，注意此数据包应该包括两部分，一部分是 ARP 协议块，另外一部分是以太网
        协议块*/
    printf("发送一个%d 字节长度的 ARP 应答数据包\n",packet_size);
        /*输出发送的 ARP 数据包的字节数*/
    libnet_destroy(l);                  /*销毁 Libnet*/
}
```

此程序的功能是构造一个 ARP 应答数据包，其 MAC 源地址为 01:02:03:04:05:06，MAC 目的地址为 06:05:04:03:02:01，源 IP 地址为 192.168.0.3，目的 IP 地址为 192.168.0.2。

整个数据包的长度为 42 字节，其中 14 字节以太网协议块，28 字节 ARP 请求协议块。

下面着重讲一下对此程序的编译过程，并进行详细讲解，所有后面程序的编译过程与此是完全一样的，所以在后面就不再讲它们的编译过程。读者掌握了对此程序的编译过程，后面的程序就都迎刃而解了。

利用 Visual C++ 6.0 来打开此程序，然后进行编译，注意在编译的过程中，需要进行一些设置工作，首先必须设置 Visual C++ 的开发路径，包括 include 的路径和 library 的路径，具体步骤如下：

打开 Visual C++ 菜单 Tools 的子菜单 Options，打开 Options 对话框，在此对话框的 Derectories 页面进行设置，根据本机的 Libnet 的安装位置，分别添加 Libnet 的 include 路径为：

```
F:\LIBNET-1.1.2.1\LIBNET\INCLUDE
```

添加 Libnet 的 library 路径为：

```
F:\LIBNET-1.1.2.1\LIBNET\WIN32\DEBUG
```

注意，这些路径的设置是根据 Libnet 的安装位置而定的，Libnet 的安装位置不同，其路径也是不同的，这点要灵活处理。

然后再把 libnet.lib 库文件加入到 Visual C++ 开发环境中，具体步骤如下：

打开 Visual C++ 菜单 Project 的子菜单 Settings，打开 Project Settings 对话框，在此对话框的 Link 页面进行设置，在 Object/library modules 一项加入文件 libnet.lib。

此时就可以利用 Visual C++ 对此程序进行编译连接了。其运行结果如图 5-23 所示，表示构造和发送了一个 42 字节长度的 ARP 应答数据包。

使用抓包工具 Analyzer 可以清楚地看到发送的 ARP 数据包。Analyzer 工具可以在网站 http://analyzer.polito.it/下载。在 Analyzer 中设置要捕获的数据包过滤条件，如图 5-24 所示。

表示只捕获网络中的 ARP 协议类型的数据包。

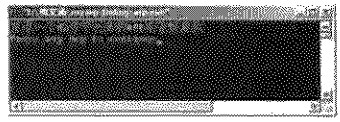

图 5-23　构造 ARP 程序的运行结果

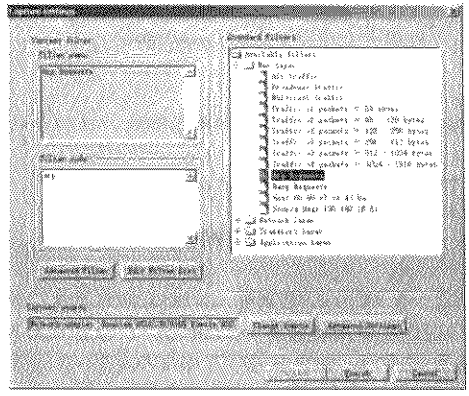

图 5-24　设定过滤规则

然后进行捕获，编译和运行上述源程序，可以看到 Analyzer 的运行结果如图 5-25 所示。

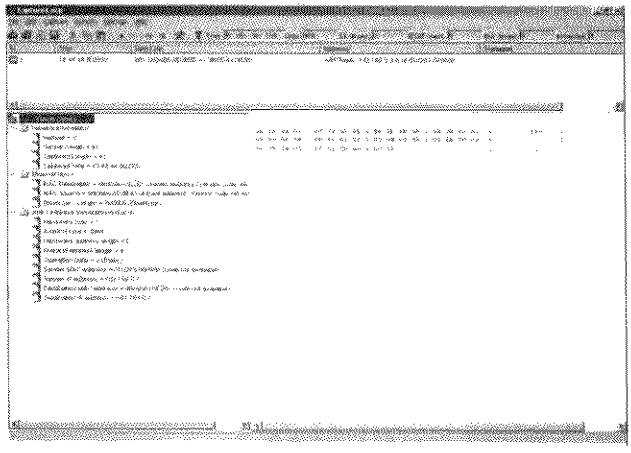

图 5-25　捕获 ARP 数据包的分析界面

ARP 应答数据包的详细内容，如图 5-26 所示。其中以太网协议块的内容和 ARP 协议块的内容都一目了然，而且与刚才构造的 ARP 数据包的内容完全一样，说明所构造的 ARP 数据包正是我们想构造的。

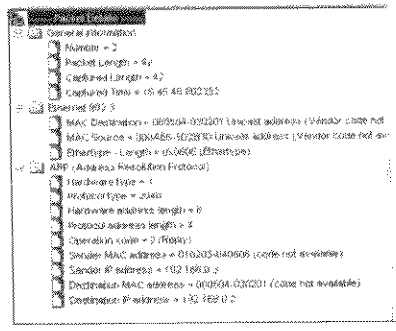

图 5-26　ARP 数据包的详细内容

在上面的程序中，没有考虑任何出错信息，但是这样写的程序，其问题是很多的。下面详细讲一下在 Libnet 开发包中的程序。首先详细讲述它的源程序，然后详细阐述对它的编译过程，最后分析它的运行结果。

注意，在这里要对这个 Libnet 开发包中程序的编译过程进行详细阐述，读者可以根据此过程来运行其他程序，其编译过程完全一样。

在 Libnet 开发包中有两个 ARP 程序，都在 sample 文件夹中。在 sample 文件夹中的程序，都有错误处理，而且还使用了参数化形式，这样使程序更健壮、更灵活。在这里以程序 arp.c 为例进行讲解，另外一个 arp-new.c 与此相似，读者可以自己查看。arp.c 的主要部分源代码如下，我们对其中一些重要的语句进行了详细的注释，使读者更容易理解：

```
#if (HAVE_CONFIG_H)
#if ((_WIN32) && !(__CYGWIN__))
#include "../include/win32/config.h"
#else
#include "../include/config.h"
#endif
#endif
#include "./libnet_test.h"                    /*此文件中包含一些测试数据*/
int
main(int argc, char *argv[])
{
    int c;
    u_int32_t i;
    libnet_t *l;                              /*Libnet 句柄指针*
    libnet_ptag_t t;                          /*协议标记变量*/
    char *device = NULL;                      /*网络接口变量，赋值为NULL，表示由 Libnet 自动搜索*/
    u_int8_t *packet;
```

```
    u_int32_t packet_s;
    char errbuf[LIBNET_ERRBUF_SIZE];
/*存储错误信息的字符串*/
    printf("libnet 1.1 packet shaping: ARP[link -- autobuilding ethernet]\n");
    if (argc > 1)
    {
        device = argv[1];
    }
    l = libnet_init(
/*对 Libnet 开发环境进行初始化*/
    LIBNET_LINK_ADV,                        /* Libnet 类型 */
    device,                                 /*设置网络接口*/
    errbuf                                  /*错误信息 */
    );
if (l == NULL)
{
    fprintf(stderr, "%s", errbuf);
    exit(EXIT_FAILURE);
}
    else
//i = libnet_get_ipaddr4(l);
/*获取本地 IP 地址，在这里我们把它改变成以下语句，设置任意的 IP 地址，在这里是 192.168.0.12，这里主要是用
于测试，在后面的程序中都可以进行这样的修改*/
i=libnet_name2addr4(l, "192.168.0.12", 0);
/*构造 ARP 协议块，插入到协议块链表之中*/
t = libnet_build_arp(                   /*构造 ARP 协议块，插入到协议块链表之中*/
    ARPHRD_ETHER,                       /*硬件地址类型，ARPHRD_ETHER 表示以太网*/
    ETHERTYPE_IP,                       /*协议地址类型，ETHERTYPE_IP 表示 IP 协议*/
    6,                                  /*硬件地址长度，在这里是以太网地址的长度，所以是 6 */
    4,                                  /*协议地址长度，在这里是 IP 地址的长度，所以是 4 */
    ARPOP_REPLYV,                       /*操作类型，在这里用 APPOP_REPLY 表示 APR 应答 */
    enet_src,
    /* 发送方硬件地址，在头文件 libnet_test.h 中有定义，其值为 0D:0E:0A:0D:00:00 */
    (u_int8_t *)&i,
    /* 发送方协议地址 ，在头文件 libnet_test.h 中有定义，其值为 00:10:67:00:B1:86*/
    enet_dst,                           /*目标硬件地址*/
    (u_int8_t *)&i,                     /*目标协议地址*/
    NULL,                               /*负载，此时为空*/
    0,                                  /*负载长度，为 0*/
    1,                                  /*Libnet 句柄*/
    0                       /*协议标记，此时是 0 表示构造一个新的协议块，而不是修改已经存在的。*/
    );
if (t == -1)                            /*出错处理*/
```

```
{
    fprintf(stderr, "Can't build ARP header: %s\n", libnet_geterror(l));
    goto bad;
}
t = libnet_autobuild_ethernet(          /*构造以太网协议块*/
    enet_dst,                           /*以太网目的地址*/
    ETHERTYPE_ARP,                      /*协议类型，在这里 ETHERTYPE_ARP 表示是 ARP 协议*/
    l                                   /*Libnet 句柄*/
    );
if (t == -1)                            /*出错处理*/
    {
        fprintf(stderr, "Can't build ethernet header: %s\n",
                libnet_geterror(l));
        goto bad;
    }
    if (libnet_adv_cull_packet(l, &packet, &packet_s) == -1)
    {
        fprintf(stderr, "%s", libnet_geterror(l));
    }
    else
    {
        fprintf(stderr, "packet size: %d\n", packet_s);
        libnet_adv_free_packet(l, packet);
    }
    c = libnet_write(l);                    /*发送 ARP 数据包*/
    if (c == -1)
    {
        fprintf(stderr, "Write error: %s\n", libnet_geterror(l));
        goto bad;
    }
    else
    {
        fprintf(stderr, "Wrote %d byte ARP packet from context \"%s\"; "
                "check the wire.\n", c, libnet_cq_getlabel(l));
    }
    libnet_destroy(l);                      /*销毁 Libnet*/
    return (EXIT_SUCCESS);
    bad:
    libnet_destroy(l);
    return (EXIT_FAILURE);
}
```

在 Windows 平台下的编译过程如下：

假设此文件名为 arp.c，首先用 Visual C++ 打开此文件，然后选择 build 菜单，创建一个

新的工作空间，如图 5-27 所示。

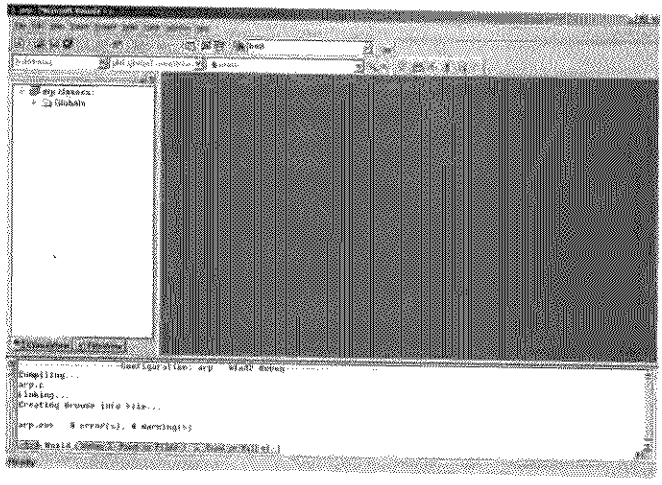

图 5-27　利用 Visual C++ 编译程序

然后进行设置，首先必须对 Visual C++ 的包含文件路径进行设置。打开菜单 tools 到 options，然后设置 directories 页面，选择 include files，然后添加一个路径，此路径应该是 Libnet 开发包安装路径中的 include，在这里是

```
F:\LIBNET-1.1.2.1\LIBNET\INCLUDE
F:\LIBNET-1.1.2.1\LIBNET\SAMPLE
```

因为上面在安装 Libnet 开发包时，把 Libnet 开发包安装在 F:\LIBNET-1.1.2.1 目录中。当然这要根据个人安装的路径而定，如图 5-28 所示。

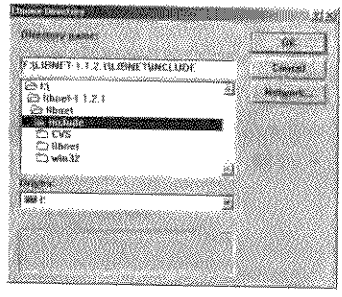

图 5-28　添加 include 路径

然后再选择 library files 选项，添加一个路径，此路径应该是 Libnet 开发包安装路径中的库文件路径，在这里是

```
F:\LIBNET-1.1.2.1\LIBNET\WIN32\DEBUG
```

因为上面在安装 Libnet 开发包的时候，把 Libnet 开发包安装在 F:\LIBNET-1.1.2.1 目录中（当然这要根据个人安装的路径而定），并且在 Windows 下安装 Libnet 时，其生成的 library 文件就在目录 F:\LIBNET-1.1.2.1\LIBNET\WIN32\DEBUG 里面。

在上面安装 Libnet 时生成了一个 libnet.lib 库文件，此时要编译源代码，就必须把此文件夹加入到 Visual C++ 工作空间之中。具体方法是：选择菜单 Project 的子菜单 setting...，打开对话框 Project Settings，然后选择 Link 页面，对 Object/library modules:选项加入一个文件，即 libnet.lib，如图 5-29 所示。

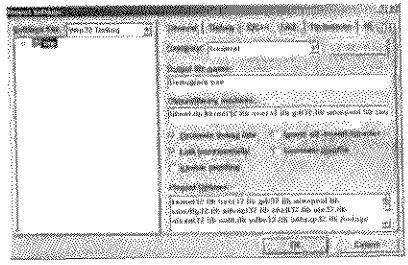

图 5-29　添加 libnet.lib 文件

在完成了上述的一些设置之后，就可以编译了，编译结果如图 5-30 所示，这表示成功了。此程序构造了一个具有 42 字节长度的 ARP 数据包。

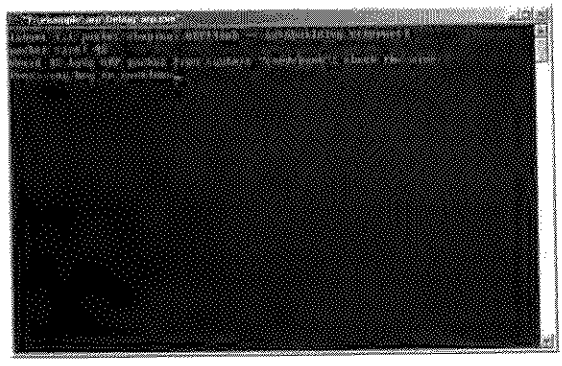

图 5-30　编译运行结果

使用抓包工具 Analyzer 可以清楚地看到所发送的 ARP 数据包如图 5-31 所示。

在软件 Analyzer 中，构造的 ARP 数据包的信息都可以清楚地看到。其源 MAC 地址为 0D:0E:0A:0D:00:00，目的 MAC 地址为 00:10:67:00:B1:86，源 IP 地址为 192.168.0.12，目的地址也为 192.168.0.12。这说明我们构造的 ARP 数据包完全正确。

如果是在 Linux 平台下使用 Libnet 开发包，其编译过程如下。

```
gcc -o arp arp.c -lnet
```

gcc 是 Linux 下的编译命令，-o 选项表示生成的目标文件名称，-lnet 表示要使用 Libnet

开发包的链接库。

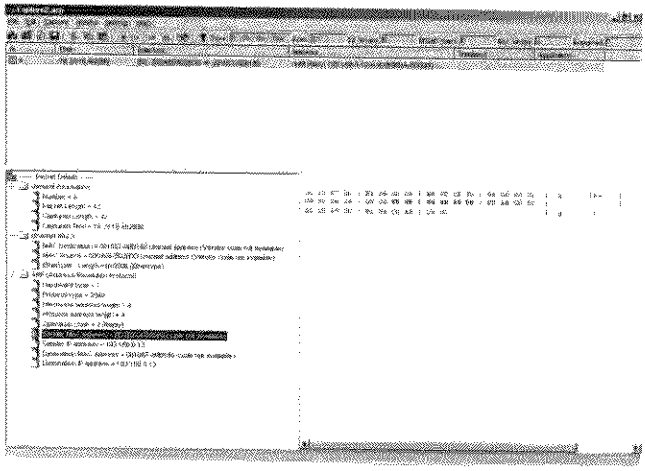

图 5-31　使用 Analyzer 的分析结果

编译之后，在命令行执行以下命令：

```
./arp
```

其运行结果如图 5-32 所示。

图 5-32　Linux 平台下的运行结果

此时可以利用 Tcpdump 软件来查看我们的运行结果是否正确，如图 5-33 所示。

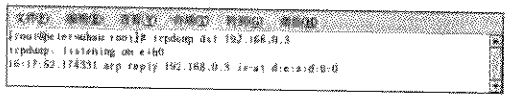

图 5-33　使用 Tcpdump 的分析结果

5.6.2　构造 IP 数据包

构造一个 IP 数据包，其包含的内容可以是任意内容，也就是说传输层和应用层的内容都可以任意设定。那么构造的协议块主要是 IP 协议块和以太网协议块。

下面写一个程序，开发环境为 Visual C++ 6.0，其中没有考虑错误处理，非常简单。其源代码如下：

```
#include <winsock/libnet.h>                    /*包含的头文件*/
void  main( )
{
```

```
int packet_size;                        /*构造的数据包大小*/
libnet_t *l;                            /*Libnet 句柄*/
char *device = NULL;                    /*设备变量*/
char *destination_ip_str = "192.168.0.2";
/*目的 IP 地址字符串变量，可以指定任意一个合法的 IP 地址*/
      char *source_ip_str = "192.168.0.3";
/*源 IP 地址字符串变量，可以指定任意一个合法的 IP 地址*/
u_char hardware_source[6] = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06};
/*源硬件地址，可以指定任意的 MAC 地址*/
u_char hardware_destination[6] = {0x06, 0x05, 0x04, 0x03, 0x02, 0x01};
/*目的硬件地址，可以指定任意的 MAC 地址*/
u_long     destination_ip;                  /*目的 IP 地址*/
     u_long  source_ip;                     /*源 IP 地址*/
char error_buffer[LIBNET_ERRBUF_SIZE];      /*存放错误信息*/
libnet_ptag_t ip_protocol_tag = 0;          /*IP 协议块的协议标记*/
libnet_ptag_t ethernet_protocol_tag = 0;    /*以太网协议块的协议标记*/
u_short protocol = IPPROTO_UDP;             /*表示 IP 上一层的协议类型，在这里是 UDP 协议*/
u_char payload[255] ="";                    /*负载的内容，初始值为空*/
      u_long payload_size = 0;              /*负载的长度，初始值为 0*/
/*对 Libnet 进行初始化操作*/
l = libnet_init(
      LIBNET_LINK,                          /* Libnet 类型 */
           device,                          /* 网络设备 */
      error_buffer);                        /* 错误信息 */
      destination_ip = libnet_name2addr4(l, destination_ip_str, LIBNET_RESOLVE);
      /*把目的 IP 地址字符串类型转换为网络顺序的数值*/
source_ip = libnet_name2addr4(l, source_ip_str, LIBNET_RESOLVE);
/*把源 IP 地址字符串类型转换为网络顺序的数值*/
strncpy(payload, "test", sizeof(payload)-1);
/*构造负载的内容，此时的内容为"test"，只是测试*/
payload_size = strlen(payload);
/*计算负载内容的长度*/
ip_protocol_tag = libnet_build_ipv4(
/*构造 IP 协议块*/
    LIBNET_IPV4_H + payload_size,
    /* IP 协议块的整个长度，在这里是 IP 协议头加上负载的长度 */
    0,                          /* 服务质量，在这里为 0 */
    242,                        /* IP 标记 */
    0,                          /* IP 紧急指针*/
    64,                         /* 生存时间 */
    protocol,                   /*上层协议类型 */
    0,                          /* 校验和，此时为 0，表示由 Libnet 自动计算*/
    source_ip,                  /*源 IP 地址*/
```

```
        destination_ip,                 /*目的 IP 地址*/
        payload,                        /*负载内容*/
        payload_size,                   /*负载内容的大小*/
        l,                              /* Libnet 句柄，由 libnet_init()函数生成*/
        ip_protocol_tag                 /* IP 协议块标记，此时为 0，表示构造新的 IP 协议块*/
        );
    ethernet_protocol_tag = libnet_build_ethernet(          /*构造一个以太网协议块*/
        hardware_destination,           /*以太网目的地址*/
        hardware_source,                /*以太网源地址 */
        ETHERTYPE_IP,                   /*上层协议类型，在这里必须是 IP 协议 */
        NULL,                           /*负载，这里为空*/
        0,                              /*负载大小*/
        l,                              /*Libnet 句柄*/
        ethernet_protocol_tag           /*以太网协议块标记，此时为 0，表示构造新的以太网协议块*/
        );
    packet_size = libnet_write(l);      /*发送数据包*/
    printf("发送一个%d 字节的 IP 数据包\n",packet_size);
    libnet_destroy(l);                  /*销毁 Libnet */
}
```

经过编译，运行后的结果如图 5-34 所示。

图 5-34　构造 IP 数据包程序的运行结果

图 5-34 表示发送了一个 38 字节长度的 IP 数据包，包括以太网 14 字节、IP 协议头 20 字节，以及负载“test”的 4 字节。

使用 Analyzer 分析的结果如图 5-35 所示。

可以在 Analyzer 中清晰地看到 IP 数据包的组成情况，其中 14 字节的以太网协议块如图 5-36 中所标记的部分所示。

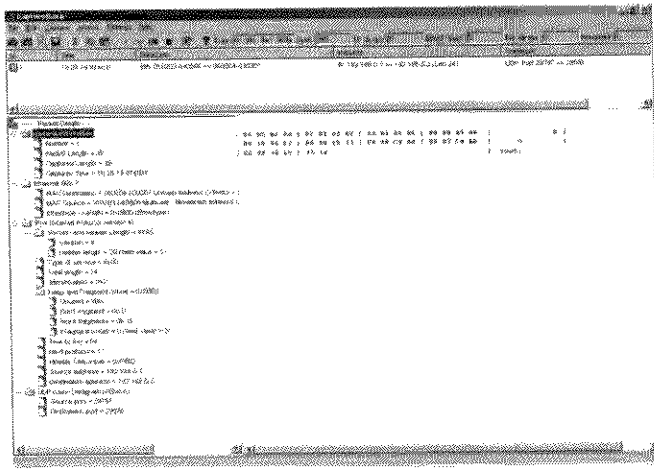

图 5-35　使用 Analyzer 工具捕获的 IP 数据包

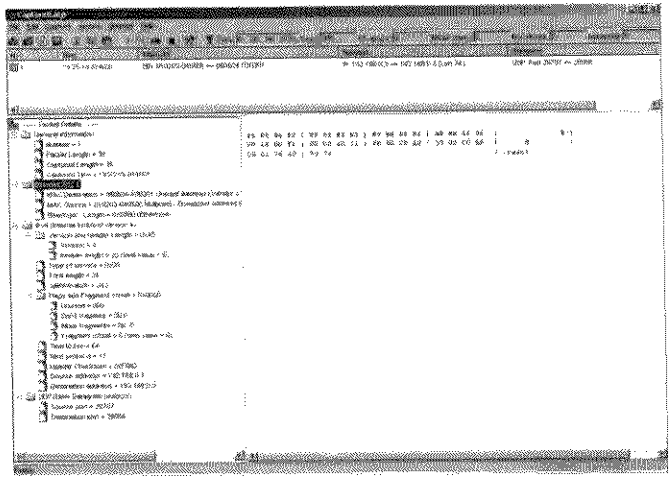

图 5-36　IP 数据包中的以太网协议块内容

IP 协议块中所有的数据元素都可以看到，如 IP 源地址、IP 目的地址、TOS、TTL 等，如图 5-37 所示。

其中 4 字节的负载内容如图 5-38 中所标记的部分和图 5-39 所示。使用 Analyzer 分析工具可以很好地分析 Libnet 开发包所构造的各种数据包，而且可以根据 Analyzer 来判断所构造的数据包正确与否。

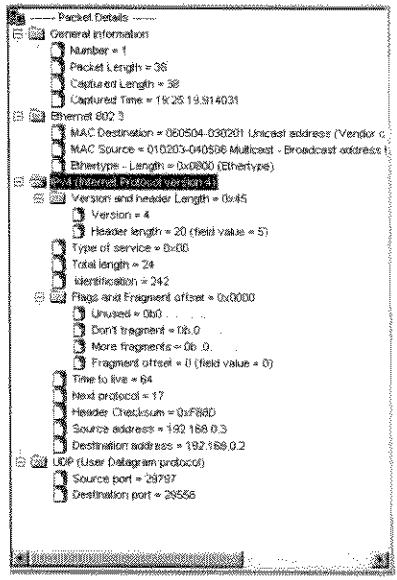

图 5-37　IP 数据包的详细内容

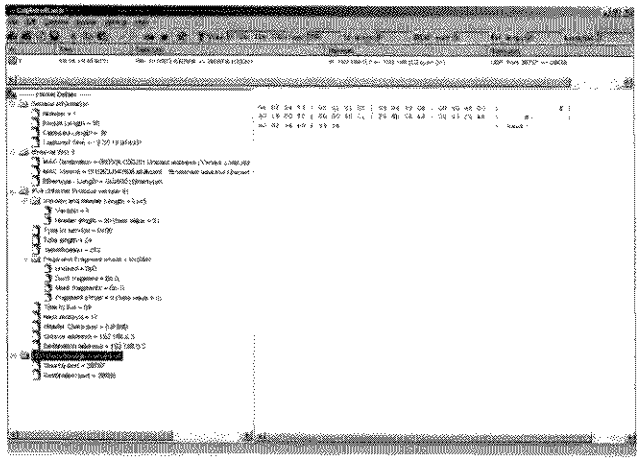

图 5-38　IP 数据包的负载内容

```
06 05 04 03 | 02 01 01 02 | 03 04 05 06 | 08 00 45 00   [..............E.]
00 18 00 F2 | 00 00 40 11 | F8 8D C0 A8 | 00 03 C0 A8   [......@.........]
00 02 74 65 | 73 74                                    [..test]
```

图 5-39　IP 数据包的十六进制显示

下面再以 Libnet 开发包中的程序 ip_raw.c 为例来进行讲解。ip_raw.c 源代码的主要部分如下：

```
#if (HAVE_CONFIG_H)
#include "../include/config.h"
#endif
#include "./libnet_test.h"
int
main(int argc, char *argv[])
{
    int c;                                          /*数据包大小*/
    libnet_t *l;                                    /*Libnet 句柄*/
    char *device = NULL;                            /* 网络设备名字*/
    char *dst = "2.2.2.2", *src = "1.1.1.1";        /*目的 IP 地址和源 IP 地址*/
    u_long src_ip, dst_ip;
    char errbuf[LIBNET_ERRBUF_SIZE];                /*错误信息*/
    libnet_ptag_t ip_ptag = 0;                      /*协议标记*/
    u_short proto = IPPROTO_UDP;                    /*IP 上一层的协议类型，默认为 UDP 协议*/
    u_char payload[255] = {0x11, 0x11, 0x22, 0x22, 0x00, 0x08, 0xc6, 0xa5};
    /*负载，这里是用十六进制写的*/
    u_long payload_s = 8;                           /*负载的长度*/
    printf("libnet 1.1 packet shaping: IP + payload[raw]\n");
    while ((c = getopt(argc, argv, "d:s:tp:i:h")) != EOF)
          /*顺序读取各种参数*/
    {
        switch (c)
        {
            case 'd':                               /*-d 表示目标 IP 地址*/
                  dst = optarg;
              break;
            case 's':                               /*-s 表示源 IP 地址*/
                  src = optarg;
              break;
            case 'i':                               /*-i 表示网络设备名字*/
              device = optarg;
              break;
            case 't':                               /*-t 表示 IP 上层协议类型*/
              proto = IPPROTO_TCP;
              break;
            case 'p':                               /*-p 表示负载的内容*/
              strncpy(payload, optarg, sizeof(payload)-1);
              payload_s = strlen(payload);
              break;
            case 'h':                               /*-h 表示用法*/
```

```
            usage(argv[0]);
            exit(EXIT_SUCCESS);
        default:                                /*退出*/
            exit(EXIT_FAILURE);
    }
}
l = libnet_init(                                /*初始化 Libnet*/
            LIBNET_RAW4,
        /*Libnet 类型，在这里是 LIBNET_RAW4 表示 IPv4 原始套接子接口，所构造数据包的底层应该是 IP 协议
        块。而前面的一个程序中 LIBNET_LINK 表示构造的数据包的底层应该是以太网协议块，即链路层协议块*/
            device,                             /*网络设备接口*/
    errbuf                                      /*错误信息*/
    );
printf("Using device %s\n", l->device);
if (l == NULL)                          /*出错处理*/
{
    fprintf(stderr, "libnet_init() failed: %s", errbuf);
    exit(EXIT_FAILURE);
}
if ((dst_ip = libnet_name2addr4(l, dst, LIBNET_RESOLVE)) == -1)
/*进行 IP 地址格式的转换，把目的 IP 地址的字符串形式转化为网络字节顺序的数据*/
{
            fprintf(stderr, "Bad destination IP address: %s\n", dst);
            exit(EXIT_FAILURE);
}
if ((src_ip = libnet_name2addr4(l, src, LIBNET_RESOLVE)) == -1)
/*把源 IP 地址字符串形式转化为网络字节顺序的数据*/
{
        fprintf(stderr, "Bad source IP address: %s\n", src);
            exit(EXIT_FAILURE);
}

ip_ptag = libnet_build_ipv4(        /*构造 IPv4 协议块*/
    LIBNET_IPV4_H + payload_s,      /*IP 协议块的长度，在这里应该为 IP 头部长度加上负载长度*/
    0,                              /*服务质量，这里赋值为 0 */
    242,                            /*IP 标识，这里赋值为 242*/
    0,                              /*IP 偏移量，这里赋值为 0*/
    64,                             /*生存时间，这里赋值为 64*/
    proto,                          /*上层协议类型*/
    0,                              /*校验和，这里为 0，表示由 Libnet 自动计算校验和*/
    src_ip,                         /*源 IP 地址*/
    dst_ip,                         /*目的 IP 地址*/
    payload,                        /*负载*/
```

```
        payload_s,                      /*负载大小*/
        l,                              /*Libnet 句柄*/
        ip_ptag                         /*IP 协议块标记*/
        );
    if (ip_ptag == -1)                  /*出错处理*/
    {
        fprintf(stderr, "Can't build IP header: %s\n", libnet_geterror(l));
        goto bad;
    }
    c = libnet_write(l);                /*发送数据包*/
    if (c == -1)                        /*出错处理*/
    {
        fprintf(stderr, "Write error: %s\n", libnet_geterror(l));
        goto bad;
    }
    else
    {
        fprintf(stderr, "Wrote %d byte IP packet; check the wire.\n", c);
    }
    libnet_destroy(l);                  /*销毁 Libnet*/
    return (EXIT_SUCCESS);
    bad:                                /*出错处理*/
    libnet_destroy(l);                  /*销毁 Libnet*/
    return (EXIT_FAILURE);
}
void
usage(char *name)                       /*使用方法*/
{
    fprintf(stderr,
        "usage: %s [-s source_ip] [-d destination_ip]"
         " [-i iface] [-p payload] [-t]\n",
         name);
}
```

注意：在编译此源代码时，需要把文件 getopt.c 加入到 Visual C++ 6.0 项目中，getopt.c 在 Libnet 开发包的文件夹/sample/win32 中。

最后编译通过，然后用参数来进行测试，看其运行的结果如何。如图 5-40 所示，打开 Visual C++ 菜单 Project 的子菜单 Settings 中的对话框 Project Settings，然后在页面 Debug 的选项 Program arguments 中填入：

```
-s 192.168.0.2 -d 192.168.0.3
-i \Device\NPF_{5973DFB0-AD5B-4065-A22B-C61FFB64624F}
-p 0123456789
```

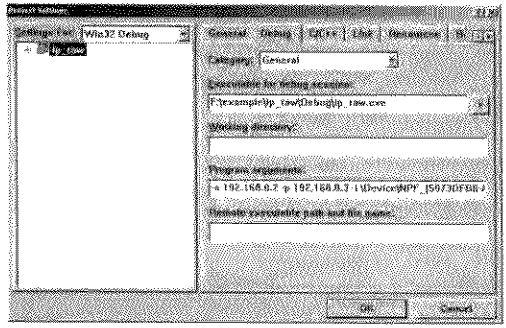

图 5-40　设置程序运行参数

其运行的结果如图 5-41 所示。

图 5-41　程序运行结果

利用 Analyzer 工具捕获所运行的结果，如图 5-42 所示。

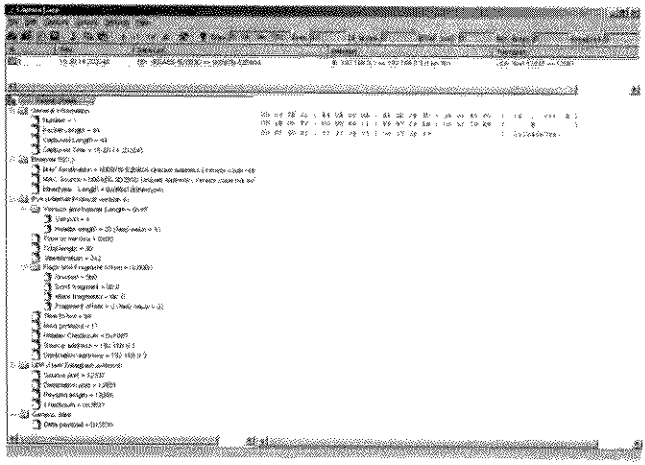

图 5-42　Analyzer 捕获的结果

总体信息是捕获一个 IP 数据包。此 IP 数据包的详细信息如图 5-43 所示。

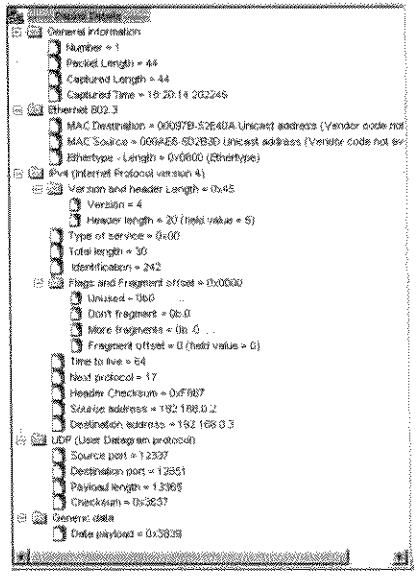

图 5-43　IP 数据包的详细信息

可以看出，所构造的 IP 数据包的源地址为 192.168.0.2，目的地址为 192.168.0.3，其传输层协议为 UDP 协议。但 UDP 协议的数据是我们写的任意数据“0123456789”，如图 5-44 所示。

```
00 09 7B 52 | E4 0A 00 0A | E6 5D 2B 3D | 08 00 45 00   [. {R.....]+=..E.]
00 1E 00 F2 | 00 00 40 11 | F8 87 C0 A8 | 00 02 C0 A8   [......@.........]
00 03 30 31 | 32 33 34 35 | 36 37 38 39                 [..0123456789]
```

图 5-44　数据包的十六进制显示

5.6.3　构造 TCP 数据包

TCP 协议格式如图 5-45 所示。

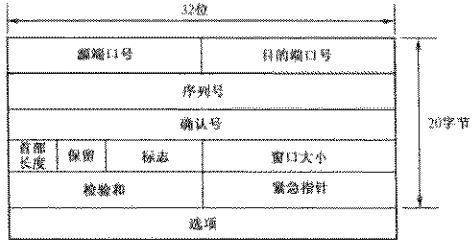

图 5-45　TCP 协议格式

在构造 TCP 数据包时，必须按照 Libnet 构造数据包的顺序来进行构造。首先构造 TCP 协议块，再构造 IP 协议块，最后构造 Ethernet 协议块。如果 TCP 有选项部分，则必须最先构造 TCP 选项数据。在 Libnet 中用函数 libnet_build_tcp_options()构造 TCP 选项数据。

以 Libnet 开发包中的程序 tcp1.c 进行讲解，其部分核心代码如下：

```
#if (HAVE_CONFIG_H)
#include "../include/config.h"
#endif
#include "libnet_test.h"
#ifdef __WIN32__
#include "../include/win32/getopt.h"
#endif
int
main(int argc, char *argv[])
{
    int c;
    u_char *cp;
    libnet_t *l;
    libnet_ptag_t t;
    char *payload;
    u_short payload_s;
    u_long src_ip, dst_ip;                          /*源 IP 地址和目的 IP 地址*/
    u_short src_prt, dst_prt;                       /*源端口号和目的端口号*/
    char errbuf[LIBNET_ERRBUF_SIZE];

    printf("libnet 1.1 packet shaping: TCP + options[link]\n");
    l = libnet_init(
            LIBNET_LINK,                            /*Libnet 类型*/
            NULL,                                   /*网络设备接口*/
            errbuf);                                /*错误信息*/
    if (l == NULL)
    {
        fprintf(stderr, "libnet_init() failed: %s", errbuf);
        exit(EXIT_FAILURE);
    }
         src_ip  = 0;
    dst_ip  = 0;
    src_prt = 0;
    dst_prt = 0;
    payload = NULL;                                 /*负载*/
    payload_s = 0;                                  /*负载大小*/
    while ((c = getopt(argc, argv, "d:s:p:")) != EOF)
    {
        switch (c)
```

```
    {
        case 'd':                                   /*目的IP地址*/
            if (!(cp = strrchr(optarg, '.')))
            {
                usage(argv[0]);
            }
            *cp++ = 0;
            dst_prt = (u_short)atoi(cp);
            if ((dst_ip = libnet_name2addr4(l, optarg, LIBNET_RESOLVE)) == -1)
            {
                fprintf(stderr, "Bad destination IP address: %s\n", optarg);
                exit(EXIT_FAILURE);
            }
            break;
        case 's':                                   /*源IP地址*/
            if (!(cp = strrchr(optarg, '.')))
            {
                usage(argv[0]);
            }
            *cp++ = 0;
            src_prt = (u_short)atoi(cp);
            if ((src_ip = libnet_name2addr4(l, optarg, LIBNET_RESOLVE)) == -1)
            {
                fprintf(stderr, "Bad source IP address: %s\n", optarg);
                exit(EXIT_FAILURE);
            }
            break;
        case 'p':                                   /*负载*/
            payload = optarg;
            payload_s = strlen(payload);
            break;
        default:
            exit(EXIT_FAILURE);
    }
}
if (!src_ip || !src_prt || !dst_ip || !dst_prt)
{
    usage(argv[0]);
    exit(EXIT_FAILURE);
}
t = libnet_build_tcp_options(                       /*构造TCP选项*/
    "\003\003\012\001\002\004\001\011\010\012\077\077\077\077\000\000\000\000\000\000",
    20,
```

```
    1,
    0);
if (t == -1)
{
    fprintf(stderr, "Can't build TCP options: %s\n", libnet_geterror(l));
    goto bad;
}
t = libnet_build_tcp(                    /*构造 TCP 协议*/
    src_prt,                             /* 源端口号 */
    dst_prt,                             /*目的端口号*/
    0x01010101,                          /*序列号 */
    0x02020202,                          /*确认序列号*/
    TH_SYN,                              /*控制标记*/
    32767,                               /*窗口大小*/
    0,                                   /*校验和*/
    10,                                  /*紧急指针*/
    LIBNET_TCP_H + 20 + payload_s,       /*数据包大小*/
          payload,                       /* 负载*/
    payload_s,                           /*负载大小*/
    l,                                   /*Libnet 句柄*/
    0);                                  /*协议标记*/
if (t == -1)
{
    fprintf(stderr, "Can't build TCP header: %s\n", libnet_geterror(l));
    goto bad;
}
    t = libnet_build_ipv4(               /*构造 IPv4 协议块*/
    LIBNET_IPV4_H + LIBNET_TCP_H + 20 + payload_s,        /* 长度 */
    0,                                   /*服务质量 */
    242,                                 /* 标识*/
    0,                                   /* 偏移 */
    64,                                  /* 生存时间*/
    IPPROTO_TCP,                         /*上层协议类型 */
    0,                                   /*校验和*/
    src_ip,                              /* 源 IP 地址 */
    dst_ip,                              /* 目的 IP 地址 */
    NULL,                                /*负载*/
    0,                                   /*负载大小*/
    l,                                   /* Libuet 句柄*/
    0);                                  /* 协议块标记*/
if (t == -1)
{
    fprintf(stderr, "Can't build IP header: %s\n", libnet_geterror(l));
```

```
        goto bad;
    }
     t = libnet_build_ethernet(            /*构造以太网协议块*/
        enet_dst,                          /*以太网目的地址*/
        enet_src,                          /*以太网源地址*/
        ETHERTYPE_IP,                      /*上层协议类型 */
        NULL,                              /*负载*/
        0,                                 /*负载大小*/
        1,                                 /*Libnet 句柄 */
        0);                                /*协议块标记*/
    if (t == -1)
    {
        fprintf(stderr, "Can't build ethernet header: %s\n", libnet_geterror(l));
        goto bad;
    }
    c = libnet_write(l);                   /*发送数据包*/
    if (c == -1)
    {
        fprintf(stderr, "Write error: %s\n", libnet_geterror(l));
        goto bad;
    }
    else
    {
        fprintf(stderr, "Wrote %d byte TCP packet; check the wire.\n", c);
    }
    libnet_destroy(l);                     /*销毁Libnet*/
    return (EXIT_SUCCESS);
    bad:
    libnet_destroy(l);
    return (EXIT_FAILURE);
}
void
usage(char *name)
{
    fprintf(stderr,
        "usage: %s -s source_ip.source_port -d destination_ip.destination_port"
        " [-p payload]\n",
        name);
}
#if defined(__WIN32__)
#include <../include/win32/getopt.h>
#include <winsock2.h>
#include <ws2tcpip.h>
```

```
#endif
```

运行此程序，设置运行参数如下：

```
-s 192.168.0.2
-d 192.168.0.3
-p 0123456789012345678 9
```

运行结果如图 5-46 所示。

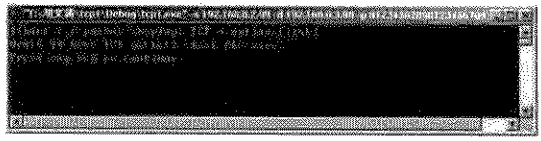

图 5-46　构造 TCP 程序的运行结果

利用 Analyzer 工具分析，其捕获的数据包如图 5-47 所示。

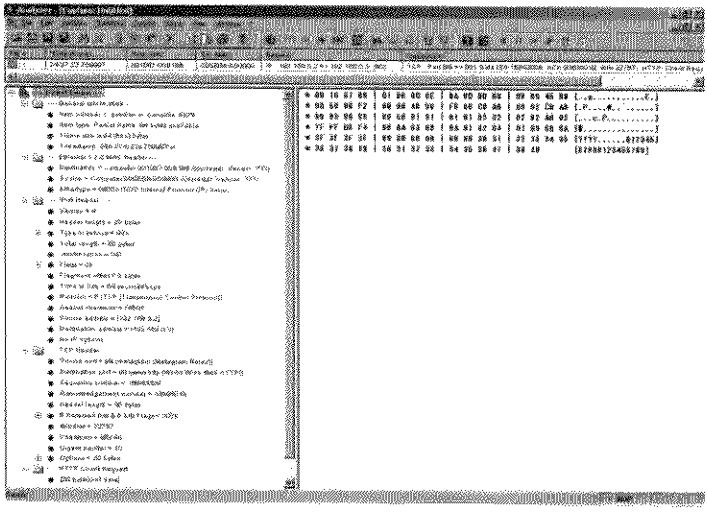

图 5-47　捕获 TCP 数据包的结果

所捕获数据包的详细信息如下：

```
----- Packet Details -----
----- General Information -----
Item number 1, position in dumpfile 1009
Item type: Partial frame, 94 bytes available
Frame size is 94 (5Ex) bytes
Timestamp: 24h:37m:23s:759997us
----- Ethernet v.2.0 MAC Header ----- [0-13]
Destination = Computer 001067-008186 (Universal; Vendor: 777) - [0-13]
```

```
Source = Computer 0D0E0A-0D0000 (Universal; Vendor: ???) - [6-11]
Ethertype = 0800h (DOD Internet Protocol (IP) Xerox) - [12-13]
----- IPv4 Header ----- [14-33]
Version = 4 - (14-14)
Header length = 20 bytes - (14-14)
Type of service = 00h - [15-15]
 000. .... = priority 0: routine - (15-15)
 ...0 .... = normal delay - (15-15)
 .... 0... = normal throughput - (15-15)
 .... .0.. = normal reliability - (15-15)
 .... ..0. = normal monetary cost - (15-15)
 .... ...0 = Reserved bit, for future usage, which has to be 0 - (15-15)
Total length = 80 bytes - [16-17]
Identification = 242 - [18-19]
Flags = 0h - (20-20)
 0... .... = must be 0 - (20-20)
 .0.. .... = may fragment - (20-20)
 ..0. .... = last fragment - (20-20)
Fragment offset = 0 bytes - (20-21)
Time to live = 64 seconds/hops - [22-22]
Protocol = 6 (TCP [Transmission Control Protocol]) - [23-23]
Header checksum = F860h - [24-25]
Source address = [192.168.0.2] - [26-29]
Destination address = [192.168.0.3] - [30-33]
No IP options - [14-14]
----- TCP Header ----- - [34-73]
Source port = 99 (metagram (Metagram Relay)) - [34-35]
Destination port = 80 (www-http (World Wide Web HTTP)) - [36-37]
Sequence number = 16843009 - [38-41]
Acknowledgement number = 33686018 - [42-45]
Header length = 40 bytes - (46-46)
6 Reserved bits & 6 bits Flags = 002h - (46-47)
 0000 00.. .... = 6 Reserved bits, for future usage, which have to be 0 - (46-47)
 0000 000. .... = No urgent pointer - [46-47]
 0000 00.0 .... = No acknowledgement - (46-47)
 0000 00.. 0... = No push - (46-47)
 0000 00.. .0.. = No reset - (46-47)
 0000 00.. ..1. = SYN - (46-47)
 0000 00.. ...0 = No FIN - (46-47)
Window = 32767 - [48-49]
Checksum = B6F4h - [50-51]
Urgent pointer = 10 - [52-53]
Options = 20 bytes - [54-73]
```

```
Other Options
----- HTTP Client Request ----- - [74-93]
[20 byte(s) of data]
20
01234567890123456789
================================================================================
* 00 10 67 00 | B1 86 0D 0E | 0A 0D 00 00 | 08 00 45 00 [..g...........E.]
* 00 50 00 F2 | 00 00 40 06 | F8 60 C0 A8 | 00 02 C0 A8 [.P....@..`......]
* 00 03 00 63 | 00 50 01 01 | 01 01 02 02 | 02 02 A0 02 [...c.P..........]
* 7F FF B6 F4 | 00 0A 03 03 | 0A 01 02 04 | 01 09 08 0A [■...............]
* 3F 3F 3F 3F | 00 00 00 00 | 00 00 30 31 | 32 33 34 35 [????......012345]
* 36 37 38 39 | 30 31 32 33 | 34 35 36 37 | 38 39       [67890123456789]
```

5.6.4 构造 UDP 数据包

UDP 协议格式如图 5-48 所示。

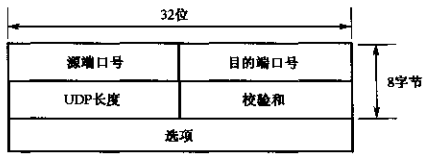

图 5-48 UDP 协议格式

其中在 UDP 协议中重要的信息就是端口号，不同的端口号代表不同的应用。所以，端口号的控制是构造 UDP 数据包的主要内容。在 Libnet 中使用了端口列表的数据结构，用来管理和控制端口号。

以 Libnet 开发包中的程序 udp2.c 进行讲解，其部分核心代码如下：

```
    #if (HAVE_CONFIG_H)
#include "../include/config.h"
#endif
#include "./libnet_test.h"
int
main(int argc, char **argv)
{
   int c, build_ip;
   struct timeval r;
   struct timeval s;
   struct timeval e;
   libnet_t *l;                          /*Libnet 句柄*/
   libnet_ptag_t udp;                    /*协议标记*/
   char *payload;
   libnet_ptag_t t;                      /*协议标记*/
   struct libnet_stats ls;               /*Libnet 状态*/
```

```
u_int16_t payload_s;                    /*负载*/
u_int32_t src_ip, dst_ip;               /*源 IP 地址，目的 IP 地址*/
u_int16_t bport, eport, cport;          /*端口*/
libnet_plist_t plist, *plist_p;         /*端口列表*/
char errbuf[LIBNET_ERRBUF_SIZE];        /*错误信息*/
printf("libnet 1.1.2 packet shaping: UDP2[link]\n");
l = libnet_init(                        /*初始化*/
        LIBNET_LINK,                    /*Libnet 类型*/
        NULL,                           /*网络设备接口*/
        errbuf                          /*错误信息*/
        );
if (l == NULL)
{
    fprintf(stderr, "libnet_init() failed: %s", errbuf);
    exit(EXIT_FAILURE);
}
src_ip = 0;
dst_ip = 0;
payload = NULL;
payload_s = 0;
plist_p = NULL;
while ((c = getopt(argc, argv, "d:s:p:P:")) != EOF)
{
    switch (c)
    {
        case 'd':                       /*-d 表示目的 IP 地址*/
            if ((dst_ip = libnet_name2addr4(l, optarg,
                    LIBNET_RESOLVE)) == -1)
            {
                fprintf(stderr, "Bad destination IP address: %s\n", optarg);
                exit(1);
            }
            break;
        case 's':                       /*-s 表示源 IP 地址*/
            if ((src_ip = libnet_name2addr4(l, optarg,
                    LIBNET_RESOLVE)) == -1)
            {
                fprintf(stderr, "Bad source IP address: %s\n", optarg);
                exit(1);
            }
            break;
        case 'P':                       /*-P 表示端口列表*/
            plist_p = &plist;
```

```
            if (libnet_plist_chain_new(1, &plist_p, optarg) == -1)
            {
                fprintf(stderr, "Bad token in port list: %s\n",
                        libnet_geterror(1));
                exit(1);
            }
            break;
        case 'p':                       /*-p 表示负载*/
            payload = optarg;
            payload_s = strlen(payload);
            break;
        default:
            usage(argv[0]);
            exit(EXIT_FAILURE);
        }
    }
    if (!src_ip || !dst_ip || !plist_p)
    {
        usage(argv[0]);
        exit(EXIT_FAILURE);
    }
    udp = 0;
#if !(__WIN32__)
    gettimeofday(&s, NULL);
#else
    /* This obviously is not as good - but it compiles now. */
    s.tv_sec = time(NULL);
    s.tv_usec = 0;
#endif
    build_ip = 1;
    while (libnet_plist_chain_next_pair(plist_p, &bport, &eport))
            /*循环构造不同端口的数据包*/
    {
        while (!(bport > eport) && bport != 0)
        {
            cport = bport++;                        /*端口自动增加*/
            udp = libnet_build_udp(                 /*构造 UDP 协议块*/
                1025,                               /*源端口号*/
                cport,                              /*目的端口号*/
                LIBNET_UDP_H + payload_s,           /*协议长度*/
                0,                                  /*校验和*/
                payload,                            /*负载*/
                payload_s,                          /*负载大小*/
```

```
            l,                                  /*Libnet 句柄*/
            udp                                 /*协议标记*/
            );
        if (udp == -1)
        {
            fprintf(stderr, "Can't build UDP header (at port %d): %s\n",
                    cport, libnet_geterror(l));
            goto bad;
        }
if (build_ip)
{
    build_ip = 0;
    t = libnet_build_ipv4(                      /*构造 IP 协议块*/
        LIBNET_IPV4_H + LIBNET_UDP_H + payload_s,   /*协议长度 */
        0,                                      /*服务质量*/
        242,                                    /*IP 标识*/
        0,                                      /*IP 偏移*/
        64,                                     /*生存时间*/
        IPPROTO_UDP,                            /*上层协议类型*/
        0,                                      /* 校验和*/
        src_ip,                                 /*源 IP 地址*/
        dst_ip,                                 /*目的 IP 地址*/
        NULL,                                   /*负载内容*/
        0,                                      /*负载大小*/
        l,                                      /*Libnet 句柄*/
        0                                       /*协议标记*/
        );
    if (t == -1)
    {
        fprintf(stderr, "Can't build IP header: %s\n", libnet_geterror(l));
        goto bad;
    }
    t = libnet_build_ethernet(                  /*构造 Ethernet 协议块*/
        enet_dst,                               /*MAC 目的地址*/
        enet_src,                               /*MAC 源地址*/
        ETHERTYPE_IP,                           /*协议类型*/
        NULL,                                   /*负载*/
        0,                                      /*负载大小 */
        l,                                      /*Libnet 句柄*/
        0                                       /*协议标记*/
        );
    if (t == -1)
    {
```

```
            fprintf(stderr, "Can't build ethernet header: %s\n",
                    libnet_geterror(l));
            goto bad;
        }
    }
            c = libnet_write(l);                        /*发送数据包*/
            if (c == -1)
            {
                fprintf(stderr, "write error: %s\n", libnet_geterror(l));
            }
            else
            {
                fprintf(stderr, "wrote %d byte UDP packet to port %d\r", c,
                        cport);
            }
        }
        fprintf(stderr, "\n");
    }
#if !(__WIN32__)
    gettimeofday(&e, NULL);
#else
    s.tv_sec = time(NULL);
    s.tv_usec = 0;
#endif
    libnet_timersub(&e, &s, &r);
    fprintf(stderr, "Total time spent in loop: %d.%d\n", r.tv_sec, r.tv_usec);
    libnet_stats(l, &ls);                       /*获取 Libent 的状态*/
    fprintf(stderr, "Packets sent:  %lld\n"
                    "Packet errors: %lld\n"
                    "Bytes written: %lld\n",
                    ls.packets_sent, ls.packet_errors, ls.bytes_written);
    libnet_destroy(l);                          /*销毁 Libnet*/
    return (EXIT_SUCCESS);
bad:
    libnet_destroy(l);
    return (EXIT_FAILURE);
}
void
usage(char *name)
{
    fprintf(stderr, "usage: %s -s s_ip -d d_ip -P port list [-p payload]\n", name);
}
```

此程序是发送一系列的 UDP 数据包，只是端口号不同而已。在 Libnet 中为了管理端口

号，使用了端口列表这个数据结构：

```
typedef struct libnet_port_list_chain libnet_plist_t;
struct libnet_port_list_chain
{
    u_int16_t node;                    /*结点个数*/
    u_int16_t bport;                   /*开始端口号*/
    u_int16_t eport;                   /*最后端口号*/
    u_int8_t  id;                      /*偏移*/
    libnet_plist_t *next;              /* 指向下一个结点*/
};
```

然后利用函数 libnet_plist_new() 新建一个端口列表。使用函数 libnet_plist_chain_next_pair() 来获取端口列表中的每个端口号。

程序的运行结果如图 5-49 所示。

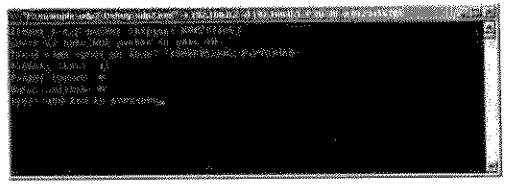

图 5-49　程序的运行结果

利用 Analyzer 工具捕获此数据包，其结果如图 5-50 所示。

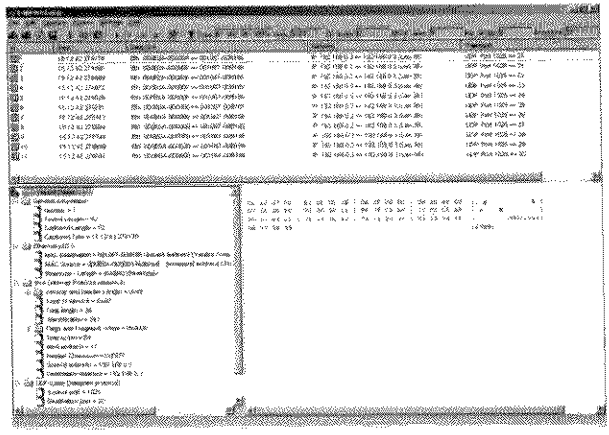

图 5-50　Analyzer 捕获 UDP 数据包的结果

在 Analyzer 中捕获的 UDP 数据包，是来自端口号 20～30 的 UDP 数据包，共构造和发送了 11 个数据包，每个数据包的端口号不一样。在这里是利用端口号链表来构造不同的 UDP 数据包的。

5.6.5 构造 ICMP 数据包

Internet 控制报文协议（ICMP）使用范围非常广泛，ICMP 报文的类型也非常丰富。Libnet 开发包可以构造和发送各种各样的 ICMP 数据包。

1. 构造 ICMP 回显数据包

ICMP 回显数据包也包括回显请求和回显应答两种类型。它们是根据 ICMP 类型字段进行区别的，如果 ICMP 类型字段为 0，就表示回显应答；如果 ICMP 类型字段为 8，就表示回显请求。在 Libnet 开发包中，使用数据常量 ICMP_ECHO 代表回显请求，使用数据常量 ICMP_ECHOREPLY 代表回显应答，它们都在 Libnet 开发包中的文件 libnet-headers.h 中有定义，读者可以详细查看。

ICMP 协议是在 IP 协议之上的，所以构造一个完整的 ICMP 数据包就必须构造 ICMP 协议块和 IP 协议块两部分，并且根据 Libnet 的开发规律，先构造高一级的协议，即首先构造 IP 协议块，然后构造 ICMP 协议块。

构造一个 ICMP 回显数据包并发送出去的核心源代码如下（其中同样没有考虑错误处理和参数化）：

```
    #include <win32/libnet.h>
void main()
{
    libnet_t *l = NULL;                          /*Libnet 句柄*/
   libnet_ptag_t protocol_tag;                   /*协议标记*/
   char *payload_liu_wen_tao = NULL;             /*负载*/
   u_short payload_length = 0;                   /*负载长度*/
   char *device ="\\Device\\NPF_{5973DFB0-AD5B-4065-A22B-C61FFB64624F}";
   /*网络设备接口，在这里用的是 Windows 下的一个接口，根据计算机的类型而定，可以通过很多方法获得，例如
   使用 windump 或 ngrep 程序来获得即可*/
   char *destination_ip_str = "192.168.0.3";     /*目的 IP 地址字符串*/
         char *source_ip_str ="192.168.0.2";     /*源 IP 地址字符串*/
         u_long source_ip = 0;                   /*源 IP 地址*/
         u_long destination_ip = 0;              /*目的 IP 地址*/
   char errbuf[LIBNET_ERRBUF_SIZE];              /*错误信息*/
         int packet_length;                      /*发送的数据包的长度*/
   l = libnet_init(                              /*初始化 Libnet*/
      LIBNET_RAW4,                               /*Libnet 类型，为原始套接字 IPv4 类型*/
      device,                                    /*网络设备接口*/
      errbuf                                     /* 错误信息*/
         );
    source_ip=libnet_name2addr4(l,source_ip_str,LIBNET_RESOLVE);
    /*把源 IP 地址字符串形式转化为网络字节顺序的数据*/
    destination_ip=libnet_name2addr4(l,destination_ip_str,LIBNET_RESOLVE);
    /*把目的 IP 地址字符串形式转化为网络字节顺序的数据*/
   protocol_tag = libnet_build_icmpv4_echo(      /*构造 ICMP 回显数据包*/
```

```
            ICMP_ECHO,                          /*类型，此时为回显请求*/
                0,                              /*代码，应该为 0*/
            0,                                  /*校验和，为 0，表示由 Libnet 句柄自动计算*/
            123,                                /*标识符，赋值为 123，自己任意填写数值*/
            456,                                /*序列号，赋值为 456，自己任意填写数值*/
            NULL,                               /*负载，赋值为空*/
            0,                                  /*负载的长度，赋值为 0*/
            l,                                  /*Libnet 句柄，应该是由 libnet_init()函数得到的*/
            0                                   /*协议块标记，赋值为 0，表示构造一个新的协议块*/
                );
    protocol_tag = libnet_build_ipv4(           /*构造 IP 协议块*/
        LIBNET_IPV4_H + LIBNET_ICMPV4_ECHO_H + payload_length,
                /*IP 协议块的长度*/
            0,                                  /*服务质量，这里赋值为 0*/
            10,                                 /*标识符，这里赋值为 10*/
            0,                                  /*偏移，这里赋值为 0*/
            20,                                 /*生存时间，这里赋值为 20*/
            IPPROTO_ICMP,                       /*上层协议类型，这里是 ICMP 协议*/
            0,                                  /*校验和，这里为 0 表示由 Libnet 计算校验和*/
            source_ip,                          /*源 IP 地址*/
            destination_ip,                     /*目的 IP 地址*/
            payload_ptr_wen_tao,                /*负载*/
                payload_length,                 /*负载的长度*/
                l,                              /*Libnet 句柄*/
                0                               /*协议块标记，为 0 表示构造一个新的 IP 协议块*/
                );
        packet_length = libnet_write(l);        /*发送由 Libnet 句柄 l 表示的数据包*/
            printf("发送一个%d 字节长度的 ICMP 回显请求数据包\n",packet_length);
            /*输出发送的数据包信息*/
        libnet_destroy(l);                      /*销毁 Libnet*/
}
```

编译运行此程序，其运行结果如图 5-51 所示。所构造的是一个 ICMP 回显请求数据包，其源 IP 地址为 192.168.0.2，其目的 IP 地址为 192.168.0.3，数据包总长度为 42 字节。

图 5-51　构造 ICMP 回显程序的运行结果

利用 Analyzer 工具进行查看，如图 5-52 所示。其显示的结果与构造和反送的数据包是一样的，证明程序运行成功。

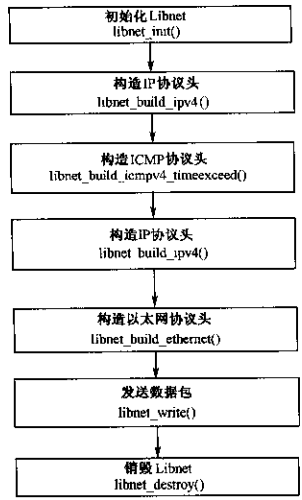

图 5-54　构造 ICMP 超时数据包的流程

在 Libnet 开发包中，程序 icmp_timeexceed.c 完成了构造 ICMP 超时数据包的功能。下面对其部分核心代码进行讲解，并进行详细分析。其部分核心代码如下：

```
#if (HAVE_CONFIG_H)
#include "../include/config.h"
#endif
#include "./libnet_test.h"
int
main(int argc, char **argv)
{
    int c;                                  /*表示数据包长度*/
    libnet_t *l;                            /*表示 Libnet 句柄*/
    libnet_ptag_t t;                        /*表示协议块标记*/
    u_long src_ip, dst_ip;                  /*源 IP 地址和目的 IP 地址*/
    u_char payload[8] = {0x11, 0x11, 0x22, 0x22, 0x00, 0x08, 0xc6, 0xa5};
    /*负载内容*/
    u_long payload_s = 8;                   /*负载长度*/
    char errbuf[LIBNET_ERRBUF_SIZE];        /*错误信息*/
    printf("libnet 1.1 packet shaping: ICMP timeexceed[link]\n");
    l = libnet_init(                        /*初始化 Libnet*/
        LIBNET_LINK,                        /*Libnet 类型，LIBNET_LIKN 表示从链路层进行构造*/
        NULL,                               /*网络设备接口*/
        errbuf                              /*错误信息*/
          );
    if (l == NULL)                          /*错误处理*/
```

```
{
    fprintf(stderr, "libnet_init() failed: %s", errbuf);
    exit(EXIT_FAILURE);
}
src_ip = 0;
dst_ip = 0;
while((c = getopt(argc, argv, "d:s:")) != EOF)        /*循环获取参数*/
    {
    switch (c)
    {
        case 'd':                                     /* -d表示目的IP地址*/
        if ((dst_ip = libnet_name2addr4(l, optarg, LIBNET_RESOLVE)) == -1)
        {
            fprintf(stderr, "Bad destination IP address: %s\n", optarg);
            exit(1);
        }
        break;
        case 's':                                     /*表示源IP地址*/
        if ((src_ip = libnet_name2addr4(l, optarg, LIBNET_RESOLVE)) == -1)
        {
            fprintf(stderr, "Bad source IP address: %s\n", optarg);
            exit(1);
        }
        break;
    }
}
if (!src_ip || !dst_ip)
{
    usage(argv[0]);
    exit(EXIT_FAILURE);
}
t = libnet_build_ipv4(                        /*构造IPv4协议块*/
      LIBNET_IPV4_H + payload_s,              /*IP协议块长度，应该为IP头长度加上负载长度*/
      IPTOS_LOWDELAY | IPTOS_THROUGHPUT,      /*服务质量*/
      0xee,                                   /*IP标识*/
      0,                                      /*IP偏移*/
      64,                                     /*生存时间*/
      IPPROTO_UDP,                            /*上层类型为UDP协议*/
      0,                                      /*校验和，这里为0表示由Libnet自动计算*/
      dst_ip,                                 /*源IP地址*/
      src_ip,                                 /*目的IP地址*/
      payload,                                /*负载*/
      payload_s,                              /*负载大小*/
```

```
        l,                                      /*Libnet 句柄*/
        0                                       /*协议块标记，这里为 0 表示构造新的标记*/
            );
if (t == -1)                                    /*错误处理*/
    {
    fprintf(stderr, "Can't build error IP header: %s\n",
            libnet_geterror(l));
    goto bad;
}
t = libnet_build_icmpv4_timeexceed(             /*构造 ICMPv4 超时数据包*/
    ICMP_TIMXCEED,                              /*ICMP 类型，这里 ICMP_TIMECEED 表示超时*/
    ICMP_TIMXCEED_INTRANS,                      /*ICMP 代码，这里表示传输期间生存时间为 0*/
    0,                                          /*校验和，这里为 0 表示由 Libnet 计算*/
    NULL,                                       /*负载*/
    0,                                          /*负载长度为 0*/
    l,                                          /*Libnet 句柄*/
            0                                   /*协议块句柄*/
            );
if (t == -1)                                    /*错误处理*/
{
    fprintf(stderr, "Can't build ICMP header: %s\n", libnet_geterror(l));
    goto bad;
}
t = libnet_build_ipv4(                          /*构造 IPv4 协议块*/
        LIBNET_IPV4_H + LIBNET_ICMPV4_TIMXCEED_H +
        LIBNET_IPV4_H + payload_s,
        /*IPv4 协议的长度，应该是本 IP 头的长度加 ICMPv4 的长度再加 ICMPv4 数据包中的 IPv4 的长度和负载
        的长度*/
        IPTOS_LOWDELAY | IPTOS_THROUGHPUT,          /*服务质量*/
        0xee,                                       /*IP 标识*/
        0,                                          /*IP 偏移*/
        64,                                         /*生存时间*/
        IPPROTO_ICMP,                               /*上层协议类型为 ICMP 协议*/
        0,                                          /*校验和*/
        src_ip,                                     /*源 IP 地址*/
        dst_ip,                                     /*目的 IP 地址*/
        NULL,                                       /*负载*/
        0,                                          /*负载大小*/
        l,                                          /*Libnet 句柄*/
        0                                           /*协议块标记*/
            );
if (t == -1)                                        /*错误处理*/
{
```

```
        fprintf(stderr, "Can't build IP header: %s\n", libnet_geterror(l));
        goto bad;
    }
    t = libnet_build_ethernet(                          /*构造以太网协议块*/
          enet_dst,                                     /*MAC 目的地址*/
          enet_src,                                     /*MAC 源地址*/
          ETHERTYPE_IP,                                 /*上层协议类型为 IP 协议*/
          NULL,                                         /*负载*/
          0,                                            /*负载大小*/
          1,                                            /*Libnet 句柄*/
          0                                             /*协议块标记*/
              );
    if (t == -1)                                        /*出错处理*/
    {
        fprintf(stderr, "Can't build ethernet header: %s\n", libnet_geterror(l));
        goto bad;
    }
    c = libnet_write(l);                                /*发送数据包*/
          if (c == -1)                                  /*出错处理*/
    {
        fprintf(stderr, "Write error: %s\n", libnet_geterror(l));
        goto bad;
    }
    else
    {
        fprintf(stderr, "Wrote %d byte ICMP packet; check the wire.\n", c);
    }
    libnet_destroy(l);                                  /*销毁 Libnet*/
    return (EXIT_SUCCESS);
bad:
    libnet_destroy(l);
    return (EXIT_FAILURE);
}
void
usage(char *name)                                       /*使用方法*/
{
    fprintf(stderr, "usage: %s -s source_ip -d destination_ip\n ", name);
}
```

此程序构造的 ICMP 超时数据包，在 ICMP 超时报文中添加了 IPv4 协议头和负载内容。负载内容为 0x11, 0x11, 0x22, 0x22, 0x00, 0x08, 0xc6, 0xa5。在以太网协议块中构造的 MAC 源地址 enet_src 和目的地址 enet_dst，在 libnet_test.h 中定义为：

```
u_char enet_src[6] = {0x0d, 0x0e, 0x0a, 0x0d, 0x00, 0x00};
u_char enet_dst[6] = {0x00, 0x10, 0x67, 0x00, 0xb1, 0x86};
```

由此程序构造的数据包，其内容和协议格式如图 5-55 所示，总长度为 70 字节，其中以太网协议头为 14 字节，IP 协议头为 20 字节。整个 ICMP 超时报文为 36 字节，其 ICMP 头部为 8 字节，IP 头部为 20 字节，负载为 8 字节。

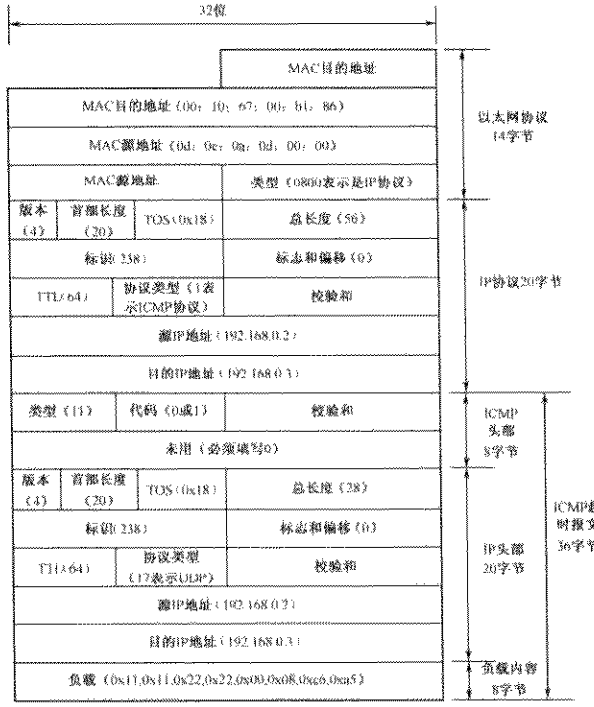

图 5-55　ICMP 超时数据包的结构

该程序的运行结果如图 5-56 所示。

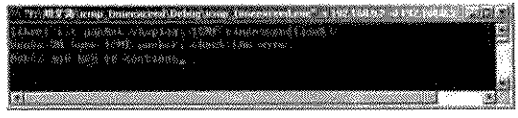

图 5-56　构造 ICMP 超时数据包的运行结果

为了检测构造的数据包是否正确，使用协议分析工具 Analyzer 进行分析，其分析的结果如图 5-57 所示。

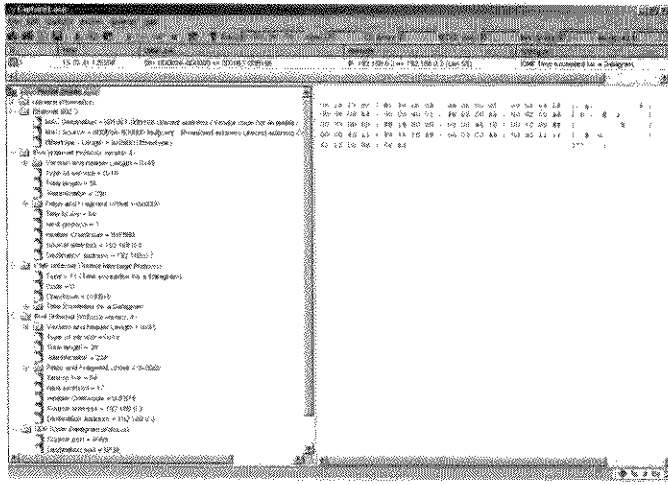

图 5-57　Analyzer 捕获 ICMP 超时数据包的运行结果

利用 Analyzer 的功能，将 Analyzer 捕获的数据包的详细信息展示如下：

```
⊟       General information
    ⊟ ●     Number              = 1
    ⊟ ●     Packet Length       = 70
    ⊟ ●     Captured Length     = 70
    ⊟ ●     Captured Time       = 15:32:31.125206
⊟       Ethernet 802.3
    ⊟ ●     MAC Destination     = 001067-00B186 Unicast address (Vendor code not available)
    ⊟ ●     MAC Source          = 000B0A-000000 Multicast - Broadcast address Unicast
                                  address (Vendor code not available)
    ⊟ ●     Ethertype - Length = 0x0800 (Ethertype)
⊟       IPv4 (Internet Protocol version 4)
    ⊟ ●     Version and header Length= 0x45
        ⊟ ●     Version         = 4
        ⊟ ●     Header length = 20 (field value = 5)
    ⊟ ●     Type of service     = 0x18
    ⊟ ●     Total length        = 56
    ⊟ ●     Identification      = 238
    ⊟ ●     Flags and Fragment offset    = 0x0000
        ⊟ ●     Unused          = 0b0...............
        ⊟ ●     Don't fragment= 0b.0..............
        ⊟ ●     More fragments= 0b..0.............
        ⊟ ●     Fragment offset     = 0 (field value = 0)
    ⊟ ●     Time to live        = 64
    ⊟ ●     Next protocol       = 1
```

```
⊟ ● Header Checksum     = 0xF869
⊟ ● Source address      = 192.168.0.2
⊟ ● Destination address     = 192.168.0.3
⊟   ICMP (Internet Control Message Protocol)
⊟ ● Type = 11 (Time exceeded for a Datagram)
⊟ ● Code = 0
⊟ ● Checksum            = 0xFB1E
⊟ ● Time Exceeded for a Datagram=
    ⊟ ● Unused (must be zero)   = 0
⊟   IPv4 (Internet Protocol version 4)
⊟ ● Version and header Length   = 0x45
    ⊟ ● Version         = 4
    ⊟ ● Header length = 20 (field value = 5)
⊟ ● Type of service     = 0x18
⊟ ● Total length        = 28
⊟ ● Identification      = 238
⊟ ● Flags and Fragment offset   = 0x0000
    ⊟ ● Unused          = 0b0...............
    ⊟ ● Don't fragment= 0b.0.............
    ⊟ ● More fragments= 0b..0............
    ⊟ ● Fragment offset     = 0 (field value = 0)
⊟ ● Time to live        = 64
⊟ ● Next protocol       = 17
⊟ ● Header Checksum     = 0xF875
⊟ ● Source address      = 192.168.0.3
⊟ ● Destination address     = 192.168.0.2
⊟   UDP (User Datagram protocol)
⊟ ● Source port         = 4369
⊟ ● Destination port    = 8738
⊟ ● Payload length      = 8
⊟ ● Checksum            = 0xC6A5
```

可以看到，捕获的数据包信息与程序构造的数据包的信息是一样的，由此证实程序构造的数据包是正确的。

3. 构造 ICMP 重定向数据包

ICMP 重定向数据包有四种类型，根据 ICMP 代码的不同进行区分，其中代码 0 表示网络重定向，代码 1 表示主机重定向，代码 2 表示服务类型和网络重定向，代码 3 表示服务类型和主机重定向。在 Libnet 中，ICMP_REDIRECT 代表 ICMP 重定向数据包，并且 ICMP_REDIRECT_NET 代表网络重定向，ICMP_REDIRECT_HOST 代表主机重定向，ICMP_REDIRECT_TOSNET 代表服务类型和网络重定向，ICMP_REDIRECT_TOSHOST 代表服务类型和主机重定向。

ICMP 重定向的协议格式如图 5-58 所示。

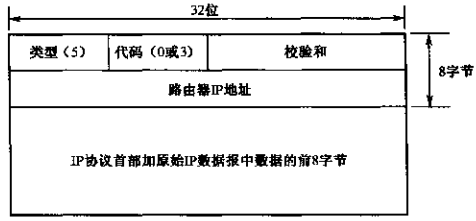

图 5-58　ICMP 重定向协议格式

根据 Libnet 构造数据包的规律，构造 ICMP 重定向数据包的过程也应该先构造高层协议块，然后依次从高到低构造各个协议块。其构造的流程如图 5-59 所示。

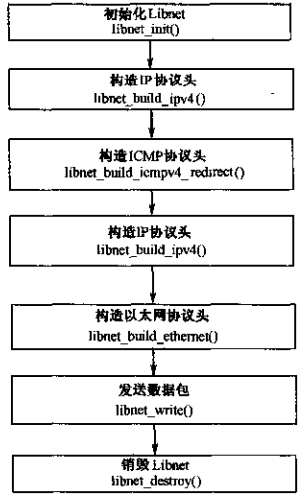

图 5-59　ICMP 重定向数据包的构造流程

下面详细介绍 icmp_redirect.c 程序，它由 Mike D. Schiffma 和 Alberto Ornaghi 编写。在这里，对其核心代码进行详细讲解，并对其结果进行详细分析。其部分核心代码如下：

```
#if (HAVE_CONFIG_H)
#include "../include/config.h"
#endif
#include "./libnet_test.h"
int
main(int argc, char **argv)
{
    int c;                                  /*数据包长度*/
    libnet_t *l;                            /*Libnet 句柄*/
    libnet_ptag_t t;                        /*协议块标记*/
```

```
u_long src_ip, dst_ip, gw_ip;           /*源 IP 和目的 IP*/
u_char payload[8] = {0x11, 0x11, 0x22, 0x22, 0x00, 0x08, 0xc6, 0xa5};        /*负载*/
   u_long payload_s = 8;                /*负载长度*/
char errbuf[LIBNET_ERRBUF_SIZE];
printf("libnet 1.1 packet shaping: ICMP redirect[link]\n");
l = libnet_init(                        /*初始化 Libnet*/
   LIBNET_LINK,                         /*Libnet 类型*/
   NULL,                                /*网络设备接口*/
   errbuf                               /*错误信息*/
         );
if (l == NULL)                          /*出错信息*/
{
   fprintf(stderr, "libnet_init() failed: %s", errbuf);
   exit(EXIT_FAILURE);
}
src_ip = 0;
dst_ip = 0;
gw_ip = 0;
while((c = getopt(argc, argv, "d:s:g:")) != EOF)
{
   switch (c)
   {
      case 'd':                         /*-d 表示目的 IP 地址*/
         if ((dst_ip = libnet_name2addr4(l, optarg, LIBNET_RESOLVE)) == -1)
                                        /*对地址类型进行转换*/
         {
            fprintf(stderr, "Bad destination IP address: %s\n", optarg);
            exit(1);
         }
         break;
      case 's':                         /* -s 表示源 IP 地址*/
         if ((src_ip = libnet_name2addr4(l, optarg, LIBNET_RESOLVE)) == -1)
         {
            fprintf(stderr, "Bad source IP address: %s\n", optarg);
            exit(1);
         }
         break;
      case 'g':                         /* -g 表示路由器 IP 地址*/
         if ((gw_ip = libnet_name2addr4(l, optarg, LIBNET_RESOLVE)) == -1)
         {
            fprintf(stderr, "Bad gateway IP address: %s\n", optarg);
            exit(1);
         }
         break;
   }
```

```
}
if (!src_ip || !dst_ip || !gw_ip)
{
    usage(argv[0]);
    exit(EXIT_FAILURE);
}
t = libnet_build_ipv4(                          /*构造 IPv4 协议块*/
        LIBNET_IPV4_H + payload_s,              /*长度，即 IP 协议头长度加负载长度*/
        IPTOS_LOWDELAY | IPTOS_THROUGHPUT,      /*服务质量*/
        0x42,
            //0xee,                             /*IP 标识*/
        0,                                      /*IP 偏移*/
        64,                                     /*生存时间*/
        IPPROTO_UDP,                            /*上层协议为 ICMP 协议*/
        0,                                      /*校验和*/
        dst_ip,                                 /*源 IP 地址*/
        src_ip,                                 /*目的 IP 地址 */
        payload,                                /*负载 */
        payload_s,                              /*负载长度*/
        1,                                      /*Libnet 句柄*/
        0                                       /*协议块标记*/
        );
 if (t == -1)                                   /*出错处理*/
 {
    fprintf(stderr, "Can't bnild error IP header: %s\n",
            libnet_geterror(l));
    goto bad;
 }
t = libnet_build_icmpv4_redirect(       /*构造 ICMP 重定向协议块*/
        ICMP_REDIRECT,                  /*ICMP 类型，ICMP_REDIRECT 表示重定向*/
        ICMP_REDIRECT_HOST,             /*ICMP 代码，ICMP_REDIRECT_HOST 表示主机重定向*/
        0,                              /*校验和*/
        gw_ip,                          /*路由 IP*/
        NULL,                           /*负载*/
        0,                              /*负载长度*/
        1,                              /*Libnet 句柄*/
        0                               /*协议块标记*/
            );
if (t == -1)
 {
    fprintf(stderr, "Can't build ICMP header: %s\n", libnet_geterror(l));
    goto bad;
 }
 t = libnet_build_ipv4(                         /*构造 IP 协议块标记*/
            LIBNET_IPV4_H + LIBNET_ICMPV4_REDIRECT_H +
```

```
            LIBNET_IPV4_H + payload_s,              /*长度，注意此时的长度不要弄错*/
            IPTOS_LOWDELAY | IPTOS_THROUGHPUT,      /*服务质量*/
            0xee,                                   /*IP 标识*/
            0,                                      /*IP 偏移*/
            64,                                     /*生存时间*/
            IPPROTO_ICMP,                           /*上层协议类型*/
            0,                                      /*校验和*/
            src_ip,                                 /*源 IP 地址*/
            dst_ip,                                 /*目的 IP 地址*/
            NULL,                                   /*负载*/
            0,                                      /*负载长度*/
            1,                                      /*Libnet 句柄*/
            0                                       /*协议块标记*/
                );
if (t == -1)
{
    fprintf(stderr, "Can't build IP header: %s\n", libnet_geterror(l));
    goto bad;
}
t = libnet_build_ethernet(                          /*构造以太网协议块*/
        enet_dst,                                   /*MAC 目的地址*/
        enet_src,                                   /*以太网源地址*/
        ETHERTYPE_IP,                               /*上层协议类型*/
        NULL,                                       /*负载*/
        0,                                          /*负载大小*/
        1,                                          /*Libnet 句柄*/
        0                                           /*协议块标记*/
            );
if (t == -1)
{
    fprintf(stderr, "Can't build ethernet header: %s\n", libnet_geterror(l));
    goto bad;
}
c = libnet_write(l);                                /*发送数据包*/
if (c == -1)
{
    fprintf(stderr, "Write error: %s\n", libnet_geterror(l));
    goto bad;
}
else
{
    fprintf(stderr, "Wrote %d byte ICMP packet; check the wire.\n", c);
}
libnet_destroy(l);                                  /*销毁 Libnet*/
return (EXIT_SUCCESS);
```

```
bad:
    libnet_destroy(1);
    return (EXIT_FAILURE);
}
void
usage(char *name)                                    /*使用方法*/
{
    fprintf(stderr, "usage: %s -s source_ip -d destination_ip -g gateway_ip\n ", name);
}
```

在 Visual C++ 中设置运行参数为

```
-s 192.168.0.2
-d 192.168.0.3
-g 192.168.0.4
```

运行此程序，其结果如图 5-60 所示。

图 5-60　构造 ICMP 重定向数据包的运行结果

可见，该程序发送了一个 70 字节长的 ICMP 重定向数据包，其中 14 字节的以太网协议，20 字节的 IP 协议头，8 字节的 ICMP 重定向头，20 字节的 IP 协议首部和 8 字节的负载内容。

使用工具 Analyzer 进行分析，捕获一个 ICMP 数据包，其结果如图 5-61 所示。

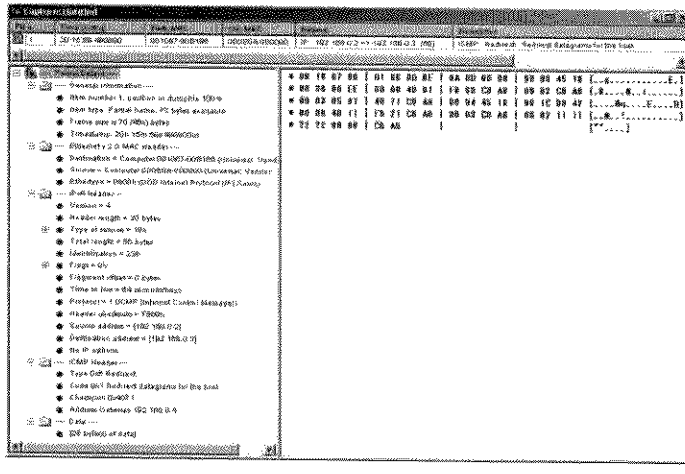

图 5-61　Analyzer 捕获 ICMP 重定向数据包的运行结果

从图 5-61 中可以看到，捕获的数据包信息与程序构造和发送的数据包的信息是相同的。利用 Analyzer 可输出如下信息：

```
----- Packet Details -----
----- General Information -----
Item number 1, position in dumpfile 100%
Item type: Partial frame, 70 bytes available
Frame size is 70 (46x) bytes
Timestamp: 20h:15m:59s:480800us
----- Ethernet v.2.0 MAC Header ----- [0-13]
Destination = Computer 001067-00B186 (Universal; Vendor: ???) - [0-13]
Source = Computer 0D0E0A-0D0000 (Universal; Vendor: ???) - [6-11]
Ethertype = 0800h (DOD Internet Protocol (IP) Xerox) - [12-13]
----- IPv4 Header ----- [14-33]
Version = 4 - (14-14)
Header length = 20 bytes - (14-14)
Type of service = 18h - [15-15]
 000. .... = priority 0: routine - (15-15)
 ...1 .... = low delay - (15-15)
 .... 1... = high throughput - (15-15)
 .... .0.. = normal reliability - (15-15)
 .... ..0. = normal monetary cost - (15-15)
 .... ...0 = Reserved bit, for future usage, which has to be 0 - (15-15)
Total length = 56 bytes - [16-17]
Identification = 238 - [18-19]
Flags = 0h - (20-20)
 0... .... = must be 0 - (20-20)
 .0.. .... = may fragment - (20-20)
 ..0. .... = last fragment - (20-20)
Fragment offset = 0 bytes - (20-21)
Time to live = 64 seconds/hops - [22-22]
Protocol = 1 (ICMP [Internet Control Message]) - [23-23]
Header checksum = F869h - [24-25]
Source address = [192.168.0.2] - [26-29]
Destination address = [192.168.0.3] - [30-33]
No IP options - [14-14]
----- ICMP Header ----- (34-37]
Type 0x5 Redirect. - [34-34]
Code 0x1 Redirect datagrams for the host. - [35-35]
Checksum 0x4071 - [36-37]
Address Gateway 192.168.0.4 - (38-41]
----- Data ----- [42-69]
[28 byte(s) of data]
```

```
=============================================================================
* 00 10 67 00 | B1 86 0D 0E | 0A 0D 00 00 | 08 00 45 18 [..g...........E.]
* 00 38 00 EE | 00 00 40 01 | F8 69 C0 A8 | 00 02 C0 A8 [.8....@..i......]
* 00 03 05 01 | 40 71 C0 A8 | 00 04 45 18 | 00 1C 00 42 [....@q....E....B]
* 00 00 40 11 | F9 21 C0 A8 | 00 03 C0 A8 | 00 02 11 11 [..@..!..........]
* 22 22 00 08 | C6 A5      |            |            [""....]
#############################################################################
```

4. 构造 ICMP 时间戳数据包

ICMP 时间戳数据包有两种类型，一种是时间戳请求数据包，其 ICMP 类型值为 13；另一种是时间戳应答数据包，其 ICMP 类型值为 14。在 Libnet 中以 ICMP_TSTAMP 表示时间戳请求，以 ICMP_TSTAMPREPLY 表示时间戳应答。

ICMP 时间戳协议格式如图 5-62 所示。

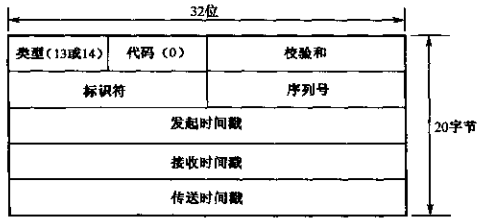

图 5-62　ICMP 时间戳协议格式

以 Libnet 开发包中的程序 timestamp.c 进行讲解，它实现了构造和发送 ICMP 时间戳数据包的功能。其核心代码如下。

```
    #if (HAVE_CONFIG_H)
#include "../include/config.h"
#endif
#include "./libnet_test.h"
int
main(int argc, char **argv)
{
    int c;
    libnet_t *l;
    libnet_ptag_t t;
    u_long src_ip, dst_ip;
    char errbuf[LIBNET_ERRBUF_SIZE];
    printf("libnet 1.1 packet shaping: ICMP timestamp[raw]\n");
    l = libnet_init(                                /*初始化 Libnet*/
        LIBNET_RAW4,                                /*Libnet 类型*/
        // NULL,
        "\\Device\\NPF_{5973DFB0-AD5B-4065-A22B-C61FFB64624F}",
            /*网络设备接口，样例中是 NULL，在此应该给一个固定的数值*/
```

```
            errbuf                                  /*错误信息*/
            );
if (l == NULL)
{
    fprintf(stderr, "libnet_init() failed: %s", errbuf);
    exit(EXIT_FAILURE);
}
src_ip = 0;
dst_ip = 0;
while((c = getopt(argc, argv, "d:s:")) != EOF)
{
    switch (c)
    {
        case 'd':
            if ((dst_ip = libnet_name2addr4(l, optarg, LIBNET_RESOLVE)) == -1)
            {
                fprintf(stderr, "Bad destination IP address: %s\n", optarg);
                exit(1);
            }
            break;
        case 's':
            if ((src_ip = libnet_name2addr4(l, optarg, LIBNET_RESOLVE)) == -1)
            {
                fprintf(stderr, "Bad source IP address: %s\n", optarg);
                exit(1);
            }
            break;
    }
}
if (!src_ip || !dst_ip)
{
    usage(argv[0]);
    exit(EXIT_FAILURE);
}
t = libnet_build_icmpv4_timestamp(                  /*构造 ICMP 时间戳协议块*/
        ICMP_TSTAMP,                                /*ICMP 类型，ICMP_TSTAMP 表示时间戳*/
        0,                                          /*ICMP 代码，为 0*/
        0,                                          /*校验和*/
        242,                                        /*ICMP 标识符*/
        424,                                        /*ICMP 序列码*/
        1000,                                       /*发起时间*/
        2000,                                       /*接收时间*/
        3000,                                       /*传送时间*/
```

```
        NULL,                                   /*负载*/
        0,                                      /*负载大小*/
        1,                                      /*Libnet 句柄*/
        0                                       /*协议块标记*/
            };
if (t == -1)
{
    fprintf(stderr, "Can't build ICMP header: %s\n", libnet_geterror(l));
    goto bad;
}
t = libnet_build_ipv4(                          /*构造 IPv4 协议块*/
        LIBNET_IPV4_H + LIBNET_ICMPV4_TS_H,     /*长度*/
        0,                                      /*服务质量*/
        242,                                    /*IP 标识*/
        0,                                      /*IP 偏移*/
        64,                                     /*生存时间*/
        IPPROTO_ICMP,                           /*上层协议类型*/
        0,                                      /*校验和*/
        src_ip,                                 /*源 IP 地址*/
        dst_ip,                                 /*目的 IP 地址*/
        NULL,                                   /*负载*/
        0,                                      /*负载大小*/
        1,                                      /*Libnet 句柄*/
        0                                       /*协议块标记*/
            };
if (t == -1)
{
    fprintf(stderr, "Can't build IP header: %s\n", libnet_geterror(l));
    goto bad;
        }
c = libnet_write(l);                            /*发送数据包*/
        if (c == -1)
{
    fprintf(stderr, "Write error: %s\n", libnet_geterror(l));
    goto bad;
}
else
{
    fprintf(stderr, "Wrote %d byte ICMP packet; check the wire.\n", c);
}
libnet_destroy(l);                              /*销毁 Libnet*/
return (EXIT_SUCCESS);
bad:
```

```
    libnet_destroy(l);
    return (EXIT_FAILURE);
}
void
usage(char *name)                                    /*使用方法*/
{
    fprintf(stderr, "usage: %s -s source_ip -d destination_ipin ", name);
}
```

设置运行参数如下：

```
-s 192.168.0.2
-d 192.168.0.3
```

运行此程序，其结果如图 5-63 所示。

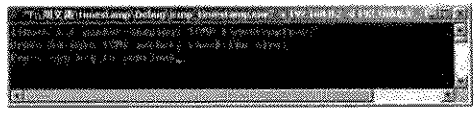

图 5-63　构造 ICMP 时间戳程序的运行结果

此程序构造和发送了一个 54 字节长的 ICMP 时间戳请求数据包，包括 14 字节的以太网头、20 字节的 IP 协议头和 20 字节的 ICMP 协议头。

用协议分析工具 Analyzer 进行分析，其结果如图 5-64 所示。

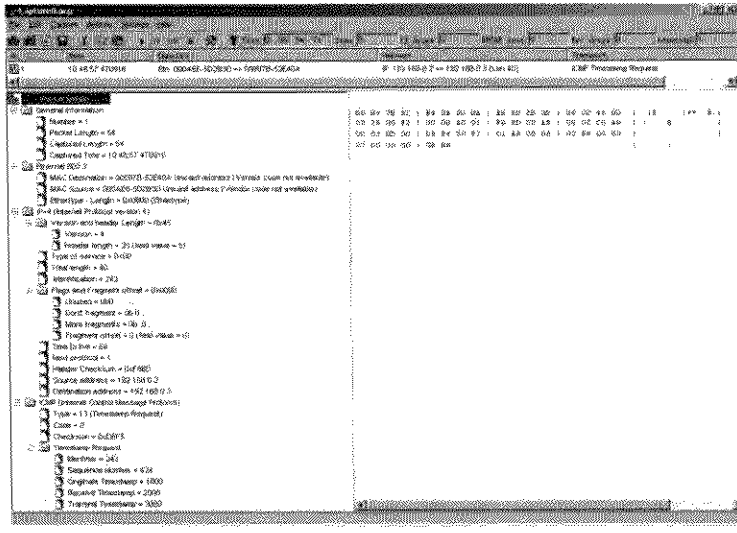

图 5-64　捕获 ICMP 时间戳数据包

分析捕获的 ICMP 数据包，其详细信息如图所示 5-65，每个协议头的字段值都能够显示出来，对比构造的数据包的信息，可以看出它们与程序构造的数值是相同的，说明构造和发送的数据包是正确的。

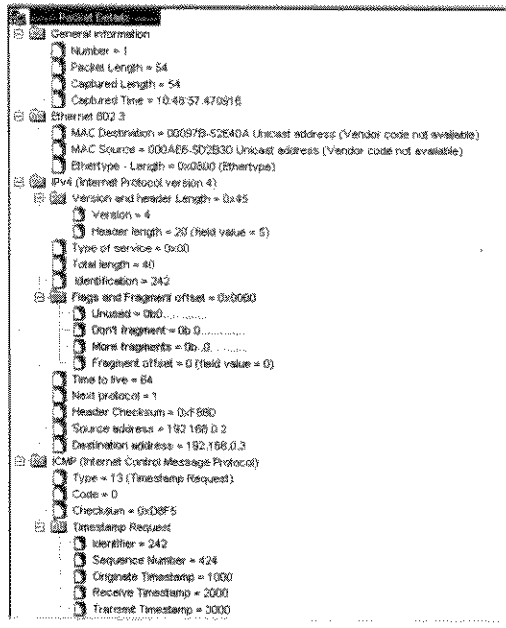

图 5-65　ICMP 时间戳请求数据包的详细信息

5. 构造 ICMP 不可达数据包

ICMP 不可达数据包有 16 种类型，由 ICMP 代码的值来确定，其值分别为 0~15，所代表的含义如下：

- 0 表示网络不可达，在 Libnet 中以 ICMP_UNREACH_NET 表示；
- 1 表示主机不可达，在 Libnet 中以 ICMP_UNREACH_HOST 表示；
- 2 表示协议不可达，在 Libnet 中以 ICMP_UNREACH_PROTOCOL 表示；
- 3 表示端口不可达，在 Libnet 中以 ICMP_UNREACH_PORT 表示；
- 4 表示需要进行分片但设置了不分片比特，在 Libnet 中以 ICMP_UNREACH_NEEDFRAG 表示；
- 5 表示源站选路失败，在 Libnet 中以 ICMP_UNREACH_SRCFAIL 表示；
- 6 表示目的网络不认识，在 Libnet 中以 ICMP_UNREACH_NET_UNKNOWN 表示；
- 7 表示目的主机不认识，在 Libnet 中以 ICMP_UNREACH_HOST_UNKNOWN 表示；
- 8 表示源主机被隔离，在 Libnet 中以 ICMP_UNREACH_ISOLATED 表示；
- 9 表示目的网络被强制禁止，在 Libnet 中以 ICMP_UNREACH_NET_PROHIB 表示；

- 10 表示目的主机被强制禁止，在 Libnet 中以 ICMP_UNREACH_HOST_PROHIB 表示；
- 11 表示由于服务类型 TOS，网络不可达，在 Libnet 中以 ICMP_UNREACH_TOSNET 表示；
- 12 表示由于服务类型 TOS，主机不可达，在 Libnet 中以 ICMP_UNREACH_TOSHOST 表示；
- 13 表示由于过滤，通信被强制禁止，在 Libnet 中以 ICMP_UNREACH_FILTER_PROHIB 表示；
- 14 表示主机越权，在 Libnet 中以 ICMP_UNREACH_HOST_PRECEDENCE 表示；
- 15 表示优先权中止生效，在 Libnet 中以 ICMP_UNREACH_PRECEDENCE_CUTOFF 表示。

ICMP 不可达报文格式如图 5-66 所示。

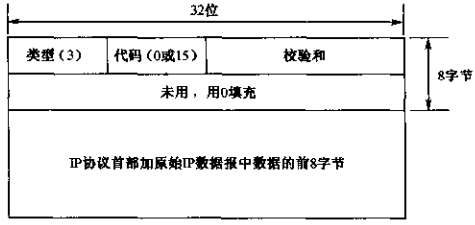

图 5-66 ICMP 不可达报文格式

以 Libnet 开发包中的程序 icmp_unreach.c 进行讲解，其部分核心源代码如下：

```
    #if (HAVE_CONFIG_H)
#include "../include/config.h"
#endif
#include "./libnet_test.h"
int
main(int argc, char **argv)
{
    int c, i;
    libnet_t *l = NULL;                                    /*Libnet 句柄*/
    libnet_ptag_t ip_err = 0, icmp = 0, ip = 0, eth = 0;   /*协议标记*/
    u_long src_ip, dst_ip;                                 /*源 IP 地址和目的 IP 地址*/
    u_char payload[8] = {0x11, 0x11, 0x22, 0x22, 0x00, 0x08, 0xc6, 0xa5};   /*负载*/
    u_long payload_s = 8;                                  /*负载长度*/
    int mode = LIBNET_LINK;                                /*Libnet 类型*/
    char errbuf[LIBNET_ERRBUF_SIZE];
    printf("libnet 1.1 packet shaping: ICMP unreachable[link]\n");
    src_ip = 0;
    dst_ip = 0;
    while((c = getopt(argc, argv, "d:s:r")) != EOF)
    {
```

```
        switch (c)
        {
            case 'd':                                   /*-d 表示目的IP地址*/
                if ((dst_ip = libnet_name2addr4(l, optarg, LIBNET_RESOLVE)) == -1)
                {
                    fprintf(stderr, "Bad destination IP address: %s\n", optarg);
                    exit(1);
                }
                break;
            case 's':                                   /*-s 表示源IP地址*/
                if ((src_ip = libnet_name2addr4(l, optarg, LIBNET_RESOLVE)) == -1)
                {
                    fprintf(stderr, "Bad source IP address: %s\n", optarg);
                    exit(1);
                }
                break;
            case 'r':                                   /*-r 表示Libnet类型*/
                mode = LIBNET_RAW4;
                break;
        }
    }
    if (!src_ip || !dst_ip)
    {
        usage(argv[0]);
        exit(EXIT_FAILURE);
    }
    l = libnet_init(                                    /*初始化Libnet*/
        mode,                                           /*Libnet类型*/
        NULL,                                           /*网络设备接口*/
        errbuf                                          /*错误信息*/
        );
    if (l == NULL)
    {
        fprintf(stderr, "libnet_init() failed: %s", errbuf);
        exit(EXIT_FAILURE);
    }
    for (i=0; i<255; i++)                               /*总共发送255个数据包*/
    {
        ip_err = libnet_build_ipv4(                     /*构造IP协议块*/
        LIBNET_IPV4_H + payload_s,                      /*长度*/
        IPTOS_LOWDELAY | IPTOS_THROUGHPUT,              /*服务质量*/
        (u_int16_t)i,                                   /*IP标识*/
        0,                                              /*偏移*/
```

```
        64,                                     /*生存时间*/
        IPPROTO_UDP,                            /*上层协议类型*/
        0,                                      /*校验和*/
        dst_ip,                                 /*源 IP 地址*/
        src_ip,                                 /*目的 IP 地址*/
        payload,                                /*负载*/
        payload_s,                              /*负载大小*/
        l,                                      /*Libnet 句柄*/
        ip_err                                  /*协议标记*/
            );
if (ip_err == -1)
{
    fprintf(stderr, "Can't build error IPv4 header: %s\n",
            libnet_geterror(l));
    goto bad;
}
icmp = libnet_build_icmpv4_unreach(             /*构造 ICMP 协议块*/
        ICMP_UNREACH,                           /*ICMP 类型，ICMP_UNREACH 表示 ICMP 不可达*/
        ICMP_UNREACH_PORT,                      /*ICMP 代码，ICMP_UNREACH_PORT 代表端口不可达*/
        0,                                      /*校验和*/
        NULL,                                   /*负载*/
        0,                                      /*负载大小*/
        l,                                      /*Libnet 句柄*/
        icmp                                    /*协议标记*/
            );
if (icmp == -1)
{
    fprintf(stderr, "Can't build ICMP header: %s\n", libnet_geterror(l));
    goto bad;
}
ip = libnet_build_ipv4(                         /*构造 IP 协议块*/
        LIBNET_IPV4_H + LIBNET_ICMPV4_UNREACH_H +
        LIBNET_IPV4_H + payload_s,              /*长度*/
        IPTOS_LOWDELAY | IPTOS_THROUGHPUT,      /*服务质量*/
        (u_int16_t)i + 1,                       /*IP 标识*/
        0,                                      /*偏移*/
        64,                                     /*生存时间*/
        IPPROTO_ICMP,                           /*上层协议类型为 ICMP*/
        0,                                      /*校验和*/
        src_ip,                                 /*源 IP 地址*/
        dst_ip,                                 /*目的 IP 地址*/
        NULL,                                   /* 负载*/
        0,                                      /*负载大小*/
```

```
        l,                                  /*Libnet 句柄*/
        ip                                  /*协议标记*/
            );
    if (ip == -1)
    {
        fprintf(stderr, "Can't build IP header: %s\n", libnet_geterror(l));
        goto bad;
    }
    if (mode == LIBNET_LINK) {              /*判断 Libnet 类型*/
    eth = libnet_build_ethernet(            /*构造 Ethernet 协议块*/
        enet_dst,                           /*MAC 目的地址*/
        enet_src,                           /*MAC 源地址*/
        ETHERTYPE_IP,                       /*上层协议类型为 IP 协议*/
        NULL,                               /*负载 */
        0,                                  /*负载大小 */
        l,                                  /*Libnet 句柄*/
        eth                                 /*协议标记*/
            );
    if (eth == -1)
    {
        fprintf(stderr, "Can't build ethernet header: %s\n", libnet_geterror(l));
        goto bad;
    }
    }
    libnet_diag_dump_pblock(l);             /*测试函数，输出所有协议块的信息*/
        libnet_diag_dump_context(l);        /*测试函数，输出 Libnet 句柄的信息。注意，此函数是
                                              另外添加的，原程序没有*/
    c = libnet_write(l);                    /*发送数据包*/
    if (c == -1)
    {
        fprintf(stderr, "Write error: %s\n", libnet_geterror(l));
        goto bad;
    }
    else
    {
        fprintf(stderr, "Wrote %d byte ICMP packet; check the wire.\n", c);
    }
}
    libnet_destroy(l);                      /*销毁 Libnet*/
    return (EXIT_SUCCESS);
bad:
    libnet_destroy(l);
    return (EXIT_FAILURE);
```

```
}
void
usage(char *name)
{
    fprintf(stderr, "usage: %s [-r] -s source_ip -d destination_ip\n ", name);
}
```

在本程序中使用了一个测试函数：

```
libnet_diag_dump_pblock ()
```

此函数的功能是输出一个数据包所有协议块的详细信息。此程序总共构造了 255 个数据包，在此只给出第一个数据包的详细信息，其他数据包的信息以此类推。

此程序的第一个数据包的输出结果如下：

```
    libnet 1.1 packet shaping: ICMP unreachable[link]
pblock type:    ip data
ptag number:    2
IP offset:      28
pblock address: 00E11C90
next pblock     00E11DD0 (ipv4 header)
prev pblock     00000000
buf:            111122220008c6a5
buffer length:  8
checksum flag:  No
bytes copied:   8

pblock type:    ipv4 header
ptag number:    1
IP offset:      28
pblock address: 00E11DD0
next pblock     00E11C00 (icmpv4 unreachable header)
prev pblock     00E11C90 (ip data)
buf:            4518001c00000000401100000c0a80003c0a80002
buffer length:  20
checksum flag:  Yes
chksum length:  20
bytes copied:   20

pblock type:    icmpv4 unreachable header
ptag number:    3
IP offset:      56
pblock address: 00E11C00
next pblock     00E11B70 (ipv4 header)
prev pblock     00E11DD0 (ipv4 header)
buf:            0303000000000000
```

```
buffer length:  8
checksum flag:  Yes
chksum length:  36
bytes copied:   8

pblock type:    ipv4 header
ptag number:    4
IP offset:      56
pblock address: 00E11B70
next pblock     00E11AE0 (ethernet header)
prev pblock     00E11C00 (icmpv4 unreachable header)
buf:            4518003800010000400100000c0a80002c0a80003
buffer length:  20
checksum flag:  Yes
chksum length:  20
bytes copied:   20

pblock type:    ethernet header
ptag number:    5
IP offset:      0
pblock address: 00E11AE0
next pblock     00000000
prev pblock     00E11B70 (ipv4 header)
buf:            00106700b1860d0e0a0d000p0800
buffer length:  14
checksum flag:  No
bytes copied:   14
fd:       0
injection type:   LIBNET_LINK
    pblock start: 00E11C90
    pblock end:   00E11AE0
link type:      1
link offset:    14
aligner :       0
device : \Device\NPF_{5973DFB0-AD5B-4065-A22B-C61FFB64624F}
packets sent:   0
packet errors:  0
bytes written:  0
ptag  state:    5
context label: cardshark
last errbuf:
total size: 70
Wrote 70 byte ICMP packet; check the wire.
```

可以看到，函数 libnet_diag_dump_pblock() 输出了一个数据包的详细信息，这主要是为了测试程序构造数据包的正确性。

为了更好地查看 ICMP 端口不可达数据包的构造过程，在上述程序中加上一条语句：

```
libnet_diag_dump_context(l);
```

此函数输出一个 Libnet 句柄的详细信息。

根据上面运行的结果，此程序所构造的 ICMP 端口不可达数据包，在 Libnet 开发包中的构造机理如图 5-67 所示。其中每个数据包用一个 Libnet 句柄来表示，一个数据包又由很多协议块构成，这些协议块组成一个链表，此链表由 Libnet 中的成员来指定。

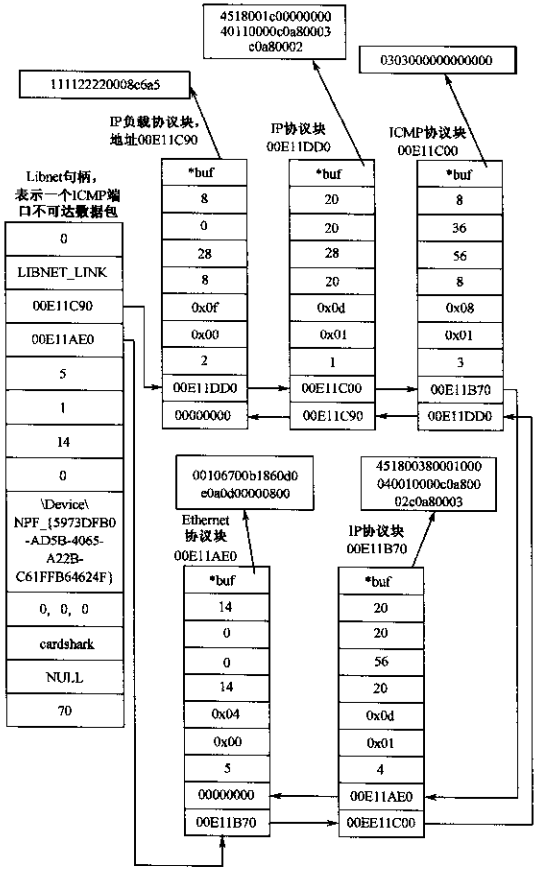

图 5-67　一个 ICMP 端口不可达数据包的构造机理

为了检验构造的 ICMP 不可达数据包是否正确，使用 Analyzer 进行捕获，其结果如图

5-68 所示。

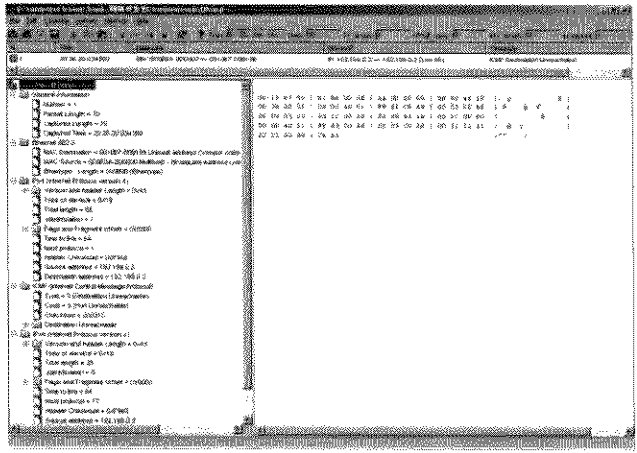

图 5-68　Analyzer 捕获 ICMP 不可达数据包的结果

由 Analyzer 进行处理，输出数据包的详细信息如下：

```
⊟ General information
  ⊟ ● Number              = 1
  ⊟ ● Packet Length       = 70
  ⊟ ● Captured Length     = 70
  ⊟ ● Captured Time       = 20:34:12.482996
⊟ Ethernet 802.3
  ⊟ ● MAC Destination     = 001067-00B186 Unicast address(Vendor code not available)
  ⊟ ● MAC Source          = 0D0E0A-0D0000 Multicast - Broadcast address Unicast
                            address (Vendor code not available)
  ⊟ ● Ethertype - Length = 0x0800 (Ethertype)
⊟ IPv4 (Internet Protocol version 4)
  ⊟ ● Version and Header Length   = 0x45
    ⊟ ● Version                   = 4
    ⊟ ● Header length             = 20 (field value = 5)
  ⊟ ● Type of service             = 0x18
  ⊟ ● Total length                = 56
  ⊟ ● Identification              = 1
  ⊟ ● Flags and Fragment offset   = 0x0000
    ⊟ ● Unused                    = 0b0...............
    ⊟ ● Don't fragment            = 0b.0..............
    ⊟ ● More fragments            = 0b..0.............
    ⊟ ● Fragment offset           = 0 (field value = 0)
  ⊟ ● Time to live                = 64
```

```
⊟ ● Next protocol                = 1
⊟ ● Header Checksum              = 0xF956
⊟ ● Source address               = 192.168.0.2
⊟ ● Destination address          = 192.168.0.3
⊟   ICMP (Internet Control Message Protocol)
⊟ ● Type                         = 3 (Destination Unreachable)
⊟ ● Code                         = 3 (Port Unreachable)
⊟ ● Checksum                     = 0x031C
⊟ ● Destination Unreachable      =
    ⊟ ● Unused (must be zero)    = 0
⊟   IPv4 (Internet Protocol version 4)
⊟ ● Version and header Length    = 0x45
    ⊟ ● Version                  = 4
    ⊟ ● Header length            = 20 (field value = 5)
⊟ ● Type of service              = 0x18
⊟ ● Total length                 = 28
⊟ ● Identification               = 0
⊟ ● Flags and Fragment offset    = 0x0000
    ⊟ ● Unused                   = 0b0..............
    ⊟ ● Don't fragment           = 0b.0.............
    ⊟ ● More fragments           = 0b..0............
    ⊟ ● Fragment offset          = 0 (field value = 0)
⊟ ● Time to live                 = 64
⊟ ● Next protocol                = 17
⊟ ● Header Checksum              = 0xF963
⊟ ● Source address               = 192.168.0.3
⊟ ● Destination address          = 192.168.0.2
⊟   UDP (User Datagram protocol)
⊟ ● Source port                  = 4369
⊟ ● Destination port             = 8738
⊟ ● Payload length               = 8
⊟ ● Checksum                     = 0xC6A5
```

5.6.6 构造 BGP 数据包

BGP 在 TCP/IP 网中实现域间路由。BGP 是一种外部网关协议，即它在多个自治系统或域间执行路由，并与其他 BGP 系统交换路由和可达性信息。在 RFC1771 中规定了四种 BGP 消息类型，分别为初始消息、更新消息、通知消息和 keep-alive 消息。

所有的 BGP 消息类型都使用基本的分组信头。初始、更新和通知消息有附加的域，而 keep-alive 消息只使用基本的分组信头。BGP 信头的协议格式如图 5-69 所示。

每个 BGP 分组都包含信头，其主要目的是标识该分组的功能。信头中标记字段含有认证值；长度字段表示信息的总长度；类型字段表示消息的类型，有四种类型；数据字段是可变化的，包含上层信息。

图 5-69　BGP 信头的协议格式

程序 bgp4_hdr.c 实现的就是构造和发送 BGP 数据包，其核心代码如下。

```
#if (HAVE_CONFIG_H)
#include "../include/config.h"
#endif
#include "./libnet_test.h"
int
main(int argc, char *argv[])
{
    int c;
    libnet_t *l;
    u_long src_ip, dst_ip, length;
    libnet_ptag_t t = 0;
    char errbuf[LIBNET_ERRBUF_SIZE];
    u_char *payload = NULL;
    u_long payload_s = 0;
    u_char marker[LIBNET_BGP4_MARKER_SIZE];                /*BGP 标记*/
    u_char type;                                           /*BGP 类型*/
    printf("libnet 1.1 packet shaping: BGP4 hdr + payload[raw]\n");
    l = libnet_init(                                       /*Libnet 初始化*/
            LIBNET_RAW4,                                   /*Libnet 类型*/
            //NULL,
            /*网络设备，注意在这里因为是 Windows 平台，所以应该改为下述字符串值，根据个人机器而定*/
                "\\Device\\NPF_{5973DFB0-AD5B-4065-A22B-C61FFB64624F}",
            errbuf);
    if (l == NULL)
    {
        fprintf(stderr, "libnet_init() failed: %s", errbuf);
        exit(EXIT_FAILURE);
    }
    src_ip  = 0;
    dst_ip  = 0;
    memset(marker, 0x1, LIBNET_BGP4_MARKER_SIZE);
    type = 0;
    while ((c = getopt(argc, argv, "d:s:t:m:p:")) != EOF)
    {
        switch (c)
        {
          case 'd':
              if ((dst_ip = libnet_name2addr4(l, optarg, LIBNET_RESOLVE)) == -1)
```

```
            {
                fprintf(stderr, "Bad destination IP address: %s\n", optarg);
                exit(EXIT_FAILURE);
            }
            break;
        case 's':
            if ((src_ip = libnet_name2addr4(l, optarg, LIBNET_RESOLVE)) == -1)
            {
                fprintf(stderr, "Bad source IP address: %s\n", optarg);
                exit(EXIT_FAILURE);
            }
            break;
        case 'm':
        memcpy(marker, optarg, LIBNET_BGP4_MARKER_SIZE);
        break;
        case 't':
        type = atoi(optarg);
        break;
        case 'p':
        payload = optarg;
        payload_s = strlen(optarg);
        break;
         default:
            exit(EXIT_FAILURE);
         }
}
if (!src_ip || !dst_ip)
{
    usage(argv[0]);
    exit(EXIT_FAILURE);
}
length = LIBNET_BGP4_HEADER_H + payload_s;
t = libnet_build_bgp4_header(                          /*构造 BGP 协议块*/
            marker,                                    /*标记 */
            length,                                    /*信息总长度 */
            type,                                      /*信息的类型 */
        payload,                                       /*负载 */
        payload_s,                                     /*负载大小*/
        l,                                             /*Libnet 句柄 */
        0);                                            /*协议块标记*/
if (t == -1)
{
    fprintf(stderr, "Can't build BGP4 header: %s\n", libnet_geterror(l));
```

```
    if (n == -1)
    {
        fprintf(stderr, "Write error: %s\n", libnet_geterror(l));
        goto bad;
    }
    else
    {
        fprintf(stderr, "Wrote %d byte TCP packet; check the wire.\n", c);
    }
    libnet_destroy(l);
    return (EXIT_SUCCESS);
    bad:
    libnet_destroy(l);
    return (EXIT_FAILURE);
}
void
usage(char *name)
{
    fprintf(stderr,
        "usage: %s -s source_ip -d destination_ip"
        " [-m marker] [-p payload] [-t BGP type]\n",
        name);
}
```

编译并运行此程序，其运行结果如图 5-70 所示。

图 5-70　构造 BGP 数据包程序的运行结果

使用 Analyzer 工具查看运行结果，捕获到的数据包如图 5-71 所示。

其中，数据包的各个协议块的内容如下：

```
⊟ General information
  ⊟ ● Number                  = 1
  ⊟ ● Packet Length           = 73
  ⊟ ● Captured Length         = 73
  ⊟ ● Captured Time           = 17:47:33.233955
⊟ Ethernet 802.3
  ⊟ ● MAC Destination         = 00097B-52E40A Unicast address (Vendor code not
                                available)
  ⊟ ● MAC Source              = 000AE6-5D2B3D Unicast address (Vendor code not
                                available)
  ⊟ ● Ethertype - Length      = 0x0800 (Ethertype)
```

```
⊟   IPv4 (Internet Protocol version 4)
  ⊟ ● Version and header Length   = 0x45
    ⊟ ● Version                   = 4
    ⊟ ● Header length             = 20 (field value = 5)
  ⊟ ● Type of service             = 0x00
  ⊟ ● Total length                = 59
  ⊟ ● Identification              = 242
  ⊟ ● Flags and Fragment offset   = 0x0000
    ⊟ ● Unused                    = 0b0...............
    ⊟ ● Don't fragment            = 0b.0.............
    ⊟ ● More fragments            = 0b..0............
    ⊟ ● Fragment offset           = 0 (field value = 0)
  ⊟ ● Time to live                = 64
  ⊟ ● Next protocol               = 6
  ⊟ ● Header Checksum             = 0xF875
  ⊟ ● Source address              = 192.168.0.2
  ⊟ ● Destination address         = 192.168.0.3
⊟   TCP (Transmission Control protocol)
  ⊟ ● Source port                 = 26214
  ⊟ ● Destination port            = 179
  ⊟ ● Sequence number             = 16843009
  ⊟ ● Acknowledgement Number      = 33686018
  ⊟ ● Flags                       = 0x5002
    ⊟ ● Header length             = 20 (field value = 5)
    ⊟ ● Reserved (must be zero)   = 0b....000000......
    ⊟ ● Urgent pointer            = 0b..........0.....
    ⊟ ● Ack valid                 = 0b...........0....
    ⊟ ● Push requested            = 0b............0...
    ⊟ ● Reset requested           = 0b.............0..
    ⊟ ● Syn requested             = 0b..............1.
    ⊟ ● Fin requested             = 0b...............0
  ⊟ ● Window size                 = 32767
  ⊟ ● Checksum                    = 0x3443
  ⊟ ● Urgent Pointer              = 0x0000
⊟   BGP4 (Border Gateway Protocol v.4)
  ⊟ ● Authentification            = 0x01010101010101010101010101010101
  ⊟ ● Length                      = 19
  ⊟ ● Type                        = 0x04 (Keepalive)
```

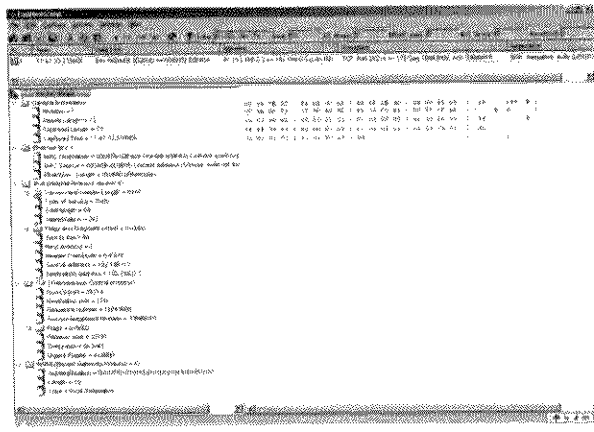

图 5-71　Analyzer 捕获 BGP 数据包的结果

5.6.7　构造 DHCP 数据包

动态主机配置协议（DHCP，Dynamic Host Configuration Protocol）可以自动为局域网中的每一台计算机自动分配 IP 地址，并完成每台计算机的 TCP/IP 协议配置，包括 IP 地址、子网掩码、网关以及 DNS 服务器等。这样，在局域网（特别是大型局域网）中，管理员就不必为每一台计算机手工配置 TCP/IP 协议了，也避免了 IP 地址重复的问题。

DHCP 使 IP 地址可以租用，租用时间不定。通过租期，DHCP 能够自动地重新配置网络。DHCP 支持计算机的静态地址，它是对网络 IP 管理协议 BOOTP 的扩展。

DHCP 的协议格式如图 5-72 所示，其中各字段含义如下：

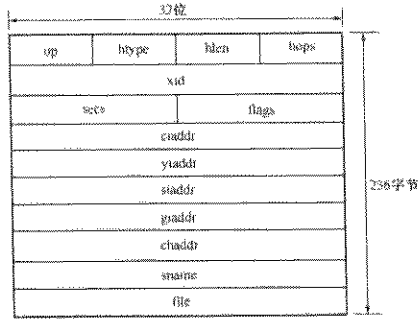

图 5-72　DHCP 的协议格式

- 字段 op 表示消息类型，用 1 表示 BOOTREQUEST，用 2 表示 BOOTREPLY；
- 字段 htype 表示硬件地址类型；
- 字段 hlen 表示硬件地址长度；

- 字段 hops 必须由客户设置为 0；
- 字段 xid 表示操作 ID，这是一个随机数；
- 字段 secs 表示由客户指定的时间；
- 字段 flags 表示标志；
- 字段 ciaddr 表示用户 IP 地址；
- 字段 yiaddr 表示客户 IP 地址；
- 字段 siaddr 表示服务器 IP 地址；
- 字段 giaddr 表示转发代理 IP 地址；
- 字段 chaddr 表示客户硬件地址；
- 字段 sname 表示可选的服务器主机名字；
- 字段 file 表示启动文件名。

程序 dhcp_discover.c 实现了 DHCP 数据包的构造和发送。其核心代码如下。

```
#if (HAVE_CONFIG_H)
#include "../include/config.h"
#endif
#include "./libnet_test.h"
#ifdef __WIN32__
#include "../include/win32/getopt.h"
#endif
void
usage(char *prog)
{
   fprintf(stderr, "Usage: %s interface\n", prog);
   exit(1);
}
int
main(int argc, char *argv[])
{
     char *intf;
     u_long src_ip, options_len, orig_len;
     int i;
     u_int8_t ethaddr_1[6] = {0x01, 0x02, 0x03, 0x04, 0x05, 0x06};
     /*此处是另外添加的，原程序没有*/
          libnet_t *l;                          /*Libnet 句柄*/
     libnet_ptag_t t;
     libnet_ptag_t ip;                          /*IP 协议块标记*/
     libnet_ptag_t udp;                         /*UDP 协议块标记*/
     libnet_ptag_t dhcp;                        /*DHCP 协议块标记*/
          struct libnet_ether_addr *ethaddr;
          struct libnet_stats ls;               /*Libnet 状态分析*/
   char errbuf[LIBNET_ERRBUF_SIZE];
   u_char options_req[] = { LIBNET_DHCP_SUBNETMASK , LIBNET_DHCP_BROADCASTADDR ,
```

```
    LIBNET_DHCP_TIMEOFFSET , LIBNET_DHCP_ROUTER , LIBNET_DHCP_DOMAINNAME ,
    LIBNET_DHCP_DNS , LIBNET_DHCP_HOSTNAME };
    u_char *options;
    u_char enet_dst[6] = {0xff, 0xff, 0xff, 0xff, 0xff, 0xff};
    u_char *tmp;
    if (argc != 2)
       usage(argv[0]);
    intf = argv[1];
    l = libnet_init(                          /*初始化 Libnet*/
          LIBNET_LINK,                        /*Libnet 类型*/
          intf,                               /*网络设备接口*/
          errbuf);
     if (!l)
     {
       fprintf(stderr, "libnet_init: %s", errbuf);
       exit(EXIT_FAILURE);
     }
     else {
      rc_ip = libnet_get_ipaddr4(l);
      src_ip = inet_addr("192.168.0.1");        /*添加的*/
          /*
    if ((ethaddr = libnet_get_hwaddr(l)) == NULL)
       {
           fprintf(stderr, "libnet_get_hwaddr: %s\n", libnet_geterror(l));
           exit(EXIT_FAILURE);
       }
          */
          ethaddr=malloc(48);
          for(i=0;i<6;i++)
          {
          ethaddr->ether_addr_octet[i]=ethaddr_l[i];
          }
          /*另外添加的，对以太网地址进行赋值*/
printf("ip addr     : %s\n", libnet_addr2name4
       (src_ip, LIBNET_DONT_RESOLVE));
       printf("eth addr    : ");
       for (i = 0; i < 6; i++) {
          printf("%2.2x", ethaddr->ether_addr_octet[i]);
          if (i != 5) {
             printf(":");
          }
       }
       printf("\n");
```

```
/*下面一段程序用来构造选项部分内容*/
i = 0;
options_len = 3;                                /*选项长度*/
options = malloc(3);
options[i++] = LIBNET_DHCP_MESSAGETYPE;         /*类型*/
options[i++] = 1;                               /*长度*/
options[i++] = LIBNET_DHCP_MSGDISCOVER;         /*数据*/
orig_len = options_len;
options_len += sizeof(options_req) + 2;
tmp = malloc(options_len);
memcpy(tmp, options, orig_len);
free(options);
options = tmp;
options[i++] = LIBNET_DHCP_PARAMREQUEST;
options[i++] = sizeof(options_req);
memcpy(options + i, options_req, sizeof(options_req));
i += sizeof(options_req);
if (src_ip)
{
    orig_len = options_len;
    options_len += 2 + sizeof(src_ip);
    tmp = malloc(options_len);
    memcpy(tmp, options, orig_len);
    free(options);
    options = tmp;
    options[i++] = LIBNET_DHCP_DISCOVERADDR;
    options[i++] = sizeof(src_ip);
    memcpy(options + i, (char *)&src_ip, sizeof(src_ip));
    i += sizeof(src_ip);
}
orig_len = options_len;
options_len += 1;
tmp = malloc(options_len);
memcpy(tmp, options, orig_len);
free(options);
options = tmp;
options[i++] = LIBNET_DHCP_END;
if (options_len + LIBNET_DHCPV4_H < LIBNET_BOOTP_MIN_LEN)
{
    orig_len = options_len;
    options_len = LIBNET_BOOTP_MIN_LEN - LIBNET_DHCPV4_H;
    tmp = malloc(options_len);
    memcpy(tmp, options, orig_len);
```

```
            inet_addr("255.255.255.255"),                    /*目的 IP 地址*/
        NULL,                                                /*负载*/
        0,                                                   /*负载大小*/
        1,                                                   /*Libnet 句柄*/
        0                                                    /*Libnet 标记*/
        );
    t = libnet_autobuild_ethernet(
        enet_dst,                                            /*以太网目的地址*/
        ETHERTYPE_IP,                                        /*协议类型*/
        l);
    if (libnet_write(l) == -1)
    {
        fprintf(stderr, " %s: libnet_write: %s\n", argv[0],
         strerror(errno));
        exit(EXIT_FAILURE);
    }
    libnet_stats(l, &ls);
    fprintf(stderr, "Packets sent:  %lld\n"
          "Packet errors: %lld\n"
          "Bytes written: %lld\n",
          ls.packets_sent, ls.packet_errors, ls.bytes_written);
    libnet_destroy(l);
    free(options);
    exit(0);
    }
    exit(0);
}
```

编译并运行此程序，运行结果如图 5-73 所示。

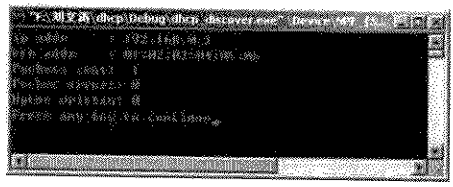

图 5-73　构造 DHCP 数据包的运行结果

利用 Analyzer 工具分析运行结果，其捕获到的数据包如图 5-74 所示。

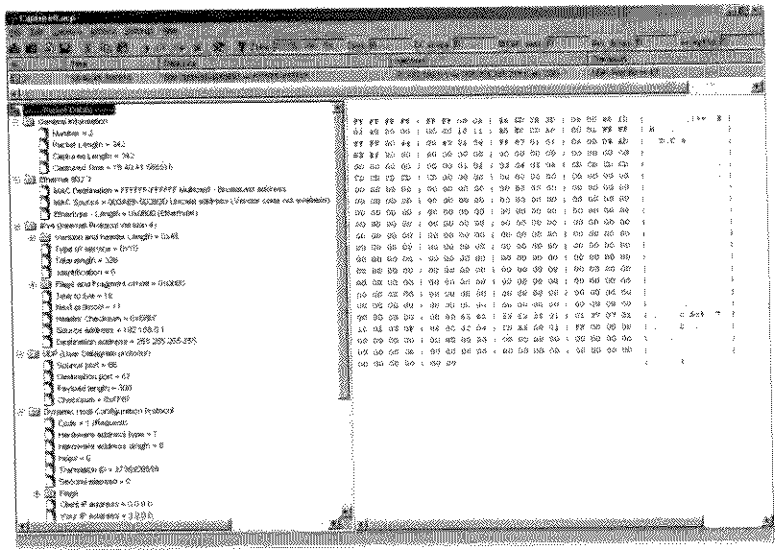

图 5-74　Analyzer 捕获 DHCP 数据包

其中，数据包的各个协议块的详细内容如下：

```
⊟ General information
    ⊟ ● Number                    = 1
    ⊟ ● Packet Length             = 92
    ⊟ ● Captured Length           = 92
    ⊟ ● Captured Time             = 18:40:38.614429
⊟ Ethernet 802.3
    ⊟ ● MAC Destination           = FFFFFF-FFFFFF Multicast - Broadcast address
    ⊟ ● MAC Source                = 000AE6-60A5AD Unicast address (Vendor code not
                                    available)
    ⊟ ● Ethertype - Length        = 0x0800 (Ethertype)
⊟ IPv4 (Internet Protocol version 4)
    ⊟ ● Version and header Length = 0x45
        ⊟ ● Version               = 4
        ⊟ ● Header length         = 20 (field value = 5)
    ⊟ ● Type of service           = 0x00
    ⊟ ● Total length              = 78
    ⊟ ● Identification            = 7940
    ⊟ ● Flags and Fragment offset = 0x0000
        ⊟ ● Unused                = 0b0...............
        ⊟ ● Don't fragment        = 0b.0..............
        ⊟ ● More fragments        = 0b..0.............
```

```
    ⊟ ● Fragment offset          = 0 (field value = 0)
  ⊟ ● Time to live               = 128
  ⊟ ● Next protocol              = 17
  ⊟ ● Header Checksum            = 0xF0C6
  ⊟ ● Source address             = 211.85.194.9
  ⊟ ● Destination address        = 211.85.194.31
⊟   UDP (User Datagram protocol)
  ⊟ ● Source port                = 137
  ⊟ ● Destination port           = 137
  ⊟ ● Payload length             = 58
  ⊟ ● Checksum                   = 0xCE76
⊟   NetBios Name Service
  ⊟ ● Transaction ID             = 33696
  ⊟ ● General Flag               =
    ⊟ ● Request (0) or Response (1)                  = 0b0...............
    ⊟ ● Opcode                                       = 0b.0000........... (Query)
    ⊟ ● Authoritative (0) or not authoritative (1)   = 0b.....0..........
    ⊟ ● Datagram not truncated (0) or truncated (1)  = 0b......0.........
    ⊟ ● Recursion desired (0) or not desired (1)     = 0b.......1........
    ⊟ ● Recursion not available (0) or available (1) = 0b........0.......
    ⊟ ● Reserved (must be zero)                      = 0b.........00.....
    ⊟ ● Unicast (0) or Broadcast (1)                 = 0b...........1....
    ⊟ ● RCode                                        = 0b...........0000 (Success)
  ⊟ ● Number of question entries                     = 1
  ⊟ ● Number of resource in the answer section       = 0
  ⊟ ● Number of resource in the authority section    = 0
  ⊟ ● Number of resource in the additional records   = 0
  ⊟ ● Question Entries                               =
    ⊟ ● Name Length                                  = 32
    ⊟ ● Question Name                                = WORKGROUP
    ⊟ ● Question Type                                = 32
    ⊟ ● Question Class                               = 1
```

5.6.8 构造多个数据包

如果在一个程序中想一次发送多个数据包，那么使用 Libnet 句柄队列就非常方便。Libnet 的作者提出 Libnet 句柄队列，就是为了实现多个数据包连发的功能。这里重点讨论一下 Libnet 句柄队列的使用方法，其具体步骤如下：

（1）构造一个数据包。

（2）向一个 Libnet 句柄队列中插入该数据包。如果 Libnet 句柄队列不存在，就马上创建一个队列，并把该数据包插入到队列的第一个结点；如果 Libnet 句柄已经存在，就把该数据包插入 Libnet 队列的第一个结点位置。也就是说，每次插入的结点都作为第一个结点。

（3）重复执行步骤（1）（2），直到插入所有的数据包位置。

（4）依次读取 Libnet 句柄队列中的每个数据包，然后发送出去。

构造后的 Libnet 句柄如图 5-75 所示。图 5-75 表示的是一个 Libnet 句柄队列，其中有 3 个 Libnet 句柄结点，每个 Libnet 句柄表示一个完整的数据包，注意，在这里不要把数据包和协议块弄混淆。数据包是一个完整的协议数据包，它是由几个协议块构成的，而协议块是构成一个数据包的组成部分。例如，一个 TCP 数据包是由 TCP 协议块、IP 协议块和以太网协议块构成的。

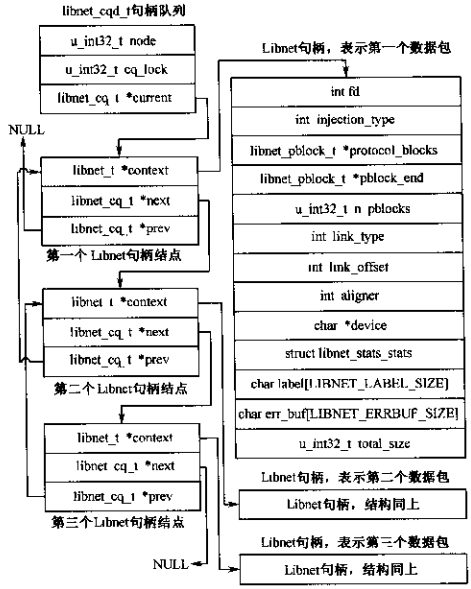

图 5-75　Libnet 句柄队列

注意，在上述 4 个步骤中，每次操作的对象都是一个数据包，但是在 Libnet 中是由一个 Libnet 句柄的描述符来惟一标识的。此描述符号其实就是一个字符串，用一个字符串来惟一标识一个构造的数据包，所以此字符串必须是惟一的，不能够重复。例如，如果在 Libnet 句柄队列中已经存在一个字符串描述的数据包，就不能在用此字符串标识另外一个数据包，即使是完全一样的数据包，也必须用另外一个字符串来标识。Libnet 句柄队列中不能有重复的 Libnet 句柄标识符。

在构造一个数据包之后，把它插入 Libnet 句柄队列中，插入的位置是第一个结点。所以可以看出，在发送时，从第一个结点开始发送，而这个结点是最后构造的。在图 5-75 中，第三个结点是首先构造的，第二个结点是第二构造的，第一个结点是最后构造的。从这里得出结论，发送数据包的顺序跟构造数据包的顺序刚好相反。

下面给出构造 ICMP 回显数据包的源代码：（为使说明问题变得更简单，这里没有考虑错误处理问题和参数化问题。读者可以参考 Libnet 开发包中的程序 icmp_echo.c，其中有详细的错误处理和参数化处理。）

```
#include <win32/libnet.h>
void main()
{
    libnet_t *l = NULL;                                  /*Libnet 句柄*/
    libnet_ptag_t protocol_tag;                          /*协议标记*/
    char *payload = NULL;                                /*负载*/
    u_short payload_length = 0;                          /*负载长度*/
    char *device ="\\Device\\NPF_{5973DFB0-AD5B-4065-A22B-C61FFB64624F}";
    /*网络设备接口，在这里用的是 windows 下的一个接口，根据各种机器的类型而定，可以通过很多方法获得，例如
    使用 windump 或 ngrep 程序来获得即可*/
    char *destination_ip_str = "192.168.0.3";            /*目的 IP 地址字符串*/
          char *source_ip_str ="192.168.0.2";            /*源 IP 地址字符串*/
     u_long source_ip = 0;                               /*源 IP 地址*/
          u_long destination_ip = 0;                     /*目的 IP 地址*/
    char errbuf[LIBNET_ERRBUF_SIZE];                     /*错误信息*/
    char label[LIBNET_LABEL_SIZE];                       /*Libnet 句柄字符串标记*/
          int j=0;                                       /*用于标记第几个数据包*/
          int i=0;                                       /*用于循环*/
          int packet_length;                             /*发送的数据包的长度*/
     int number = 5;                                     /*发送数据包的个数*/
    for (i = 0; i < number; i++)
    /*循环构造 5 个数据包，每个数据包的 Libnet 句柄字符串标记不同，也就是说用惟一的一个 Libnet 句柄字符串
    标记来指示一个构造的数据包*/
          {
          l = libnet_init(                               /*初始化 Libnet*/
               LIBNET_RAW4,                              /*Libnet 类型，为原始套接字 IPv4 类型*/
               device,                                   /*网络设备接口*/
               errbuf                                    /* 错误信息*/
                    );
               source_ip=libnet_name2addr4(l,source_ip_str,LIBNET_RESOLVE);
               /*把源 IP 地址字符串形式转化为网络字节顺序的数据*/
               destination_ip=libnet_name2addr4(l,destination_ip_str,LIBNET_RESOLVE);
               /*把目的 IP 地址字符串形式转化为网络字节顺序的数据*/
          protocol_tag = libnet_build_icmpv4_echo(       /*构造 ICMP 回显数据包*/
               ICMP_ECHO,                                /*类型，此时为回显请求*/
               0,                                        /*代码，应该为 0*/
               0,                                        /*校验和，为 0，表示由 Libnet 句柄自动计算*/
               123,                                      /*标识符，赋值为 123，自己任意填写数值*/
               456,                                      /*序列号，赋值为 245，自己任意填写数值*/
               NULL,                                     /*负载，赋值为空*/
               0,                                        /*负载的长度，赋值为 0*/
               l,                                        /*Libnet 句柄，应该是由 libnet_init()函数
                                                           得到的*/
```

```
            0                                   /*协议块标记，赋值为 0，表示构造一个新的协议块*/
            );
        protocol_tag = libnet_build_ipv4(       /*构造 IP 协议块*/
            LIBNET_IPV4_H + LIBNET_ICMPV4_ECHO_H + payload_length,
            /*IP 协议块的长度*/
            0,                                  /*服务质量，这里赋值为 0*/
            10,                                 /*标识符，这里赋值为 10*/
            0,                                  /*偏移，这里赋值为 0*/
            20,                                 /*生存时间，这里赋值为 20*/
            IPPROTO_ICMP,                       /*上层协议类型，这里是 ICMP 协议*/
            0,                                  /*校验和，这里为 0 表示由 Libnet 计算校验和*/
            source_ip,                          /*源 IP 地址*/
            destination_ip,                     /*目的 IP 地址*/
            payload,                            /*负载*/
            payload_length,                     /*负载的长度*/
            l,                                  /*Libnet 句柄*/
            0                                   /*协议块标记，为 0 表示构造一个新的 IP 协议块*/
            );
        snprintf(label, sizeof(label)-1, "test %d", i);
            /*构造一个 Libnet 句柄字符串标识，用来指示一个 Libnet 句柄*/
            libnet_cq_add(l,label);
            /*插入一个 Libnet 句柄结点到 Libnet 句柄队列中，并且是作为第一个结点，此句柄结点由 label
            字符串来惟一标识*/
    }
    for (l = libnet_cq_head(); libnet_cq_last(); l = libnet_cq_next())
    /*循环遍历 Libnet 句柄队列中所有的 Libnet 句柄，然后发送*/
        {
            j++;                                /*第几个 Libnet 句柄*/
        packet_length = libnet_write(l);        /*发送由 Libnet 句柄 l 表示的数据包*/
        printf("发送第 %d 个%d 字节长度的 ICMP 回显请求数据包，其对应的 Libnet 句柄为
        \"%s\"\n",j,packet_length,libnet_cq_getlabel(l));        /*输出发送的数据包信息*/
    }
    libnet_cq_destroy();                        /*销毁 Libnet 句柄队列，
    同时销毁所有在 Libnet 句柄队列中的每个 Libnet 句柄*/
}
```

编译并运行此程序，其运行结果如图 5-76 所示。

图 5-76　使用 Libnet 句柄队列的运行结果

使用 Analyzer 捕获数据包，其运行结果如图 5-77 所示。

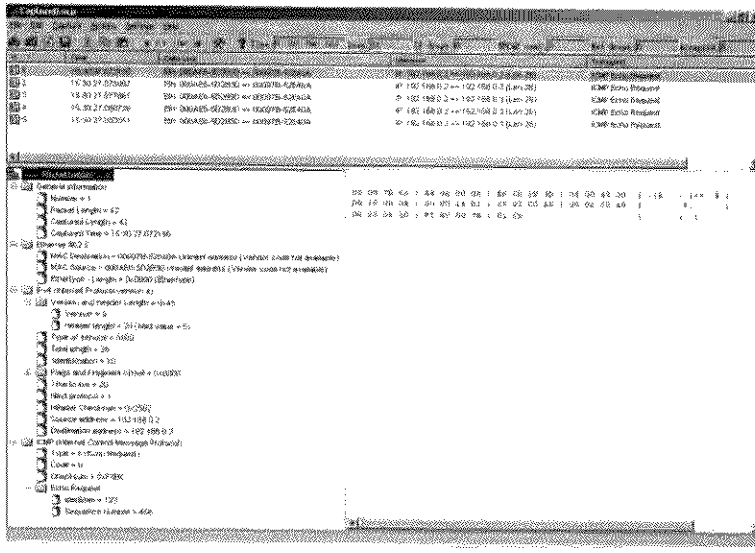

图 5-77　Analyzer 捕获 Libnet 队列程序的运行结果

可以从捕获结果分析出，共构造了 5 个数据包，其构造的顺序和发送的顺序正好相反，因为每构造一个数据包就把它插入 Libnet 句柄队列的首个结点处，而发送的过程是从第一个结点开始发送的。

5.7　小结

本章详细阐述了网络数据包构造和发送开发包 Libnet，对 Libnet 的组成部分和安装过程进行了介绍，对 Libnet 的数据结构和输出函数进行了详细分析，对 Libnet 的使用过程通过丰富的例程进行了深入讨论。

Libnet 开发包的核心功能是构造和发送各种协议的网络数据包，使构造复杂的网络数据包变得非常简单。Libnet 支持丰富的网络协议，用它构造的网络数据包种类极其丰富，本章通过举例构造了常用的几种网络协议数据。

Libnet 是用于网络数据包构造和发送的专业开发包，它可以应用在各种网络安全应用系统之中，如网络安全扫描系统、防火墙系统、入侵检测系统、网络测试系统等。如果 Libnet 被攻击者所利用，就会设计出功能强大的攻击系统，所以说 Libnet 是一把双刃剑。只要系统中需要构造和发送数据包的功能，都可以利用 Libnet 来实现。

第 6 章　通用网络安全开发包 Libdnet

6.1　Libdnet 概述

6.1.1　Libdnet 简介

Libdnet 是一个简单、平台兼容、多功能的底层网络开发包，是一个通用的网络安全开发包，它提供的很多功能在多数网络安全系统中都可以用到。

在实际的网络安全程序设计中，经常要碰到一些基本而有用的操作，如地址的管理、地址格式的转换等。这些操作在网络安全程序设计中使用非常频繁，为了使这些操作的实现变得更加容易，人们开发了 Libdnet 开发包。

Libdnet 的功能非常多，并不只有简单的地址操作，还有很多非常有用的功能，包括操作网络接口。例如，在实际的网络安全设计中，我们要操作网络接口，要对网络接口进行设置管理。例如，修改网络接口的参数，使它具备某一功能或者不具备某一功能。

Libdnet 还实现了对 ARP 缓存和路由表的管理功能。有了这些功能，我们在进行程序设计时，如果要对 ARP 缓存进行配置，就可以使用 Libdnet 来非常方便地实现。

Libdnet 实现了对路由表的管理和配置，它可以查看路由表信息，删除路由表，修改路由表，并增加路由表信息。要实现这些操作，使用 Libdnet 中的函数就会变得非常简单。

防火墙的配置和管理是网络安全程序设计中的一个重点。现在的防火墙系统非常多，如在 Linux 下比较著名的 ipchains，ipfw，pf 等系统，以及 Windows 平台下的 PktFilter，Libdnet 实现了对它们的操作和配置，并进行了封装，使用起来非常简单。使用 Libdnet 可以查看、修改、删除、增加防火墙规则，而且这些操作对各种支持的防火墙都是一致的，其函数的调用方式也都是一致的，因为在 Libdnet 中，把它们的实现都进行了统一封装，屏蔽了各种防火墙的不同之处。

在 Libdnet 中，还有一个重要的功能是构造和发送网络数据包。构造数据包的方式主要有两类，一是基于原始 IP 包，一是基于以太网帧类型。基于原始 IP 包的方式可以发送所有基于 IP 协议的数据包，如 ICMP，TCP，UDP 等；基于以太网帧类型的方式可以发送各种基于以太网的数据包，如 ARP，IP 等。

Libdnet 支持的开发语言有 C，C++，Python，Perl；支持的平台有 BSD（包括 OpenBSD，FreeBSD，NetBSD，BSD/OS），Linux（包括 Redhat，Debian，Slackware 等），Solaris，MacOS X，Windows NT/2000/XP，IRIX 和 HP-UX。本书主要讨论 Linux 平台和 Windows 平台，其中 Windows 平台以 Windows 2000 为主。

使用 Libdnet 开发的网络程序很多，如 arpscan，dhcp-agent，Firewalk，Fragroute，Ganglia，Honeyd，RING 等，还有很多系统都用到了 Libdnet 开发包。Libdnet 是一个应用非常广泛的通用网络开发包，特别在网络安全方面应用最多。

Libdnet 是由 Dug Song 开发的，它是一个开放源代码软件，可以从网站上直接下载，其下载地址为：http://prdownloads.sourceforge.net/libdnet/libdnet-1.10.tar.gz。

Libdnet 的最新版本为 1.10，本书以此版本进行讨论。

6.1.2 Libdnet 的主要功能

使用 Libdnet 可以开发很多有用的网络安全程序。Libdnet 的功能非常丰富，它主要包括以下几个方面。

1. 网络地址操作

在编写网络程序时，经常碰到网络地址在不同形式下的转换问题。Libdnet 实现了其中大部分功能，这样就使网络开发程序变得很轻松。网络地址有 IPv4 地址、IPv6 地址和以太网协议地址。

2. ARP 缓存和路由表查询和管理

在 Linux 内核中，有 arp 缓存和路由表的功能，而 Libdnet 把对这些功能的管理进行了封装。如果程序需要对它们进行设置，就可以直接调用 Libdnet 函数来实现，其中包括很多操作，如查选、修改、添加、删除等。

3. 网络防火墙

在 Libdnet 中实现了对某些网络防火墙的管理功能，其中包括对 ipfw，ipchains，pf，PktFilter 等防火墙系统的管理。主要操作包括对防火墙规则的查看、修改、添加、删除等。如果要对这些防火墙进行操作，则使用 Libdnet 开发包对它们进行管理非常容易。

4. 网络接口管理

使用 Libdnet 可以查看本机的网络接口，并且对其参数进行修改和设置。在 Libdnet 中有专门的功能来实现该功能，使用户对本机网络接口的管理更加方便。因为在一般的网络程序中都要对网络接口进行干预，如网络协议分析程序、网络防火墙等，所以 Libdnet 就把对网络接口的管理进行了封装。

5. IP 隧道

在 Libdnet 中实现了 IP 隧道功能。如果想利用 IP 隧道功能，就可以借用 Libdnet 来进行构造和配置。

6. 原始 IP 包和以太网帧的构造和发送

利用 Libdnet 可以轻松地构造原始 IP 数据包和以太网帧。这样就可以轻松地构造基于 IP 协议的任何其他协议，如 TCP，UDP，ICMP 等。Libdnet 比 Libnet 更加快捷，而使用的步骤更少。

6.1.3 Libdnet 的安装

Libdnet 的安装过程与 Libpcap 和 Libnet 开发包的安装过程大同小异。

下面介绍在 Linux 下的安装过程，其步骤如下。

（1）对 Libdnet 安装包解压缩，使用命令：

```
tar -xzvf  libdnet-1.10.tar.gz
```

其界面如图 6-1 所示。

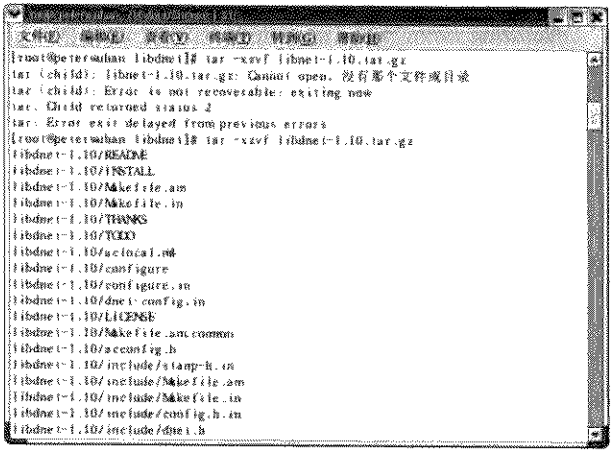

图 6-1　解压缩 Libdnet 安装包

（2）进入 Libdnet 目录，可运行配置命令：

```
./configure
```

配置界面如图 6-2 所示。

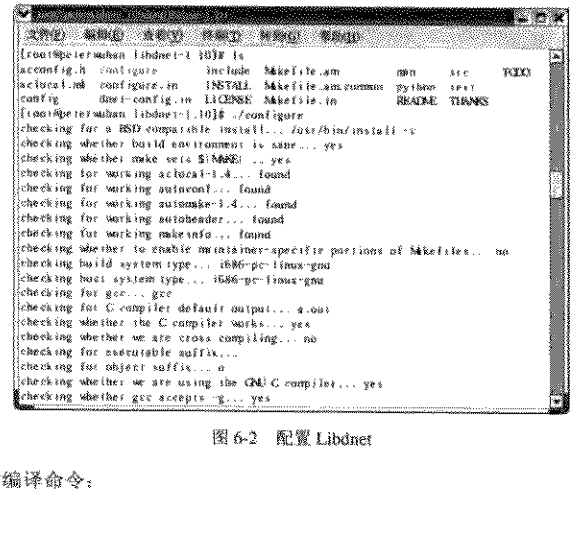

图 6-2　配置 Libdnet

（3）运行编译命令：

```
make
```

编译界面如图 6-3 所示。

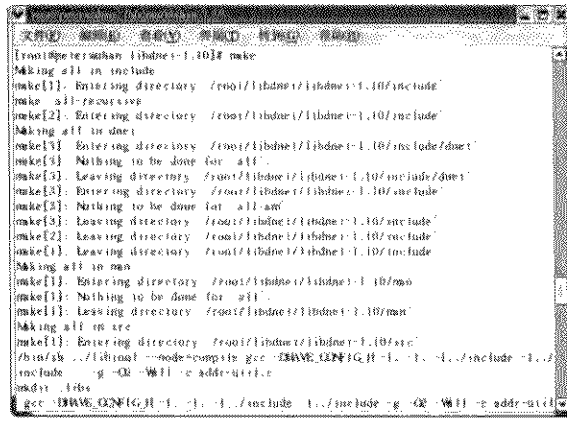

图 6-3　编译 Libdnet

（4）执行安装命令：

```
make install
```

安装界面如图 6-4 所示。

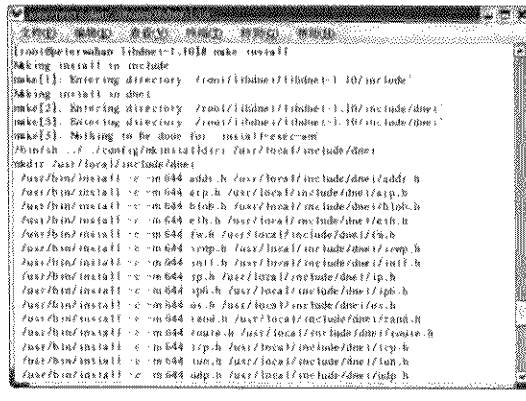

图 6-4　安装 Libdnet

至此，Libdnet 已经安装成功了。

安装完之后，Libdnet 本身自带了一个测试程序，其命令是 dnet，它带有很多参数。具体信息可以使用 man 命令进行查看：

```
man dnet
```

例如，执行如下命令：

```
dnet hex "\x12\x34\x55" |dnet icmp type 8 code 0 |dnet ip proto icmp src 192.168.0.3 dst
192.168.0.3 |dnet send
```

表示使用 dnet 构造一个 ICMP 数据包，然后发送出去，如图 6-5 所示。

图 6-5　使用 dnet 命令

使用 Tcpdump 工具进行查看，可以捕获到刚才发送的 ICMP 数据包，如图 6-6 所示。

图 6-6　捕获 ICMP 数据包

在 Windows 平台下，Libdnet 有一个 Windows 平台下的开发包，下载后其名称为 Libdnet-1.10-win32.zip。解压缩此文件即可，它里面有两个子文件夹：Include 和 Lib，分别表示 Libdnet 的头文件和静态链接库，在程序开发时都要用到。

在 Windows 平台下，可以使用 Visual C++ 6.0 设计基于 Libdnet 的程序。

6.2　Libdnet 的组成

Libdnet 开发包实现了很多有用的功能，这些功能在 Libdnet 中都具有模块化的性质。几乎每个功能都用一个源程序来实现，不同功能之间没有太多的交叉，这对于运用 Libdnet 开发包是非常方便的。

1．网络地址管理

在 Libdnet 中，实现了对网络地址之间的转换。例如，它可以把一个 IP 地址的点制文本形式转化成字节流形式，也可以把一个字节流形式的 IP 地址转化为一个点制的文本形式。

详细内容见 6.4.1 节。

2．原始 IP 包和以太网帧的构造和发送

在 Libdnet 中可以构造任意内容的网络数据包。其中构造的数据包种类有两大类，原始套接字 IP 包类和以太网帧类。

原始 IP 包类数据包从 IP 层进行构造，Libdnet 可以构造任意基于 IP 层协议的网络协议，如构造 TCP，UDP，ICMP 等协议。

以太网帧类数据包从链路层进行构造，Libdnet 可以构造基于以太网协议的数据包，如 IP 数据包，ARP 数据包等。

详细内容见 6.4.2 和 6.4.3 节。

3．内核 ARP 缓存和路由表管理

在 Libdnet 中实现了 ARP 缓存和路由表的管理。它可以显示、查询，增加和删除 ARP 缓存记录，也可以显示、查询、增加和删除路由表记录。

详细内容见 6.4.6 和 6.4.8 节。

4. 网络接口管理

Libdnet 也实现对网络接口的管理功能。

在 Libdnet 中可以查看网络接口信息，并修改网络接口的参数，如它的地址。Libdnet 可以激活网络接口，也可以使网络接口变成不活跃状态，还可以设置网络接口是否具有 ARP 协议功能，等等。

详细内容见 6.4.7 节。

5. 网络防火墙

对网络防火墙的管理是 Libdnet 比较新的功能，它可以管理几种类型的防火墙系统，如最热门的 ipchains 系统，以及 ipfw，ipchains，pf，PktFilter 等系统。

它可以显示防火墙的规则，也可以添加、修改、删除规则。

详细内容见 6.4.10 节。

6.3 Libdnet 数据结构

在 Libdnet 中定义了各种各样的数据结构，这些数据结构在具体的功能实现过程中都要用到，它们全部定义在头文件中。在 Libdnet 中有很多头文件，它们是：

- addr.h 定义关于地址操作的数据结构和函数声明；
- arp.h 定义关于 ARP 操作的数据结构和函数声明；
- blob.h 定义关于数据块的数据结构和函数声明；
- eth.h 定义以太网协议的数据结构和函数声明；
- fw.h 定义关于防火墙操作的数据结构和函数声明；
- icmp.h 定义 ICMP 协议的数据结构和函数声明；
- intf.h 定义网络接口的数据结构和函数声明；
- ip.h 定义 IPv4 协议的数据结构和函数声明；
- ip6.b 定义 IPv6 协议的数据结构和函数声明；
- os.h 定义与操作系统相关的一些设置；
- rand.h 定义随机数操作的相关数结构和函数声明；
- route.h 定义路由表操作的数据结构和函数声明；
- tcp.h 定义 TCP 协议的数据结构和函数声明；
- tun.h 定义 IP 隧道的数据结构和相关函数声明；
- udp.h 定义 UDP 协议的数据结构和相关的函数声明。

6.3.1 网络地址结构

网络地址结构的定义在文件 addr.h 中。

```
#define  ADDR_TYPE_NONE     0
#define  ADDR_TYPE_ETH      1    /*以太网地址*/
#define  ADDR_TYPE_IP       2    /*IPv4 地址*/
#define  ADDR_TYPE_IP6      3    /*IPv6 地址*/
```

下面是在 Libdnet 中的网络地址的数据结构的定义：

```
struct addr {
    uint16_t      addr_type;     /*地址类型*/
    uint16_t      addr_bits;     /*地址位数*/
    union {
        eth_addr_t    __eth;     /*以太网地址*/
        ip_addr_t __ip;          /*IPv4 地址*/
        ip6_addr_t    __ip6;     /*IPv6 地址*/
        uint8_t   __data8[16];
        uint16_t  __data16[8];
        uint32_t  __data32[4];
    } __addr_u;
};
#define addr_eth   __addr_u.__eth
#define addr_ip        __addr_u.__ip
#define addr_ip6   __addr_u.__ip6
#define addr_data8 __addr_u.__data8
#define addr_data16    __addr_u.__data16
#define addr_data32    __addr_u.__data32
#define addr_pack(addr, type, bits, data, len) do { \
    (addr)->addr_type = type;              \
    (addr)->addr_bits = bits;              \
    memmove((addr)->addr_data8, (char *)data, len);     \
} while (0)
```

在 Libdnet 中定义了三种类型的网络地址，分别是 IPv4 地址、以太网协议地址和 IPv6 地址。它可以对此三种类型的网络地址进行操作，并且操作的过程都是一样的，Libdnet 把对它们的操作用统一的函数来实现，而不是分别用三种类型的函数来实现。在实现的函数中对此三种类型的网络地址操作进行了封装，隐藏了它们的不同之处，将一个统一的接口提供给用户。

6.3.2 协议结构

在 Libdnet 中定义了一些重要的协议结构，它们在头文件 arp.h，eth.h，icmp.h，ip.h，ip6.h，tcp.h 和 udp.h 中定义。这些协议结构主要在构造和发送它们时要用到，在 Libdnet 中对这些协议的格式和各种选项进行了详细的定义，下面分别进行详细的阐述。

1. arp 协议结构

```
#define ARP_HDR_LEN     8
#define ARP_ETHIP_LEN   20
```

ARP 协议头的数据结构定义如下：

```
struct arp_hdr {
    uint16_t      ar_hrd;                /*硬件地址类型*/
    uint16_t      ar_pro;                /*协议地址类型*/
```

```
    uint8_t        ar_hln;                    /*硬件地址长度*/
    uint8_t        ar_pln;                    /*协议地址长度*/
    uint16_t       ar_op;                     /*操作码*/
};
#define ARP_HRD_ETH     0x0001                /*以太网硬件地址类型*/
#define ARP_HRD_IEEE802 0x0006                /*IEEE 802 硬件地址类型*/
#define ARP_PRO_IP 0x0800                     /*IP 协议地址类型*/
#define   ARP_OP_REQUEST          1           /* ARP 请求*/
#define   ARP_OP_REPLY            2           /*ARP 应答*/
#define   ARP_OP_REVREQUEST       3           /*RARP 请求*/
#define   ARP_OP_REVREPLY         4           /*RARP 应答*/
```

ARP 请求协议的数据结构定义如下：

```
struct arp_ethip {
    uint8_t        ar_sha[ETH_ADDR_LEN];      /*源以太网地址*/
    uint8_t        ar_spa[IP_ADDR_LEN];       /*源 IP 协议地址*/
    uint8_t        ar_tha[ETH_ADDR_LEN];      /*目的以太网地址*/
    uint8_t        ar_tpa[IP_ADDR_LEN];       /*目的 IP 地址*/
};
```

2. 以太网协议结构

```
#define ETH_ADDR_LEN        6                 /*以太网地址长度为 6 字节*/
#define ETH_ADDR_BITS       48                /*以太网地址长度为 48 位*/
#define ETH_TYPE_LEN        2                 /*以太网类型为 2 字节*/
#define ETH_CRC_LEN         4                 /*以太网校验和的长度为 4 字节*/
#define ETH_HDR_LEN         14                /*以太网协议头长度为 14 字节*/
#define ETH_LEN_MIN         64                /*最小以太网协议长度为 64 字节*/
#define ETH_LEN_MAX         1518              /*最大以太网协议长度为 1518 字节*/
#define ETH_MTU        (ETH_LEN_MAX - ETH_HDR_LEN - ETH_CRC_LEN)
#define ETH_MIN        (ETH_LEN_MIN - ETH_HDR_LEN - ETH_CRC_LEN)
```

以太网协议地址定义如下：

```
typedef struct eth_addr {
uint8_t        data[ETH_ADDR_LEN];            /*表示以太网协议地址*/
} eth_addr_t;
```

以太网协议头的数据结构定义如下：

```
struct eth_hdr {
eth_addr_t    eth_dst;                        /*目的以太网地址*/
    eth_addr_t    eth_src;                    /*源以太网地址*/
    uint16_t  eth_type;                       /*以太网类型*/
};
```

以太网类型字段定义的是上层协议类型，它主要有以下类型：

```
#define ETH_TYPE_PUP            0x0200    /*表示 PUP 协议类型*/
#define ETH_TYPE_IP             0x0800    /*表示 IP 协议类型*/
```

```
#define ETH_TYPE_ARP              0x0806    /*表示 ARP 协议类型*/
#define ETH_TYPE_REVARP           0x8035    /*表示 RARP 协议类型*/
#define ETH_TYPE_8021Q            0x8100    /*表示 IEEE 802.1Q VLAN 类型*/
#define ETH_TYPE_IPV6             0x86DD    /*表示 IPv6 协议类型*/
#define ETH_TYPE_MPLS             0x8847    /*表示 MPLS 协议类型*/
#define ETH_TYPE_MPLS_MCAST       0x8848    /*表示 MPLS 多点传送类型*/
#define ETH_TYPE_PPPOEDISC        0x8863    /*表示 PPP 发现类型*/
#define ETH_TYPE_PPPOE            0x8864    /*表示 PPP 会话类型*/
#define ETH_TYPE_LOOPBACK         0x9000    /*表示回路类型*/
#define ETH_IS_MULTICAST(ea)(*(ea) & 0x01)
#define ETH_ADDR_BROADCAST  "\xff\xff\xff\xff\xff\xff"
#define eth_pack_hdr(h, dst, src, type) do {              \
struct eth_hdr *eth_pack_p = (struct eth_hdr *)(h); \
memmove(&eth_pack_p->eth_dst, &(dst), ETH_ADDR_LEN);     \
memmove(&eth_pack_p->eth_src, &(src), ETH_ADDR_LEN);     \
eth_pack_p->eth_type = htons(type);              \
} while (0)
typedef struct eth_handle eth_t;
```

3. IP 协议结构

```
#define IP_ADDR_LEN          4                /*表示 IP 地址长度为 4 字节*/
#define IP_ADDR_BITS         32               /*表示 IP 地址长度为 32 位*/
#define IP_HDR_LEN           20               /*表示 IP 协议头为 20 字节*/
#define IP_OPT_LEN           2                /*表示基本的 IP 选项长度为 2 字节*/
#define IP_OPT_LEN_MAX       40               /*表示 IP 选项最大长度为 40 字节*/
#define IP_HDR_LEN_MAX       (IP_HDR_LEN + IP_OPT_LEN_MAX)
#define IP_LEN_MAX           65535            /*表示 IP 协议头的最大长度为 65535 字节*/
#define IP_LEN_MIN           IP_HDR_LEN
typedef uint32_t             ip_addr_t;
```

下面是 IP 协议头的数据结构的定义：

```
struct ip_hdr {
#if DNET_BYTESEX == DNET_BIG_ENDIAN
uint8_t        ip_v:4,                        /*IP 协议版本号*/
               ip_hl:4;                       /*IP 协议头长度*/
#elif DNET_BYTESEX == DNET_LIL_ENDIAN
uint8_t        ip_hl:4,                       /*IP 协议头长度*/
               ip_v:4;                        /*IP 协议版本号*/
#else
# error "need to include <dnet.h>"
#endif
uint8_t        ip_tos;                        /*服务质量参数*/
uint16_t  ip_len;                             /*总长度*/
uint16_t  ip_id;                              /*IP 标识*/
```

```
uint16_t  ip_off;                          /*偏移量 */
uint8_t        ip_ttl;                     /*生存时间*/
uint8_t        ip_p;                       /*协议类型*/
uint16_t  ip_sum;                          /*校验和*/
ip_addr_t ip_src;                          /*源地址*/
ip_addr_t ip_dst;                          /*目的地址*/
};
```

下面是常用的服务质量 TOS 的定义：

```
#define IP_TOS_DEFAULT       0x00          /*表示默认服务质量*/
#define IP_TOS_LOWDELAY      0x10          /*表示最小延迟*/
#define IP_TOS_THROUGHPUT    0x08          /*表示最大吞吐量*/
#define IP_TOS_RELIABILITY   0x04          /*表示最高可靠性*/
#define IP_TOS_LOWCOST       0x02          /*表示最小费用*/
#define IP_TOS_ECT           0x02
#define IP_TOS_CE                          0x01
#define IP_TOS_PREC_ROUTINE                0x00
#define IP_TOS_PREC_PRIORITY               0x20
#define IP_TOS_PREC_IMMEDIATE              0x40
#define IP_TOS_PREC_FLASH                  0x60
#define IP_TOS_PREC_FLASHOVERRIDE          0x80
#define IP_TOS_PREC_CRITIC_ECP             0xa0
#define IP_TOS_PREC_INTERNETCONTROL   0xc0
#define IP_TOS_PREC_NETCONTROL             0xe0
#define IP_RF       0x8000
#define IP_DF       0x4000
#define IP_MF       0x2000
#define IP_OFFMASK 0x1fff
#define IP_TTL_DEFAULT  64                 /*表示默认的 TTL*/
#define IP_TTL_MAX 255                     /*表示最大的 TTL*/
```

下面是 IP 上层协议类型的一些定义：

```
#define   IP_PROTO_IP        0             /*表示虚拟 IP*/
#define IP_PROTO_HOPOPTS     IP_PROTO_IP   /*表示 IPv6 hop-by-hop 选项*/
#define   IP_PROTO_ICMP      1             /*表示 ICMP 协议*/
#define   IP_PROTO_IGMP      2             /*表示 IGMP 协议*/
#define IP_PROTO_GGP         3             /*表示 GGP 协议*/
#define   IP_PROTO_IPIP      4             /*表示 IP 协议*/
#define IP_PROTO_ST          5             /*表示 ST 数据包模式*/
#define   IP_PROTO_TCP       6             /*表示 TCP 协议*/
#define IP_PROTO_CBT         7             /*表示 CBT*/
#define   IP_PROTO_EGP       8             /*表示 EGP 协议*/
#define IP_PROTO_IGP         9             /*表示 IGP 协议*/
#define IP_PROTO_BBNRCC      10            /*表示 BBN RCC 监视*/
#define IP_PROTO_NVP         11            /*表示 NVP 协议*/
```

```
#define  IP_PROTO_PUP        12        /*表示 PARC 通用数据包*/
#define IP_PROTO_ARGUS       13        /*表示 ARGUS*/
#define IP_PROTO_EMCON       14        /*表示 EMCON*/
#define IP_PROTO_XNET        15        /*表示 XNET*/
#define IP_PROTO_CHAOS       16        /*表示 CHAOS*/
#define  IP_PROTO_UDP        17        /*表示 UDP 协议*/
#define IP_PROTO_MUX         18        /*表示 MUX*/
#define IP_PROTO_DCNMEAS     19        /*表示 DCN 度量*/
#define IP_PROTO_HMP         20        /*表示 HMP 协议*/
#define IP_PROTO_PRM         21        /*表示 PRM*/
#define  IP_PROTO_IDP        22        /*表示 IDP 协议*/
#define IP_PROTO_TRUNK1      23        /*表示 Trunk-1*/
#define IP_PROTO_TRUNK2      24        /*表示 Trunk-2*/
#define IP_PROTO_LEAF1       25        /*表示 Leaf-1*/
#define IP_PROTO_LEAF2       26        /*表示 Leaf-2*/
#define IP_PROTO_RDP         27        /*表示 RDP 协议*/
#define IP_PROTO_IRTP        28        /*表示 IRTP*/
#define  IP_PROTO_TP         29        /*表示 TP*/
#define IP_PROTO_NETBLT      30        /*表示 NETBLT*/
#define IP_PROTO_MFPNSP      31        /*表示 MFE 网络服务*/
#define IP_PROTO_MERITINP    32        /*表示 MERITINP*/
#define IP_PROTO_SEP         33        /*表示 SEP 协议*/
#define IP_PROTO_3PC         34        /*表示第 3 方连接协议*/
#define IP_PROTO_IDPR        35        /*表示 IDPR*/
#define IP_PROTO_XTP         36        /*表示 XTP 协议*/
#define IP_PROTO_DDP         37        /*表示 DDP 协议*/
#define IP_PROTO_CMTP        38        /*表示 CMTP*/
#define IP_PROTO_TPPP        39        /*表示 TPPP 协议*/
#define IP_PROTO_IL          40        /*表示 IL 传输协议*/
#define IP_PROTO_IPV6        41        /*表示 IPv6 协议*/
#define IP_PROTO_SDRP        42        /*表示 SDRP 协议*/
#define IP_PROTO_ROUTING     43        /*表示 IPv6 路由头*/
#define IP_PROTO_FRAGMENT    44        /*表示 IPv6 碎片头*/
#define IP_PROTO_RSVP        46        /*表示 RSVP 协议*/
#define  IP_PROTO_GRE        47        /*表示 GRE 协议*/
#define IP_PROTO_MHRP        48        /*表示 MHRP 协议*/
#define IP_PROTO_ENA         49        /*表示 ENA*/
#define  IP_PROTO_ESP        50        /*表示 ESP*/
#define  IP_PROTO_AH         51        /*表示 AH*/
#define IP_PROTO_INLSP       52        /*表示 INLSP*/
#define IP_PROTO_SWIPE       53        /*表示 SWIPE*/
#define IP_PROTO_NARP        54        /*表示 NARP*/
#define  IP_PROTO_MOBILE     55        /*表示移动 IP*/
```

```
#define IP_PROTO_TLSP        56        /*表示传输层安全协议*/
#define IP_PROTO_SKIP        57        /*表示 SKIP*/
#define IP_PROTO_ICMPV6      58        /*表示 ICMPv6 协议*/
#define IP_PROTO_NONE        59        /*表示 IPv6 无下一个协议头*/
#define IP_PROTO_DSTOPTS     60        /*表示 IPv6 目的选项*/
#define IP_PROTO_ANYHOST     61        /*表示任何主机内部协议*/
#define IP_PROTO_CFTP        62        /*表示 CFTP*/
#define IP_PROTO_ANYNET      63        /*表示任何局域网*/
#define IP_PROTO_EXPAK       64        /*表示 SATNET 和 EXPAK*/
#define IP_PROTO_KRYPTOLAN   65        /*表示/Kryptolan*/
#define IP_PROTO_KVD         66        /*表示 RVD*/
#define IP_PROTO_IPPC        67        /*表示 IPPC*/
#define IP_PROTO_DISTFS      68        /*表示 DISTFS*/
#define IP_PROTO_SATMON      69        /*表示 SATNET 监视*/
#define IP_PROTO_VISA        70        /*表示 VISA 协议*/
#define IP_PROTO_IPCV        71        /*表示 IPCV*/
#define IP_PROTO_CPNX        72        /*表示 CPNX*/
#define IP_PROTO_CPHB        73        /*表示 CPHB*/
#define IP_PROTO_WSN         74        /*表示 WSN*/
#define IP_PROTO_PVP         75        /*表示数据包视频协议 PVP*/
#define IP_PROTO_BRSATMON    76        /*表示 BRSATMON*/
#define IP_PROTO_SUNND       77        /*表示 SUNND 协议*/
#define IP_PROTO_WBMON       78        /*表示 WIDEBAND 监视*/
#define IP_PROTO_WBEXPAK     79        /*表示 WIDEBAND EXPAK*/
#define  IP_PROTO_EON        80        /*表示 ISO CNLP*/
#define IP_PROTO_VMTP        81        /*表示 VMTP*/
#define IP_PROTO_SVMTP       82        /*表示安全 VMTP*/
#define IP_PROTO_VINES       83        /*表示 VINES*/
#define IP_PROTO_TTP         84        /*表示 TTP*/
#define IP_PROTO_NSFIGP      85        /*表示 NSFNET-IGP*/
#define IP_PROTO_DGP         86        /*表示 DGP 协议*/
#define IP_PROTO_TCF         87        /*表示 TCF*/
#define IP_PROTO_EIGRP       88        /*表示 EIGRP*/
#define IP_PROTO_OSPF        89        /*表示开放最短路径优先协议 OSPF*/
#define IP_PROTO_SPRITERPC   90        /*表示 RPC 协议*/
#define IP_PROTO_LARP        91        /*表示 LARP 协议*/
#define IP_PROTO_MTP         92        /*表示 MTP 协议*/
#define IP_PROTO_AX25        93        /*表示 AX 25 帧*/
#define IP_PROTO_IPIPENCAP   94        /*表示另一个 IP*/
#define IP_PROTO_MICP        95        /*表示 MICP*/
#define IP_PROTO_SCCSP       96        /*表示 SCCSP*/
#define IP_PROTO_ETHERIP     97        /*表示以太网*/
#define  IP_PROTO_ENCAP      98        /*表示封装的头部*/
```

```
#define IP_PROTO_ANYENC      99              /*表示私有加密*/
#define IP_PROTO_GMTP        100             /*表示 GMTIP*/
#define IP_PROTO_IFMP        101             /*表示 IFMP*/
#define IP_PROTO_PNNI        102             /*表示 PNNI*/
#define IP_PROTO_PIM         103             /*表示 PIM*/
#define IP_PROTO_ARIS        104             /*表示 ARIS*/
#define IP_PROTO_SCPS        105             /*表示 SCPS*/
#define IP_PROTO_QNX         106             /*表示 QNX*/
#define IP_PROTO_AN          107             /*表示主动网络*/
#define IP_PROTO_IPCOMP      108             /*表示 IP 负载压缩*/
#define IP_PROTO_SNP         109             /*表示 SNP 协议*/
#define IP_PROTO_COMPAQPEER 110              /*表示康柏对等协议*/
#define IP_PROTO_IPXIP       111             /*表示 IPX 协议*/
#define IP_PROTO_VRRP        112             /*表示 VRRP 协议*/
#define IP_PROTO_PGM         113             /*表示 PGM 可靠传输协议*/
#define IP_PROTO_ANY0HOP     114             /*表示 0-hop 协议*/
#define IP_PROTO_L2TP        115             /*表示第 2 层隧道协议 L2TP*/
#define IP_PROTO_DDX         116             /*表示 DDX*/
#define IP_PROTO_IATP        117             /*表示 IATP*/
#define IP_PROTO_STP         118             /*表示 STP 协议*/
#define IP_PROTO_SRP         119             /*表示 SRP 协议*/
#define IP_PROTO_UTI         120             /*表示 UTI*/
#define IP_PROTO_SMP         121             /*表示简单信息协议 SMP*/
#define IP_PROTO_SM          122             /*表示 SM*/
#define IP_PROTO_PTP         123             /*表示 PTP 协议*/
#define IP_PROTO_ISIS        124             /*表示 ISIS*/
#define IP_PROTO_FIRE        125             /*表示 FIRE*/
#define IP_PROTO_CRTP        126             /*表示 CRTP 协议*/
#define IP_PROTO_CRUDP       127             /*表示 CRUDP 协议*/
#define IP_PROTO_SSCOPMCE    128             /*表示 SSCOPMCE*/
#define IP_PROTO_IPLT        129             /*表示 IPLT*/
#define IP_PROTO_SPS         130             /*表示 SPS*/
#define IP_PROTO_PIPE        131             /*表示 PIPE 协议*/
#define IP_PROTO_SCTP        132             /*表示 SCTP 协议*/
#define IP_PROTO_FC          133             /*表示 FC*/
#define IP_PROTO_RSVPIGN     134             /*表示 RSVP-E2E-IGNORE*/
#define  IP_PROTO_RAW        255             /*表示原始 IP 数据包*/
#define IP_PROTO_RESERVED    IP_PROTO_RAW
#define  IP_PROTO_MAX        255
```

下面是 IP 选项的一些定义：

```
#define IP_OPT_CONTROL          0x00                    /*表示控制*/
#define IP_OPT_DEBMEAS          0x40                    /*表示调试和测试*/
#define IP_OPT_COPY             0x80
#define IP_OPT_RESERVED1        0x20
```

```
#define IP_OPT_RESERVED2          0x60
#define IP_OPT_EOL                0                                  /*表示选项末尾*/
#define IP_OPT_NOP                1                                  /*表示无操作*/
#define IP_OPT_SEC                (2|IP_OPT_COPY)                    /*表示 DoD 基本安全*/
#define IP_OPT_LSRR               (3|IP_OPT_COPY)                    /*表示松散源路由*/
#define IP_OPT_TS                 (4|IP_OPT_DEBMEAS)                 /*表示时间戳*/
#define IP_OPT_ESEC               (5|IP_OPT_COPY)                    /*表示 DoD 扩展安全*/
#define IP_OPT_CIPSO              (6|IP_OPT_COPY)                    /*表示商业安全*/
#define IP_OPT_RR                 7                                  /*标记记录路由*/
#define IP_OPT_SATID              (8|IP_OPT_COPY)                    /*表示流 ID*/
#define IP_OPT_SSRR               (9|IP_OPT_COPY)                    /*表示严格源路由*/
#define IP_OPT_ZSU                10                                 /*表示实验测试*/
#define IP_OPT_MTUP               11                                 /*表示 MTU 探测*/
#define IP_OPT_MTUR               12                                 /*表示 MTU 应答*/
#define IP_OPT_FINN               (13|IP_OPT_COPY|IP_OPT_DEBMEAS)    /*表示流控制*/
#define IP_OPT_VISA               (14|IP_OPT_COPY)                   /*表示访问控制*/
#define IP_OPT_ENCODE             15
#define IP_OPT_IMITD              (16|IP_OPT_COPY)                   /*表示 IMI 流量描述符*/
#define IP_OPT_EIP                (17|IP_OPT_COPY)                   /*表示扩展 IP*/
#define IP_OPT_TR                 (18|IP_OPT_DEBMEAS)                /*表示路由发现*/
#define IP_OPT_ADDEXT             (19|IP_OPT_COPY)
#define IP_OPT_RTRALT             (20|IP_OPT_COPY)
#define IP_OPT_SDB                (21|IP_OPT_COPY)
#define IP_OPT_NSAPA              (22|IP_OPT_COPY)                   /*表示 NSAP 地址*/
#define IP_OPT_DPS                (23|IP_OPT_COPY)                   /*表示动态数据包状态*/
#define IP_OPT_UMP                (24|IP_OPT_COPY)
#define IP_OPT_MAX                25
#define IP_OPT_COPIED(o)          ((o) & 0x80)
#define IP_OPT_CLASS(o)           ((o) & 0x60)
#define IP_OPT_NUMBER(o)          ((o) & 0x1f)
#define IP_OPT_TYPEONLY(o)        ((o) == IP_OPT_EOL || (o) == IP_OPT_NOP)
struct ip_opt_data_sec {
uint16_t s;
uint16_t c;
uint16_t h;
uint8_t       tcc[3];
} __attribute__((__packed__));
#define IP_OPT_SEC_UNCLASS        0x0000
#define IP_OPT_SEC_CONFID         0xf135
#define IP_OPT_SEC_EFTO           0x789a
#define IP_OPT_SEC_MMMM           0xbc4d
#define IP_OPT_SEC_PROG           0x5e26
#define IP_OPT_SEC_RESTR          0xaf13
```

```
#define IP_OPT_SEC_SECRET        0xd788
#define IP_OPT_SEC_TOPSECRET     0x6bc5
struct ip_opt_data_rr {
uint8_t       ptr;
uint32_t iplist __flexarr;
} __attribute__((__packed__));
struct ip_opt_data_ts {
uint8_t       ptr;
#if DNET_BYTESEX == DNET_BIG_ENDIAN
uint8_t       oflw:4,
         flg:4;
#elif DNET_BYTESEX == DNET_LIL_ENDIAN
uint8_t       flg:4,
         oflw:4;
#endif
uint32_t ipts __flexarr;
} __attribute__((__packed__));
#define IP_OPT_TS_TSONLY    0
#define IP_OPT_TS_TSADDR    1
#define IP_OPT_TS_PRESPEC   3
struct ip_opt_data_tr {
uint16_t id;
uint16_t ohc;
uint16_t rhc;
uint32_t origip;
} __attribute__((__packed__));
```

下面是 IP 选项的数据结构的定义：

```
struct ip_opt {
uint8_t       opt_type;                /*选项类型*/
uint8_t       opt_len;                 /*选项长度*/
union ip_opt_data {
     struct ip_opt_data_sec  sec;
     struct ip_opt_data_rr   rr;
     struct ip_opt_data_ts   ts;
     uint16_t       satid;
     uint16_t       mtu;
     struct ip_opt_data_tr   tr;
     uint32_t       addext[2];
     uint16_t       rtralt;
     uint32_t       sdb[9];
     uint8_t             data8[IP_OPT_LEN_MAX - IP_OPT_LEN];
} opt_data;
} __attribute__((__packed__));
```

```
#define  IP_CLASSA(i)        (((uint32_t)(i) & htonl(0x80000000)) == \
                 htonl(0x00000000))
#define  IP_CLASSA_NET       (htonl(0xff000000))
#define  IP_CLASSA_NSHIFT    24
#define  IP_CLASSA_HOST      (htonl(0x00ffffff))
#define  IP_CLASSA_MAX       128
#define  IP_CLASSB(i)        (((uint32_t)(i) & htonl(0xc0000000)) == \
                 htonl(0x80000000))
#define  IP_CLASSB_NET       (htonl(0xffff0000))
#define  IP_CLASSB_NSHIFT    16
#define  IP_CLASSB_HOST      (htonl(0x0000ffff))
#define  IP_CLASSB_MAX       65536
#define  IP_CLASSC(i)        (((uint32_t)(i) & htonl(0xe0000000)) == \
                 htonl(0xc0000000))
#define  IP_CLASSC_NET       (htonl(0xffffff00))
#define  IP_CLASSC_NSHIFT    8
#define  IP_CLASSC_HOST      (htonl(0x000000ff))
#define  IP_CLASSD(i)        (((uint32_t)(i) & htonl(0xf0000000)) == \
                 htonl(0xe0000000))
#define  IP_CLASSD_NET       (htonl(0xf0000000))
#define  IP_CLASSD_NSHIFT    28
#define  IP_CLASSD_HOST      (htonl(0x0fffffff))
#define  IP_MULTICAST(i)          IP_CLASSD(i)
#define  IP_EXPERIMENTAL(i) (((uint32_t)(i) & htonl(0xf0000000)) == \
                 htonl(0xf0000000))
#define  IP_BADCLASS(i)      (((uint32_t)(i) & htonl(0xf0000000)) == \
                 htonl(0xf0000000))
#define  IP_LOCAL_GROUP(i)   (((uint32_t)(i) & htonl(0xffffff00)) == \
                 htonl(0xe0000000))
#define IP_ADDR_ANY              (htonl(0x00000000))     /*0.0.0.0*/
#define IP_ADDR_BROADCAST        (htonl(0xffffffff))     /*255.255.255.255*/
#define IP_ADDR_LOOPBACK         (htonl(0x7f000001))     /*127.0.0.1*/
#define IP_ADDR_MCAST_ALL        (htonl(0xe0000001))     /*224.0.0.1*/
#define IP_ADDR_MCAST_LOCAL      (htonl(0xe00000ff))     /*224.0.0.255*/
#define ip_pack_hdr(hdr, tos, len, id, off, ttl, p, src, dst) do {     \
struct ip_hdr *ip_pack_p = (struct ip_hdr *)(hdr);          \
ip_pack_p->ip_v = 4; ip_pack_p->ip_hl = 5;          \
ip_pack_p->ip_tos = tos; ip_pack_p->ip_len = htons(len);\
ip_pack_p->ip_id = htons(id); ip_pack_p->ip_off = htons(off);     \
ip_pack_p->ip_ttl = ttl; ip_pack_p->ip_p = p;          \
ip_pack_p->ip_src = src; ip_pack_p->ip_dst = dst;          \
} while (0)
typedef struct ip_handle ip_t;
```

4. ICMP 协议结构

```
#define ICMP_HDR_LEN    4                    /*表示 ICMP 协议头的长度为 4 字节*/
#define ICMP_LEN_MIN    8                    /*表示最小 ICMP 消息大小为 8 字节*/
```

下面是 ICMP 协议格式的数据结构的定义：

```
struct icmp_hdr {
uint8_t        icmp_type;                    /*ICMP 类型*/
uint8_t        icmp_code;                    /*ICMP 操作码*/
uint16_t  icmp_cksum;                        /*校验和*/
};
```

下面是 ICMP 类型和操作码的定义：

```
#define        ICMP_CODE_NONE                0
#define        ICMP_ECHOREPLY                0    /*表示回显应答*/
#define        ICMP_UNREACH                  3    /*表示目的不可达*/
#define        ICMP_UNREACH_NET              0    /*表示网络不可达*/
#define        ICMP_UNREACH_HOST             1    /*表示主机不可达*/
#define        ICMP_UNREACH_PROTO            2    /*表示协议不可达*/
#define        ICMP_UNREACH_PORT             3    /*表示端口不可达*/
#define        ICMP_UNREACH_NEEDFRAG         4    /*表示需要分片*/
#define        ICMP_UNREACH_SRCFAIL          5    /*表示源站选路失败*/
#define        ICMP_UNREACH_NET_UNKNOWN      6    /*表示目的网络不认识*/
#define        ICMP_UNREACH_HOST_UNKNOWN     7    /*表示目的主机不认识*/
#define        ICMP_UNREACH_ISOLATED         8    /*表示源主机被隔离*/
#define        ICMP_UNREACH_NET_PROHIB       9    /*表示目的网络被禁止*/
#define        ICMP_UNREACH_HOST_PROHIB      10   /*表示目的主机被禁止*/
#define        ICMP_UNREACH_TOSNET           11   /*表示由于 TOS，网路不可达*/
#define        ICMP_UNREACH_TOSHOST          12   /*表示由于 TOS，主机不可达*/
#define        ICMP_UNREACH_FILTER_PROHIB    13   /*表示由于过滤，通信被禁止*/
#define        ICMP_UNREACH_HOST_PRECEDENCE  14   /*表示主机越权*/
#define        ICMP_UNREACH_PRECEDENCE_CUTOFF 15  /*表示优先权中止生效*/
#define        ICMP_SRCQUENCH                4    /*表示源端被关闭*/
#define        ICMP_REDIRECT                 5    /*表示重定向*/
#define        ICMP_REDIRECT_NET             0    /*表示对网络重定向*/
#define        ICMP_REDIRECT_HOST            1    /*表示对主机重定向*/
#define        ICMP_REDIRECT_TOSNET          2    /*表示对 TOS 和网络重定向*/
#define        ICMP_REDIRECT_TOSHOST         3    /*表示对 TOS 和主机重定向*/
#define        ICMP_ALTHOSTADDR              6
#define        ICMP_ECHO                     8    /*表示请求回显*/
#define        ICMP_RTRADVERT                9    /*表示路由器通告*/
#define        ICMP_RTRADVERT_NORMAL         0
#define        ICMP_RTRADVERT_NOROUTE_COMMON 16
#define        ICMP_RTRSOLICIT               10
#define        ICMP_TIMEXCEED                11   /*表示超时*/
```

```
#define        ICMP_TIMEXCEED_INTRANS            0    /*表示传输时 TTL 为 0*/
#define        ICMP_TIMEXCEED_REASS              1    /*表示封装时 TTL 为 0*/
#define        ICMP_PARAMPROB                    12   /*表示参数问题*/
#define        ICMP_PARAMPROB_ERRATPTR           0    /*表示坏的 IP 首部*/
#define        ICMP_PARAMPROB_OPTABSENT          1    /*表示缺少必须的选项*/
#define        ICMP_PARAMPROB_LENGTH             2    /*表示坏的长度*/
#define        ICMP_TSTAMP                       13   /*表示时间戳请求*/
#define        ICMP_TSTAMPREPLY                  14   /*表示时间戳应答*/
#define        ICMP_INFO                         15   /*表示信息请求*/
#define        ICMP_INFOREPLY                    16   /*表示信息应答*/
#define        ICMP_MASK                         17   /*表示地址掩码请求*/
#define        ICMP_MASKREPLY                    18   /*表示地址掩码应答*/
#define        ICMP_TRACEROUTE                   30   /*表示路由发现*/
#define        ICMP_DATACONVERR                  31   /*表示数据转换错误*/
#define        ICMP_MOBILE_REDIRECT              32   /*表示移动主机重定向*/
#define        ICMP_IPV6_WHEREAREYOU             33   /*表示查询 IPv6 的位置*/
#define        ICMP_IPV6_IAMHERE                 34   /*表示 IPv6 位置的应答*/
#define        ICMP_MOBILE_REG                   35   /*表示移动注册查询*/
#define        ICMP_MOBILE_REGREPLY              36   /*表示移动注册应答*/
#define        ICMP_DNS                          37   /*表示域名查询*/
#define        ICMP_DNSREPLY                     38   /*表示域名应答*/
#define        ICMP_SKIP                         39   /*表示 SKIP*/
#define        ICMP_PHOTURIS                     40   /*表示 Photuris*/
#define        ICMP_PHOTURIS_UNKNOWN_INDEX       0    /*表示未知序号*/
#define        ICMP_PHOTURIS_AUTH_FAILED         1    /*表示确认失败*/
#define        ICMP_PHOTURIS_DECOMPRESS_FAILED   2    /*表示解压失败*/
#define        ICMP_PHOTURIS_DECRYPT_FAILED      3    /*表示译文失败*/
#define        ICMP_PHOTURIS_NEED_AUTHN          4    /*表示无确认*/
#define        ICMP_PHOTURIS_NEED_AUTHZ          5    /*表示无确认*/
#define  ICMP_TYPE_MAX      40
#define  ICMP_INFOTYPE(type)                                    \
((type) == ICMP_ECHOREPLY || (type) == ICMP_ECHO ||             \
(type) == ICMP_RTRADVERT || (type) == ICMP_RTRSOLICIT ||        \
(type) == ICMP_TSTAMP || (type) == ICMP_TSTAMPREPLY ||          \
(type) == ICMP_INFO || (type) == ICMP_INFOREPLY ||              \
(type) == ICMP_MASK || (type) == ICMP_MASKREPLY)
```

下面是回显信息数据结构的定义：

```
struct icmp_msg_echo {
uint16_t  icmp_id;                          /*ICMP 的 ID 号*/
uint16_t  icmp_seq;                         /*ICMP 的序列号*/
uint8_t        icmp_data __flexarr;         /*选项数据*/
};
```

以下是需要分片的信息的数据结构定义：

```
uint16_t icmp_ohc;                          /*跳数*/
uint16_t icmp_rhc;                          /*返回的跳数*/
uint32_t icmp_speed;                        /*速度*/
uint32_t icmp_mtu;                          /*MTU 大小*/
};
```

以下是域名应答信息的数据结构定义：

```
struct icmp_msg_dnsreply {
uint16_t icmp_id;                           /*ID 号*/
uint16_t icmp_seq;                          /*序列号*/
uint32_t icmp_ttl;                          /*TTL*/
uint8_t       icmp_names __flexarr;         /*变量数目*/
};
```

以下是一般 ID 号和序列码的数据结构定义：

```
struct icmp_msg_idseq {
uint16_t icmp_id;                           /*ID 号*/
uint16_t icmp_seq;                          /*序列码*/
};
```

以下是 ICMP 消息公用体的数据结构定义：

```
union icmp_msg {
struct icmp_msg_echo       echo;            /*ICMP 回显*/
struct icmp_msg_quote      unreach;         /*目的不可达*/
struct icmp_msg_needfrag   needfrag;        /*需要分片*/
struct icmp_msg_quote      srcquench;
struct icmp_msg_quote      redirect;        /*重定向*/
uint32_t          rtrsolicit;
struct icmp_msg_rtradvert  rtradvert;       /*路由器通告*/
struct icmp_msg_quote      timexceed;       /*超时*/
struct icmp_msg_quote      paramprob;       /*参数问题*/
struct icmp_msg_tstamp     tstamp;          /*时间戳*/
struct icmp_msg_idseq      info;            /*消息*/
struct icmp_msg_mask       mask;            /*地址掩码*/
struct icmp_msg_traceroute traceroute;      /*路由发现*/
struct icmp_msg_idseq      dns;             /*域名请求*/
struct icmp_msg_dnsreply   dnsreply;        /*域名应答*/
};
#define icmp_pack_hdr(hdr, type, code) do {                    \
struct icmp_hdr *icmp_pack_p = (struct icmp_hdr *)(hdr);       \
icmp_pack_p->icmp_type = type; icmp_pack_p->icmp_code = code;  \
} while (0)
#define icmp_pack_hdr_echo(hdr, type, code, id, seq, data, len) do {  \
struct icmp_msg_echo *echo_pack_p = (struct icmp_msg_echo *)          \
     ((uint8_t *)(hdr) + ICMP_HDR_LEN);                               \
```

```
icmp_pack_hdr(hdr, type, code);                    \
echo_pack_p->icmp_id = htons(id);                  \
echo_pack_p->icmp_seq = htons(seq);                \
memmove(echo_pack_p->icmp_data, data, len);        \
} while (0)
#define icmp_pack_hdr_quote(hdr, type, code, word, pkt, len) do { \
struct icmp_msg_quote *quote_pack_p = (struct icmp_msg_quote *)   \
    ((uint8_t *)(hdr) + ICMP_HDR_LEN);             \
icmp_pack_hdr(hdr, type, code);                    \
quote_pack_p->icmp_void = htonl(word);             \
memmove(quote_pack_p->icmp_ip, pkt, len);          \
} while (0)
#define icmp_pack_hdr_mask(hdr, type, code, id, seq, mask) do {        \
struct icmp_msg_mask *mask_pack_p = (struct icmp_msg_mask *) \
    ((uint8_t *)(hdr) + ICMP_HDR_LEN);             \
icmp_pack_hdr(hdr, type, code);                    \
mask_pack_p->icmp_id = htons(id);                  \
mask_pack_p->icmp_seq = htons(seq);                \
mask_pack_p->icmp_mask = htonl(mask);              \
} while (0)
#define icmp_pack_hdr_needfrag(hdr, type, code, mtu, pkt, len) do {    \
struct icmp_msg_needfrag *frag_pack_p =                  \
(struct icmp_msg_needfrag *)((uint8_t *)(hdr) + ICMP_HDR_LEN);    \
icmp_pack_hdr(hdr, type, code);                    \
frag_pack_p->icmp_void = 0;                        \
frag_pack_p->icmp_mtu = htons(mtu);                \
memmove(frag_pack_p->icmp_ip, pkt, len);           \
} while (0)
```

5. IPv6 协议结构

```
#define IP6_ADDR_LEN    16                  /*表示 IPv6 地址长度为 16 字节*/
#define IP6_ADDR_BITS   128                 /*表示 IPv6 地址长度为 128 位*/
#define IP6_HDR_LEN     40                  /*表示 IPv6 协议头长度*/
#define IP6_LEN_MIN     IP6_HDR_LEN
#define IP6_LEN_MAX     65535               /*表示最大协议头长度*/
#define IP6_MTU_MIN     1280                /*最小 MTU 大小*/
```

以下是 IPv6 地址定义：

```
typedef struct ip6_addr {
uint8_t        data[IP6_ADDR_LEN];          /*存放 IPv6 地址*/
} ip6_addr_t;
```

以下是 IPv6 协议头的数据结构的定义：

```
struct ip6_hdr {
union {
```

```
    struct ip6_hdr_ctl {
    uint32_t  ip6_un1_flow;                    /*ID号*/
    uint16_t  ip6_un1_plen;                    /*负载长度*/
    uint8_t       ip6_un1_nxt;                 /*下一个头*/
    uint8_t       ip6_un1_hlim;                /*跳数限制*/
    } ip6_un1;
    uint8_t  ip6_un2_vfc;                      /*版本*/
} ip6_ctlun;
ip6_addr_t    ip6_src;
ip6_addr_t    ip6_dst;
} __attribute__((__packed__));
#define ip6_vfc         ip6_ctlun.ip6_un2_vfc
#define ip6_flow   ip6_ctlun.ip6_un1.ip6_un1_flow
#define ip6_plen   ip6_ctlun.ip6_un1.ip6_un1_plen
#define ip6_nxt         ip6_ctlun.ip6_un1.ip6_un1_nxt
#define ip6_hlim   ip6_ctlun.ip6_un1.ip6_un1_hlim
#define IP6_VERSION          0x60
#define IP6_VERSION_MASK     0xf0
#if DNET_BYTESEX == DNET_BIG_ENDIAN
#define IP6_FLOWINFO_MASK    0x0fffffff
#define IP6_FLOWLABEL_MASK   0x000fffff
#elif DNET_BYTESEX == DNET_LIL_ENDIAN
#define IP6_FLOWINFO_MASK    0xffffff0f
#define IP6_FLOWLABEL_MASK   0xffff0f00
#endif
#define IP6_HLIM_DEFAULT     64
#define IP6_HLIM_MAX         255
```

以下是路由数据结构的定义：

```
struct ip6_ext_data_routing {
uint8_t  type;                                 /*路由类型*/
uint8_t  segleft;
} __attribute__((__packed__));
struct ip6_ext_data_routing0 {
uint8_t  type;
uint8_t  segleft;
uint8_t  reserved;
uint8_t  slmap[3];
ip6_addr_t  addr[1];
} __attribute__((__packed__));
```

以下碎片的数据结构定义：

```
struct ip6_ext_data_fragment {
uint16_t  offlg;
uint32_t  ident;
```

```
} __attribute__((__packed__));
#if DNET_BYTESEX == DNET_BIG_ENDIAN
#define IP6_OFF_MASK          0xfff8
#define IP6_RESERVED_MASK     0x0006
#define IP6_MORE_FRAG         0x0001
#elif DNET_BYTESEX == DNET_LIL_ENDIAN
#define IP6_OFF_MASK          0xf8ff
#define IP6_RESERVED_MASK     0x0600
#define IP6_MORE_FRAG         0x0100
#endif
```

以下是选项类型的一些定义：

```
#define IP6_OPT_PAD1              0x00
#define IP6_OPT_PADN              0x01
#define IP6_OPT_JUMBO             0xC2
#define IP6_OPT_JUMBO_LEN         6
#define IP6_OPT_RTALERT           0x05
#define IP6_OPT_RTALERT_LEN       4
#define IP6_OPT_RTALERT_MLD       0
#define IP6_OPT_RTALERT_RSVP      1
#define IP6_OPT_RTALERT_ACTNET    2
#define IP6_OPT_LEN_MIN           2
#define IP6_OPT_TYPE(o)           ((o) & 0xC0)
#define IP6_OPT_TYPE_SKIP         0x00
#define IP6_OPT_TYPE_DISCARD      0x40
#define IP6_OPT_TYPE_FORCEICMP    0x80
#define IP6_OPT_TYPE_ICMP         0xC0
#define IP6_OPT_MUTABLE           0x20
struct ip6_ext_hdr {
uint8_t  ext_nxt;
uint8_t  ext_len;
union {
     struct ip6_ext_data_routing routing;
     struct ip6_ext_data_fragmentfragment;
} ext_data;
} __attribute__((__packed__));
#define IP6_ADDR_UNSPEC \
"\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00"
#define IP6_ADDR_LOOPBACK \
"\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x01"
#define ip6_pack_hdr(hdr, fc, fl, plen, nxt, hlim, src, dst) do { \
struct ip6_hdr *ip6 = (struct ip6_hdr *)(hdr);            \
ip6->ip6_flow = htonl(((uint32_t)(fc) << 28) &            \
    (IP6_FLOWLABEL_MASK | (fl)));                         \
```

```
ip6->ip6_vfc = (IP6_VERSION | ((fc) >> 4));                    \
ip6->ip6_plen = htons((plen));                                 \
ip6->ip6_nxt = (nxt); ip6->ip6_hlim = (hlim);                  \
memmove(&ip6->ip6_src, &(src), IP6_ADDR_LEN);                  \
memmove(&ip6->ip6_dst, &(dst), IP6_ADDR_LEN);                  \
} while (0);
```

6. TCP 协议数据结构

```
#define TCP_HDR_LEN          20                    /*表示 TCP 协议长度*/
#define TCP_OPT_LEN          2                     /*表示 TCP 选项长度*/
#define TCP_OPT_LEN_MAX      40                    /*表示 TCP 选项最大长度*/
#define TCP_HDR_LEN_MAX (TCP_HDR_LEN + TCP_OPT_LEN_MAX)
```

以下是 TCP 协议头的数据结构定义：

```
struct tcp_hdr {
uint16_t th_sport;                                 /*源端口*/
uint16_t th_dport;                                 /*目的端口*/
uint32_t th_seq;                                   /*序列号*/
uint32_t th_ack;                                   /*确认号*/
#if DNET_BYTESEX == DNET_BIG_ENDIAN
uint8_t       th_off:4,                            /*偏移*/
         th_x2:4;                                  /*未用*/
#elif DNET_BYTESEX == DNET_LIL_ENDIAN
uint8_t       th_x2:4,
         th_off:4;
#else
# error "need to include <dnet.h>"
#endif
uint8_t       th_flags;                            /*标记*/
uint16_t th_win;                                   /*窗口大小*/
uint16_t th_sum;                                   /*校验和*/
uint16_t th_urp;                                   /*紧急指针*/
};
```

以下是控制标记的数据结构定义：

```
#define TH_FIN     0x01                            /*表示发送端完成发送任务*/
#define TH_SYN     0x02                            /*表示用来发起一个连接*/
#define TH_RST     0x04                            /*表示重建连接*/
#define TH_PUSH    0x08                            /*表示接收方应该尽快将这个报文段交给应用层*/
#define TH_ACK     0x10                            /*表示确认序号有效*/
#define TH_URG     0x20                            /*表示紧急指针有效*/
#define TH_ECE     0x40                            /*表示 ECN 回显*/
#define TH_CWR     0x80                            /*表示拥塞窗口*/
#define TCP_PORT_MAX    65535                      /*表示最大端口号*/
#define TCP_WIN_MAX     65535                      /*表示最大窗口*/
```

```
#define TCP_SEQ_LT(a,b)      ((int)((a)-(b)) < 0)
#define TCP_SEQ_LEQ(a,b)     ((int)((a)-(b)) <= 0)
#define TCP_SEQ_GT(a,b)      ((int)((a)-(b)) > 0)
#define TCP_SEQ_GEQ(a,b)     ((int)((a)-(b)) >= 0)
```

以下是 TCP 状态的定义：

```
#define TCP_STATE_CLOSED          0          /*表示关闭*/
#define TCP_STATE_LISTEN          1          /*表示监听连接*/
#define TCP_STATE_SYN_SENT        2          /*表示主动打开*/
#define TCP_STATE_SYN_RECEIVED    3          /*表示接收*/
#define TCP_STATE_ESTABLISHED     4          /*表示建立连接*/
#define TCP_STATE_CLOSE_WAIT      5          /*表示等待关闭*/
#define TCP_STATE_FIN_WAIT_1      6          /*表示已经关闭*/
#define TCP_STATE_CLOSING         7          /*表示同时关闭*/
#define TCP_STATE_LAST_ACK        8          /*表示被动关闭*/
#define TCP_STATE_FIN_WAIT_2      9          /*表示主动关闭*/
#define TCP_STATE_TIME_WAIT       10         /*表示 2MSL 超时*/
#define TCP_STATE_MAX             11
```

以下是选项类型的一些定义：

```
#define TCP_OPT_EOL               0
#define TCP_OPT_NOP               1
#define TCP_OPT_MSS               2          /*表示最大段大小*/
#define TCP_OPT_WSCALE            3          /*表示窗口规模参数*/
#define TCP_OPT_SACKOK            4          /*表示 SACK 禁止*/
#define TCP_OPT_SACK              5          /*表示 SACK*/
#define TCP_OPT_ECHO              6          /*表示回显*/
#define TCP_OPT_ECHOREPLY         7          /*表示回显应答*/
#define TCP_OPT_TIMESTAMP         8          /*表示时间戳*/
#define TCP_OPT_POCONN            9
#define TCP_OPT_POSVC             10
#define TCP_OPT_CC                11         /*表示连接数*/
#define TCP_OPT_CCNEW             12         /*表示 CC.NEW*/
#define TCP_OPT_CCECHO            13         /*表示 CC.ECHO*/
#define TCP_OPT_ALTSUM            14
#define TCP_OPT_ALTSUMDATA        15
#define TCP_OPT_SKEETER           16         /*表示 Skeeter*/
#define TCP_OPT_BUBBA             17         /*表示 Bubba*/
#define TCP_OPT_TRAILSUM          18
#define TCP_OPT_MD5               19         /*表示 MD5 签名*/
#define TCP_OPT_SCPS              20         /*表示 SCPS 容量*/
#define TCP_OPT_SNACK             21
#define TCP_OPT_REC               22
#define TCP_OPT_CORRUPT           23
#define TCP_OPT_SNAP              24         /*表示 SNAP*/
```

```
struct udp_hdr *udp_pack_p = (struct udp_hdr *)(hdr);    \
udp_pack_p->uh_sport = htons(sport);                     \
udp_pack_p->uh_dport = htons(dport);                     \
udp_pack_p->uh_ulen = htons(ulen);                       \
} while (0)
```

6.3.3 ARP 缓存的数据结构

在使用 Libdnet 对 ARP 缓存进行操作时，要用到 ARP 缓存的数据结构。ARP 缓存的数据结构中定义了与 ARP 缓存相关的一些信息，如协议地址和硬件地址。ARP 缓存结构的定义在 arp.h 中。

以下是 Libdnet 中 ARP 缓存的结构定义。

```
struct arp_entry {
struct addr   arp_pa;            /*协议地址*/
struct addr   arp_ha;            /*硬件地址*/
};
#define arp_pack_hdr_ethip(hdr, op, sha, spa, tha, tpa) do { \
struct arp_hdr *pack_arp_p = (struct arp_hdr *)(hdr);    \
struct arp_ethip *pack_ethip_p = (struct arp_ethip *)    \
     ((uint8_t *)(hdr) + ARP_HDR_LEN);                   \
pack_arp_p->ar_hrd = htons(ARP_HRD_ETH);                 \
pack_arp_p->ar_pro = htons(ARP_PRO_IP);                  \
pack_arp_p->ar_hln = ETH_ADDR_LEN;                       \
pack_arp_p->ar_pln = IP_ADDR_LEN;                        \
pack_arp_p->ar_op = htons(op);                           \
memmove(pack_ethip_p->ar_sha, &(sha), ETH_ADDR_LEN);     \
memmove(pack_ethip_p->ar_spa, &(spa), IP_ADDR_LEN);      \
memmove(pack_ethip_p->ar_tha, &(tha), ETH_ADDR_LEN);     \
memmove(pack_ethip_p->ar tpa, &(tpa), IP_ADDR_LEN);      \
} while (0)
typedef struct arp_handle arp_t;
typedef int (*arp_handler)(const struct arp_entry *entry, void *arg);
```

6.3.4 网络路由表数据结构

在 Libdnet 中有对路由表操作的功能，其中要用到网络路由表的数据结构。

网络路由表数据结构的定义在文件 route.h 中。该文件主要存储与网络路由相关的一些信息，如目的地址和网关地址等。

以下是路由结构的定义：

```
struct route_entry {
     struct addr   route_dst;          /*目的地址*/
     struct addr   route_gw;           /*网关地址*/
};
typedef struct route_handle route_t;
```

```
typedef int (*route_handler)(const struct route_entry *entry, void *arg);
```

6.3.5 网络接口的数据结构

网络接口的数据结构存储了与网络接口相关的信息。

```
#define INTF_NAME_LEN  16
```

以下是网络接口的数据结构的定义：

```
struct intf_entry {
    u_int           intf_len;                   /*长度*/
    char        intf_name[INTF_NAME_LEN];       /*接口名字*/
    u_short     intf_type;                      /*接口类型*/
    u_short     intf_flags;                     /*接口标记*/
    u_int           intf_mtu;                   /*接口 MTU*/
    struct addr     intf_addr;                  /*接口地址*/
    struct addr     intf_dst_addr;              /*点对点的目的地址*/
    struct addr     intf_link_addr;             /*链路层地址*/
    u_int           intf_alias_num;             /*别名的数目*/
    struct addr     intf_alias_addrs __flexarr; /*别名*/
};
```

以下是网络接口类型的一些定义：

```
#define INTF_TYPE_OTHER         1
#define INTF_TYPE_ETH           6       /*表示以太网类型*/
#define INTF_TYPE_TOKENRING     9       /*表示令牌环*/
#define INTF_TYPE_FDDI          15      /*表示 FDDI*/
#define INTF_TYPE_PPP           23      /*表示点对点协议*/
#define INTF_TYPE_LOOPBACK      24      /*表示回环*/
#define INTF_TYPE_SLIP          28
#define INTF_TYPE_TUN           53      /*表示私有虚拟网络*/
```

以下是接口标记的一些定义：

```
#define INTF_FLAG_UP            0x01    /*表示接口处于活动状态*/
#define INTF_FLAG_LOOPBACK      0x02    /*表示回环*/
#define INTF_FLAG_POINTOPOINT   0x04    /*表示点对点链路*/
#define INTF_FLAG_NOARP         0x08    /*表示不支持 ARP*/
#define INTF_FLAG_BROADCAST     0x10    /*表示支持广播*/
#define INTF_FLAG_MULTICAST     0x20    /*表示支持多播*/
typedef struct intf_handle intf_t;
typedef int (*intf_handler)(const struct intf_entry *entry, void *arg);
```

在使用 Libdnet 操作网络接口时要用到网络接口的数据结构。

6.3.6 防火墙结构

在 Libdnet 中实现了防火墙的管理功能，其数据结构在文件 fw.h 中定义。该文件主要描述了与防火墙规则相关的一些信息。

以下是防火墙的数据结构的定义：

```
struct fw_rule {
char      fw_device[INTF_NAME_LEN];        /*接口的名字*/
uint8_t       fw_op;                       /*操作*/
uint8_t       fw_dir;                      /*方向*/
uint8_t       fw_proto;                    /*协议类型*/
struct addr   fw_src;                      /*源地址*/
struct addr   fw_dst;                      /*目的地址*/
uint16_t  fw_sport[2];                     /*源端口号或 ICMP 类型*/
uint16_t  fw_dport[2];                     /*目的端口号或 ICMP 代码*/
};
#define FW_OP_ALLOW     1
#define FW_OP_BLOCK     2
#define FW_DIR_IN  1
#define FW_DIR_OUT 2
#define fw_pack_rule(rule, dev, op, dir, p, s, d, sp1, sp2, dp1, dp2) \
do {                                        \
strlcpy((rule)->fw_device, dev, sizeof((rule)->fw_device));  \
(rule)->fw_op = op; (rule)->fw_dir = dir;            \
(rule)->fw_proto = p;                                \
memmove(&(rule)->fw_src, &(s), sizeof((rule)->fw_src));      \
memmove(&(rule)->fw_dst, &(d), sizeof((rule)->fw_dst));      \
(rule) >fw_sport[0] = sp1; (rule)->fw_sport[1] = sp2;        \
(rule)->fw_dport[0] = dp1; (rule)->fw_dport[1] = dp2;        \
} while (0)
typedef struct fw_handle fw_t;
typedef int (*fw_handler)(const struct fw_rule *rule, void *arg);
```

在使用 Libdnet 来操作防火墙时要用到防火墙的数据结构。

6.4 Libdnet 函数

6.4.1 网络地址操作函数

网络地址操作函数是 Libdnet 中最基本的函数。它实现了很多网络地址的操作功能，包括各种地址格式之间的转换、匹配等。

（1）int addr_cmp(const struct addr *a, const struct addr *b);

函数返回值：如果 a 等于 b，就返回 0；如果 a>b，就返回正数；如果 a<b，就返回负数。

参数描述：参数 a 表示一个地址，参数 b 表示一个地址。

此函数比较两个地址的大小。注意这里的地址包括 IPv4 地址、以太网地址和 IPv6 地址。

（2）int addr_bcast(const struct addr *a, struct addr *b);

函数返回值：函数执行成功就返回 0，失败就返回 -1。

参数描述：参数 a 表示一个地址。参数 b 表示一个地址。

此函数返回一个地址 a 的广播地址 b。注意这里的地址包括 IPv4 地址、以太网地址。

（3）int　addr_net(const struct addr *a, struct addr *b);

函数返回值：函数执行成功就返回 0，失败就返回－1。

参数描述：参数 a 表示一个地址，参数 b 表示一个地址。

此函数返回一个地址 a 的掩码地址 b。注意这里的地址包括 IPv4 地址、以太网地址和 IPv6 地址。

（4）char *addr_ntop(const struct addr *src, char *dst, size_t size);

函数返回值：函数执行成功就返回一个地址字符串，失败就返回 NULL。

参数描述：参数 src 表示一个地址，参数 dst 表示一个地址字符串，参数 size 表示大小。

此函数的功能是把一个二进制地址转化成一个字符串形式，此字符串就是返回值。在这里，地址也包括三种类型，分别是 IPv4 地址、以太网地址和 IPv6 地址。

（5）int　addr_pton(const char *src, struct addr *dst);

函数返回值：函数执行成功就返回 0，失败就返回－1。

参数描述：参数 src 表示一个字符串形式的地址，既可以是 IPv4 点分十进制形式，如“192.168.0.2”，也可以是一个 IPv6 地址或以太网地址类型，如“11:11:11:11:11:11”，还可以是一个主机名字。参数 dst 表示一个转换后的地址。

此函数的功能是把一个字符串形式的地址转化为一个二进制形式的地址格式。字符串有四种类型，分别是 IPv4 型、IPv6 型、以太网地址型和主机名字型。

（6）char *addr_ntoa(const struct addr *a);

#define　addr_aton　addr_pton

函数返回值：函数执行成功就返回一个地址字符串，失败就返回 NULL。

参数描述：参数 a 表示一个地址，有三种类型，分别是 IPv4、以太网和 IPv6 地址。

此函数是把一个二进制的地址格式转化为一个字符串格式。

（7）int　addr_ntos(const struct addr *a, struct sockaddr *sa);

函数返回值：函数执行成功就返回 0，失败就返回－1。

参数描述：参数 a 表示一个 addr 结构的地址，参数 sa 表示一个 sockaddr 格式的地址。

此函数是要把一个 addr 格式的地址转换为一个 sockaddr 格式的地址。有三种类型的地址：IPv4、IPv6 和以太网地址。

（8）int　addr_ston(const struct sockaddr *sa, struct addr *a);

函数返回值：函数执行成功就返回 0，失败就返回－1。

参数描述：参数 sa 表示一个 sockaddr 格式的地址，参数 a 表示一个 addr 格式的地址。

此函数是把一个 sockaddr 格式的地址转换为一个 addr 格式的地址。地址也有三种类型：IPv6、IPv4 和以太网地址。

（9）int　addr_btos(uint16_t bits, struct sockaddr *sa);

函数返回值：函数执行成功就返回 0，失败就返回－1。

参数描述：参数 bits 表示字节数，参数 sa 表示 sockaddr 格式的地址。

此函数把字节数转化为一个 socaddr 格式的地址。

（10）int　addr_stob(const struct sockaddr *sa, uint16_t *bits);

如“11:11:11:11:11:11”。

（7）int　eth_pton(const char *src, eth_addr_t *dst);

函数返回值：函数执行成功就返回0，失败就返回-1。

参数描述：参数 src 表示的是字符串形式的以太网地址，参数 dst 表示的是 eth_addr 类型的以太网地址。

此函数的功能是把字符串形式的以太网地址转换为一个二进制形式的以太网地址。

（8）char *eth_ntoa(const eth_addr_t *eth);

```
#define    eth_aton eth_pton
```

函数返回值：返回一个字符串形式的以太网地址。

参数描述：参数 eth 表示的是 eth_addr_t 类型的以太网地址。

此函数的功能是把二进制形式的以太网地址转换为字符串形式的以太网地址。

6.4.3　IPv4 协议函数

（1）ip_t *ip_open(void);

函数返回值：函数执行成功就返回 IP 协议句柄，失败就返回 NULL。

参数描述：无。

此函数的功能是打开 IP 协议操作功能，进行初始化设置，并返回一个指向 IP 协议的句柄。

（2）ssize_t　ip_send(ip_t *i, const void *buf, size_t len);

函数返回值：返回发送的字节数。

参数描述：参数 i 表示 IP 协议句柄，参数 buf 表示存放的 IP 协议数据，参数 len 表示数据的长度。

此函数的功能是发送 IP 数据包，此 IP 数据包的内容由 buf 来指定。

（3）ip_t *ip_close(ip_t *i);

函数返回值：返回 NULL。

参数描述：参数 i 表示 IP 协议句柄。

此函数的功能是关闭 IP 协议操作并销毁 IP 协议句柄，与 ip_open()相对应。

（4）char *ip_ntop(const ip_addr_t *ip, char *dst, size_t len);

函数返回值：返回字符串形式的地址。

参数描述：参数 ip 表示 ip_addr_t 类型表示的 IP 地址，参数 dst 表示以字符串形式表示的 IP 地址，参数 len 表示长度。

此函数的功能是把二进制的 IP 地址转换为字符串形式的 IP 地址，如“192.168.0.3”。

（5）int　ip_pton(const char *src, ip_addr_t *dst);

函数返回值：函数执行成功就返回0，失败就返回-1。

参数描述：参数 src 表示字符串形式的 IP 地址，参数 dst 表示 ip_addr_t 表示的 IP 地址。

此函数的功能就是把字符串形式的 IP 地址转换为二进制形式的 IP 地址。

（6）char *ip_ntoa(const ip_addr_t *ip);

```
#define    ip_aton ip_pton
```

函数返回值：返回字符串形式的 IP 地址。

参数描述：参数 ip 表示一个 ip_addr_t 类型的 IP 地址。

此函数的功能是把一个二进制 IP 地址转换为字符串形式的 IP 地址。

（7）ssize_t ip_add_option(void *buf, size_t len,

int proto, const void *optbuf, size_t optlen);

函数返回值：函数执行成功就返回字节数，失败就返回－1。

参数描述：参数 buf 表示 IP 协议数据缓存，参数 len 表示 IP 协议数据的长度，参数 proto 表示协议类型，参数 optbuf 表示 IP 选项的数据缓存，参数 optlen 表示选项的长度。

此函数的功能是向 IP 协议数据中添加 IP 选项数据。

（8）void ip_checksum(void *buf, size_t len);

函数返回值：无。

参数描述：参数 buf 表示 IP 协议数据缓存，参数 len 表示 IP 协议数据的长度。

此函数的功能是计算 IP 校验和。

（9）int ip_cksum_add(const void *buf, size_t len, int cksum);

#define ip_cksum_carry(x) \

(x = (x >> 16) + (x & 0xffff), (~(x + (x >> 16)) & 0xffff))

函数返回值：返回校验和数据。

参数描述：参速 buf 表示数据缓存，参数 cksum 表示校验和。

此函数的功能是计算 buf 的校验和。

6.4.4 IPv6 协议函数

（1）char *ip6_ntop(const ip6_addr_t *ip6, char *dst, size_t size);

函数返回值：函数执行成功就返回字符串形式的 IPv6 地址，失败就返回 NULL。

参数描述：参数 ip6 表示 ip6_addr_t 类型的 IPv6 地址，参数 dst 表示字符串形式的 IPv6 地址，参数 size 表示大小。

此函数的功能是把二进制的 IPv6 地址转换为字符串形式的 IPv6 地址。

（2）int ip6_pton(const char *src, ip6_addr_t *dst);

函数返回值：函数执行成功就返回 0，失败就返回－1。

参数描述：参数 src 表示字符串形式的 IPv6 地址，参数 dst 表示 ip_addr_t 类型的 IPv6 地址。

此函数的功能是把字符串形式的 IPv6 地址转换为二进制形式的 IPv6 地址。

（3）char *ip6_ntoa(const ip6_addr_t *ip6);

#define ip6_aton ip6_pton

函数返回值：返回字符串形式的 IPv6 地址。

参数描述：参数 ip6 表示 ip6_addr_t 类型的地址。

此函数的功能是把二进制形式的 IPv6 地址转换为字符串形式的 IPv6 地址。

（4）void ip6_checksum(void *buf, size_t len);

函数返回值：无。

参数描述：参数 buf 表示的是 IPv6 协议数据，参数 len 表示的是长度。

此函数的功能是计算由 buf 表示的 IPv6 协议数据的校验和。

6.4.5 数据块操作函数

以下是数据块句柄的数据结构定义：

```
typedef struct blob {
u_char    *base;            /*数据开始指针*/
int        off;             /*偏移量*/
int        end;             /*数据结束*/
int        size;            /*数据大小*/
} blob_t;
```

（1）blob_t　*blob_new(void);

函数返回值：函数执行成功就返回数据块句柄，失败就返回 NULL。

参数描述：无。

此函数的功能是分配一段内存数据块，并由数据块句柄来指示。

（2）int　blob_read(blob_t *b, void *buf, int len);

函数返回值：返回长度。

参数描述：参数 b 表示数据块句柄，参数 buf 表示用来存储数据的缓存，参数 len 表示长度。

此函数的功能是把由 b 表示的数据块的内容读入 buf 缓存中。

（3）int　blob_write(blob_t *b, const void *buf, int len);

函数返回值：函数执行成功就返回长度，失败就返回－1。

参数描述：参数 b 表示数据块句柄，参数 buf 表示一段缓存，参数 len 表示长度。

此函数的功能是把由 buf 缓存表示的数据写入由 b 表示的数据块中。

（4）int　blob_seek(blob_t *b, int off, int whence);

函数返回值：函数执行成功就返回偏移量，失败就返回－1。

参数描述：参数 b 表示数据块句柄，参数 off 表示偏移量，参数 whence 表示位置。

此函数的功能是定位偏移量的位置：

```
#define  blob_skip(b, 1)     blob_seek(b, 1, SEEK_CUR)
#define  blob_rewind(b)      blob_seek(b, 0, SEEK_SET)
#define   blob_offset(b)     ((b)->off)
#define   blob_left(b)       ((b)->end - (b)->off)
```

（5）int　blob_index(blob_t *b, const void *buf, int len);

函数返回值：函数执行成功就返回匹配成功的位置，失败就返回－1。

参数描述：参数 b 表示数据块句柄，参数 buf 表示一段缓存，参数 len 表示 buf 的长度。

此函数的功能是定位 buf 字符串在由 b 表示的数据块中的位置。

（6）int　blob_rindex(blob_t *b, const void *buf, int len);

函数返回值：函数执行成功就返回匹配成功的位置，失败就返回－1。

参数描述：参数 b 表示数据块句柄，参数 buf 表示一段缓存，参数 len 表示 buf 的长度。

此函数的功能跟函数 blob_index()函数的功能相似，只不过它匹配的过程从数据块的右

边开始向左边进行匹配。

（7）int　blob_pack(blob_t *b, const char *fmt, ...);

函数返回值：函数执行成功就返回 0，失败就返回－1。

参数描述：参数 b 表示数据块句柄，参数 fmt 表示字符指针类型。

此函数的功能是根据不同个数的参数进行分配。

（8）int　blob_unpack(blob_t *b, const char *fmt, ...);

函数返回值：函数执行成功就返回 0，失败就返回－1。

参数描述：参数 b 表示数据块句柄，参数 fmt 表示字符指针类型。

此函数的功能是取消不同个数的参数分配。

（9）int　blob_insert(blob_t *b, const void *buf, int len);

函数返回值：函数执行成功就返回长度，失败就返回－1。

参数描述：参数 b 表示数据块句柄，参数 buf 表示一段缓存，参数 len 表示长度。

此函数的功能是把由 buf 表示的缓存插入由 b 表示的数据块中。

（10）int　blob_delete(blob_t *b, void *buf, int len);

函数返回值：函数执行成功就返回长度，失败就返回－1。

参数描述：参数 b 表示数据块句柄，参数 buf 表示一段缓存，参数 len 表示长度。

此函数的功能是从由 b 表示的数据块中删除一段数据并存放到 buf 中。

（11）int　blob_print(blob_t *b, char *style, int len);

函数返回值：返回 0。

参数描述：参数 b 表示数据块句柄；参数 style 表示输出类型，有“hex1”和 NULL 两种类型；参数 len 表示长度。

此函数的功能是按照一定的格式输出由 b 表示的数据块。

（12）blob_t　*blob_free(blob_t *b);

函数返回值：返回 NULL。

参数描述：参数 b 表示数据块句柄。

此函数的功能是释放由 b 表示的数据块的内存。

（13）int　blob_register_alloc(size_t size, void *(*bmalloc)(size_t),
void (*bfree)(void *), void *(*brealloc)(void *, size_t));

函数返回值：返回 0。

参数描述：参数 size 表示大小，参数 bmalloc 表示分配内存的函数指针，参数 bfree 表示释放内存的函数指针，参数 brealloc 表示重新分配内存的函数指针。

此函数的功能是注册三个函数。

（14）#ifdef va_start

typedef int (*blob_fmt_cb)(int pack, int len, blob_t *b, va_list *arg);

int　blob_register_pack(char c, blob_fmt_cb fmt_cb);

函数返回值：函数执行成功就返回 0。失败就返回－1。

参数描述：参数 c 表示一个字符，参数 fmt_cb 表示 blob_fmt_cb 类型的参数。

此函数的功能是注册 blob_fmt_cb 函数。

6.4.6 ARP 缓存操作函数

（1）arp_t　　*arp_open(void);

函数返回值：函数执行成功就返回一个 arp 缓存句柄，失败就返回 NULL。

参数描述：无。

此函数的功能是打开 ARP 缓存的管理功能，进行初始化处理，并返回一个指向 ARP 缓存的句柄。

（2）int　　arp_add(arp_t *arp, const struct arp_entry *entry);

函数返回值：函数执行成功就返回 0，失败就返回－1。

参数描述：参数 arp 表示 ARP 缓存句柄；参数 entry 表示一个 arp_entry 类型的参数，它描述了 ARP 缓存的信息。

此函数的功能是在 ARP 缓存中增加一条记录。

（3）int　　arp_delete(arp_t *arp, const struct arp_entry *entry);

函数返回值：函数执行成功就返回 0，失败就返回－1。

参数描述：参数 arp 表示 ARP 缓存句柄；参数 entry 表示一个 arp_entry 类型的参数，它描述了 ARP 缓存的信息。

此函数的功能是在 ARP 缓存中删除一条记录。

（4）int　　arp_get(arp_t *arp, struct arp_entry *entry);

函数返回值：函数执行成功就返回 0，失败就返回－1。

参数描述：参数 arp 表示 ARP 缓存句柄；参数 entry 表示一个 arp_entry 类型的参数，它描述了 ARP 缓存的信息。

此函数的功能是获得特定的 ARP 缓存记录。

（5）int　　arp_loop(arp_t *arp, arp_handler callback, void *arg);

函数返回值：函数执行成功就返回 0，失败就返回－1。

参数描述：参数 arp 表示 ARP 缓存句柄，参数 callback 表示一个回调函数，参数 arg 表示向回调函数传递的参数。

此函数的功能是注册回调函数 callback，然后循环调用 callback 来对每个 ARP 缓存记录进行操作。例如，可以在回调函数中显示所有 ARP 缓存记录的信息。

（6）arp_t　　*arp_close(arp_t *arp);

函数返回值：返回 NULL。

参数描述：参数 arp 表示 ARP 缓存句柄。

此函数的功能是关闭 ARP 缓存配置功能并销毁句柄，它与函数 arp_open() 相对应。

6.4.7 网络接口函数

（1）intf_t　　*intf_open(void);

函数返回值：函数执行成功就返回一个接口的句柄，失败就返回 NULL。

参数描述：无。

此函数的功能是打开一个网络接口，进行初始化设置，并返回这个网络接口的句柄。

（2）int　　intf_get(intf_t *i, struct intf_entry *entry);

函数返回值：函数执行成功就返回 0，失败就返回－1。

参数描述：参数 i 表示一个网络接口句柄，此句柄是由 intf_open()函数返回的；参数 entry 表示的是一个 intf_entry 类型的参数。

此函数的功能是返回特定的网络接口信息。

（3）int　intf_get_src(intf_t *i, struct intf_entry *entry, struct addr *src);

函数返回值：函数执行成功就返回 0，失败就返回－1。

参数描述：参数 i 表示一个网络接口句柄，参数 entry 表示一个 intf_entry 类型的参数，参数 src 表示一个网络地址。

此函数的功能是返回满足地址的网络接口信息。

（4）int　intf_get_dst(intf_t *i, struct intf_entry *entry, struct addr *dst);

函数返回值：函数执行成功就返回 0，失败就返回－1。

参数描述：参数 i 表示网络接口句柄，参数 entry 表示一个 intf_entry 类型的参数，参数 dst 表示的是目的网络地址。

此函数的功能是返回满足目的网络地址的网络接口信息。

（5）int　intf_set(intf_t *i, const struct intf_entry *entry);

函数返回值：函数执行成功就返回 0，失败就返回－1。

参数描述：参数 i 表示网络接口句柄；参数 entry 表示一个 intf_entry 类型的参数，描述的是一个网络接口的信息。

此函数的功能是对一个网络接口进行参数设置。

（6）int　intf_loop(intf_t *i, intf_handler callback, void *arg);

函数返回值：函数执行成功就返回 0，失败就返回－1。

参数描述：参数 i 表示网络接口句柄，参数 callback 表示一个回调函数，参数 arg 表示传递给回调函数的参数。

此函数的功能是注册回调函数 callback，然后该调用回调函数来执行对每个网络接口的操作。例如，可以在回调函数中实现显示所有网络接口信息的功能。

（7）intf_t　*intf_close(intf_t *i);

函数返回值：返回 NULL。

参数描述：参数 i 表示一个网络接口句柄。

此函数的功能是关闭由 i 表示的网络接口并销毁一些设置，它与 intf_open() 相对应。

6.4.8　网络路由函数

（1）route_t　*route_open(void);

函数返回值：函数执行成功就返回路由句柄，失败就返回 NULL。

参数描述：无。

此函数的功能是打开路由操作，进行初始化处理，并返回一个路由句柄。

（2）int　route_add(route_t *r, const struct route_entry *entry);

函数返回值：函数执行成功就返回 0，失败就返回－1。

参数描述：参数 r 表示一个路由句柄；参数 entry 表示一个 route_entry 类型的参数，它描述了一个路由表信息。

此函数的功能是增加一个路由记录。

（3）int　route_delete(route_t *r, const struct route_entry *entry);

函数返回值：函数执行成功就返回0，失败就返回-1。

参数描述：参数r表示一个路由句柄，参数entry表示一个route_entry类型的参数，它描述了一个路由表信息。

此函数的功能是删除一个路由记录。

（4）int route_get(route_t *r, struct route_entry *entry);

函数返回值：函数执行成功就返回0，失败就返回-1。

参数描述：参数r表示一个路由句柄；参数entry表示一个route_entry类型的参数，它描述了一个路由表信息。

此函数的功能是返回特定的路由记录。

（5）int route_loop(route_t *r, route_handler callback, void *arg);

函数返回值：函数执行成功就返回0，失败就返回-1。

参数描述：参数r表示一个路由句柄，参数callback表示一个回调函数，参数arg是传递回调函数的参数。

此函数的功能是注册回调函数，然后循环调用回调函数对每个路由表进行操作。例如，可以在回调函数中显示所有路由表信息。

（6）route_t *route_close(route_t *r);

函数返回值：返回NULL。

参数描述：参数r表示一个路由句柄。

此函数是关闭由r表示的路由句柄，它与函数route_close()函数相对应。

6.4.9 随机数函数

以下是随机数的数据结构的定义。

```
struct rand_handle {
    uint8_t        i;
    uint8_t        j;
    uint8_t        s[256];
    u_char        *tmp;
    int           tmplen;
};
typedef struct rand_handle rand_t;
```

（1）rand_t *rand_open(void);

函数返回值：返回随机数句柄。

参数描述：无。

此函数的功能是构造随机数，并返回随机数句柄。

（2）int rand_get(rand_t *r, void *buf, size_t len);

函数返回值：返回0。

参数描述：参数r表示随机数句柄，参数buf表示一段缓存，参数len表示长度。

此函数的功能是读取由r表示的随机数，由buf返回。

（3）int rand_set(rand_t *r, const void *seed, size_t len);

函数返回值：返回0。

参数描述：参数 r 表示随机数句柄，参数 seed 表示随机种子，参数 len 表示长度。

此函数的功能是根据 seek 来构造随机数。

（4）int　rand_add(rand_t *r, const void *buf, size_t len);

函数返回值：返回 0。

参数描述：参数 r 表示随机数句柄，参数 buf 表示一段缓存，参数 len 表示长度。

此函数的功能是根据 buf 来构造随机数。

（5）uint8_t　rand_uint8(rand_t *r);

函数返回值：返回随机数。

参数描述：参数 r 表示随机数句柄。

此函数的功能是返回由 r 表示的 8 位随机数。

（6）uint16_t rand_uint16(rand_t *r);

函数返回值：返回随机数。

参数描述：参数 r 表示随机数句柄。

此函数的功能是返回由 r 表示的 16 位随机数。

（7）uint32_t rand_uint32(rand_t *r);

函数返回值：返回随机数。

参数描述：参数 r 表示随机数句柄。

此函数的功能是返回由 r 表示的 32 位随机数。

（8）int　rand_shuffle(rand_t *r, void *base, size_t nmemb, size_t size);

函数返回值：函数执行成功就返回 0，失败就返回 -1。

参数描述：参数 r 表示随机数句柄，参数 base 表示一段内存，参速 nmemb 表示大小，参数 size 也表示大小。

此函数的功能是重新设置随机数。

（9）rand_t　*rand_close(rand_t *r);

函数返回值：返回 NULL。

参数描述：参数 r 表示随机数句柄。

此函数的功能是释放随机数。

6.4.10　防火墙函数

（1）fw_t*fw_open(void);

函数返回值：函数执行成功就返回防火墙操作句柄，失败就返回 NULL。

参数描述：无。

此函数的功能是打开防火墙操作，进行初始化，并返回一个防火墙句柄。

（2）int　fw_add(fw_t *f, const struct fw_rule *rule);

函数返回值：函数执行成功就返回 0，失败就返回 -1。

参数描述：参数 f 表示一个防火墙句柄，参数 rule 表示防火墙规则。

此函数的功能是向防火墙系统添加防火墙规则。

（3）int　fw_delete(fw_t *f, const struct fw_rule *rule);

函数返回值：函数执行成功就返回 0，失败就返回 -1。

参数描述：参数 f 表示防火墙句柄，参数 rule 表示防火墙规则。

此函数的功能是从防火墙系统中删除防火墙规则。

（4）int fw_loop(fw_t *f, fw_handler callback, void *arg);

函数返回值：函数执行成功就返回0，失败就返回-1。

参数描述：参数f表示防火墙句柄，参数callback表示一个回调函数，参数arg表示向回调函数传递的参数。

此函数的功能是注册回调函数callback，然后循环调用callback对每个防火墙规则进行操作。

（5）fw_t *fw_close(fw_t *f);

函数返回值：返回NULL。

参数描述：参数f表示防火墙句柄。

此函数的功能是关闭防火墙操作并销毁防火墙句柄，它与函数fw_open()相对应。

6.4.11 IP隧道函数

以下是IP隧道的数据结构的定义：

```
struct tun {
    int fd;             /*文件句柄*/
    intf_t intf;        /*接口*/
    struct ifreq ifr;
};
typedef struct tun tun_t;
```

（1）tun_t *tun_open(struct addr *src, struct addr *dst, int mtu);

函数返回值：函数执行成功就返回IP隧道句柄，失败就返回NULL。

参数描述：参数src表示源地址，参数dst表示目的地址，参数mtu表示MTU。

此函数的功能是打开IP隧道操作，进行初始化处理，并返回IP隧道句柄。

（2）int tun_fileno(tun_t *tun);

函数返回值：返回文件句柄。

参数描述：参数tun表示IP隧道句柄。

此函数的功能是返回IP隧道的文件句柄。

（3）const char *tun_name(tun_t *tun);

函数返回值：返回名字字符串。

参数描述：参数tun表示IP隧道句柄。

此函数的功能是返回IP隧道的名字。

（4）ssize_t tun_send(tun_t *tun, const void *buf, size_t size);

函数返回值：返回字节数。

参数描述：参数tun表示IP隧道句柄，参数buf表示一段内存数据，参数size表示大小。

此函数的功能是向由tun表示的IP隧道中写数据buf。

（5）ssize_t tun_recv(tun_t *tun, void *buf, size_t size);

函数返回值：返回字节数。

参数描述：参数tun表示IP隧道句柄，参数buf表示一段内存数据，参数size表示

大小。

此函数的功能是接收数据。

（6）tun_t　　*tun_close(tun_t *tun);

函数返回值：返回 NULL。

参数描述：参数 tun 表示 IP 隧道句柄。

此函数的功能是关闭 IP 隧道操作功能并销毁内存，它与函数 tun_open() 相对应。

6.5 Libdnet 开发包的使用

6.5.1 简单应用

使用 Libdnet 开发包的过程非常简单，这点从上面介绍的函数中就可以看到。由于 Libdnet 是一个综合的网络安全开发包，其中包括几个功能模块，每个功能模块都相对独立，并且其实现过程都一样，所以只要掌握了一个功能模块的使用方法，其他模块的使用就非常容易。

在这里，先举两个简单的例子，讲解 Libdnet 函数的使用方法及编译过程。

第一个例子的功能是发送一个 IP 数据包，此数据包由一段数据组成，可以任意填充，这里用一个字符串来表示，用十六进制进行显示。

这个例子用到了三个函数：ip_open()，ip_send()和 ip_close()。它们详细的定义在 6.4 节已经介绍过。

发送 IP 数据包的过程主要由三个步骤组成：首先利用 ip_open()函数打开 IP 协议操作，并对其进行初始化；然后使用 ip_send()函数发送数据；最后利用函数 ip_close()关闭 IP 协议操作功能。最重要的是有打开和关闭两个过程。在 Libdnet 中其他功能模块的实现都有这三个步骤。所以理解了这个程序，对于理解 Libdnet 功能函数的使用过程是非常有帮助的。

下面是该程序的源代码：（注意这里只是为了说明怎样使用 Libdnet 开发包，所以没有考虑程序的出错处理过程，以后所有的例子都是这样的。在实际的程序应用中都应该进行出错判断的处理过程，这样会使程序更加稳定、可靠且容易维护。）

```
#include "dnet.h"
/*必须包含此头文件，这是使用Libdnet开发包所必须的。*/
int main()
{
ip_t *handle;                          /*定义了一个IP协议句柄*/
u_char
ip_content[20]="\x45\x00\x00\x15\xce\xd2\x00\x00\xff\xc7\xc2\x87\x96\x0c\xd3\x0f\xc0\xa8\
x00\x03";                              /*定义了一段数据，注意此数据的形式是以十六机制的形式给出的。
                                         它的长度是20个字节*/
int ip_length=20;                      /*定义IP协议头的长度为20个字节*/
handle = ip_open();                    /*打开IP协议操作功能*/
ip_send(handle,                        /*IP句柄*/
ip_content,                            /*数据包内容*/
ip_length);                            /*长度*/
/*发送IP数据包*/
ip_close(handle);
```

```
/*关闭 rp 协议操作功能*/
}
```

在 Linux 平台下进行编译，其编译命令如下。

```
gcc -o test1 test1.c -ldnet
```

然后运行此程序，分析其运行的结果。使用 Tcpdump 工具的分析结果如下：

```
# tcpdump dst 192.168.0.3 -x
tcpdump: listening on eth0
20:14:23.584148 150.12.211.15 > 192.168.0.3: ip-proto-199 0
                        4500 0014 ced2 0000 ffe7 c288 960c d30f
                        c0a8 0003
```

其分析结果如图 6-7 所示。

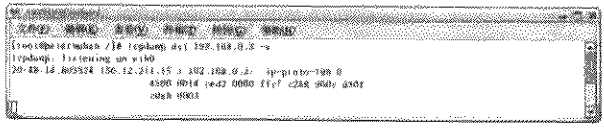

图 6-7　捕获 IP 数据包

可以看到，此例子用来构造一个 IP 数据包，然后发送出去。此 IP 协议数据包的内容与所构造的数据内容是一模一样的，证明程序构造和发送的 IP 数据包是正确的。

此程序在 Windows 平台下的编译过程如下。

（1）使用 Visual C++ 6.0 打开此文件，生成一个工作空间。

（2）对 Visual C++进行设置，单击 tools 菜单的子菜单 options，打开 options 对话框，在 directories 页面下进行设置。

（3）在 Include files 目录中添加 Libdnet 在 Windows 中的 include 路径。编者本机的 Libdnet 安装目录为：F:\libdnet-1.10-win32，所以应该添加的路径如下：

```
F:\libdnet-1.10-win32\include
```

（4）在 Library file 目录中添加 Libdnet 在 Windows 中的 lib 路径。应该添加的路径如下：

```
F:\libdnet-1.10-win32\lib
```

（5）在 Visual C++中添加 Libdnet 的静态库文件 dnet.lib。选择 project 菜单的子菜单 settings，打开对话框 Project Settings，在 Link 页面中的 Object/library Modules 选项中添加 dnet.lib 和 ws2_32.lib 两个静态库文件，如图 6-8 所示。

注意，在这里必须加上文件 ws2_32.lib，不然就会出错。

（6）对源文件进行编译。编译之后，如果出现以下错误：

```
MSVCRT.lib(MSVCRT.dll) : error LNK2005: _free already defined in LIBCD.lib(dbgheap.obj)
MSVCRT.lib(MSVCRT.dll) : error LNK2005: _calloc already defined in LIBCD.lib(dbgheap.obj)
```

就应该在 Visual C++中对运行时库选项进行修改。修改过程是，选择菜单 Project 的子菜单 Settings，打开对话框 Project Setting，在页面 C/C++中对 code generation 进行配置，修改 use run-time library 选项为 Debug MultiThreaded DLL。

如果未出现上述错误，则编译通过，然后运行。

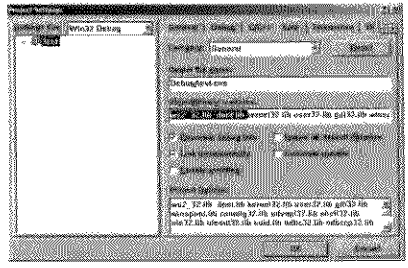

图 6-8　加入静态库文件

在 Windows 环境下可以使用软件 Ethereal 来分析所运行的结果。Ethereal 软件是一个非常著名的协议分析软件，可以从网站 http://www.ethereal.com/下载。最好下载最新的版本，在这里使用的是 0.10.10 版本。

其分析结果如图 6-9 所示。

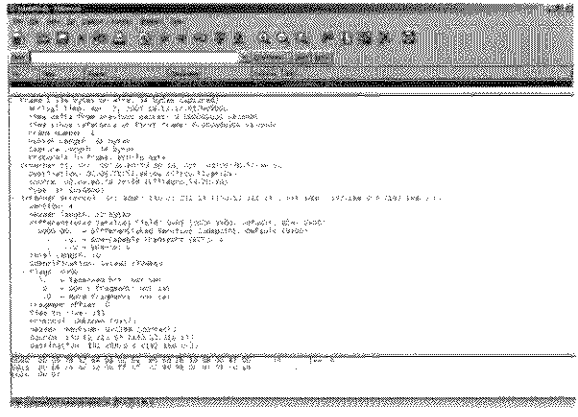

图 6-9　捕获 IP 数据包的分析结果

在图 6-9 中，最上面的窗口显示的是一个数据包的简要信息，中间的窗口显示的是此数据包的详细信息，最下面的窗口显示的是数据包的十六进制数和文本内容。

利用 Ethereal 的存储功能，得到此数据包的详细信息如下：

```
Frame 1 (34 bytes on wire, 34 bytes captured)
    Arrival Time: Apr  2, 2005 18:12:37.657909000
    Time delta from previous packet: 0.000000000 seconds
    Time since reference or first frame: 0.000000000 seconds
    Frame Number: 1
    Packet Length: 34 bytes
    Capture Length: 34 bytes
    Protocols in frame: eth:ip:data
```

```
Ethernet II, Src: 00:0a:e6:5d:2b:3d, Dst: 00:09:7b:52:e4:0a
    Destination: 00:09:7b:52:e4:0a (Cisco_52:e4:0a)
    Source: 00:0a:e6:5d:2b:3d (Elitegro_5d:2b:3d)
    Type: IP (0x0800)
Internet Protocol, Src Addr: 150.12.211.15 (150.12.211.15), Dst Addr: 192.168.0.3 (192.168.0.3)
    Version: 4
    Header length: 20 bytes
    Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
        0000 00.. = Differentiated Services Codepoint: Default (0x00)
        .... ..0. = ECN-Capable Transport (ECT): 0
        .... ...0 = ECN-CE: 0
    Total Length: 20
    Identification: 0xced2 (52946)
    Flags: 0x00
        0... = Reserved bit: Not set
        .0.. = Don't fragment: Not set
        ..0. = More fragments: Not set
    Fragment offset: 0
    Time to live: 255
    Protocol: Unknown (0xc7)
    Header checksum: 0xc288 (correct)
    Source: 150.12.211.15 (150.12.211.15)
    Destination: 192.168.0.3 (192.168.0.3)
```

第二个例子的功能是发送一个简单的以太网数据包。其实现的过程也是三个步骤，即利用 eth_open()，eth_send()和 eth_close()分别打开以太网协议操作、发送以太网数据和关闭以太网协议操作。

该例子的源代码如下：

```
#include "dnet.h"
/*Libdnet 头文件*/
int main()
{
    eth_t *handle;                                  /*以太网协议句柄*/
    u_char content[IP_LEN_MAX]="\x11\x11\x11\x11\x11\x11\x11\x11\x11\x11\x11\x11\x11\x11\x11\";
                                                    /*以太网协议数据，以十六进制数进行显示*/
    int length=14;                                  /*以太网协议的长度*/
    handle= eth_open("eth0");                       /*打开以太网协议操作*/
    eth_send(handle,                                /*以太网协议句柄*/
    content,                                        /*数据包内容*/
    length);                                        /*长度*/
    /*发送以太网协议数据*/
    eth_close(handle);                              /*关闭以太网协议操作*/
}
```

在 Linux 下进行编译，其编译命令为：

```
gcc -o test test.c -ldnet
```

使用 tcpdump 进行分析的结果如下：

```
[root@peterwuhan /]# tcpdump ether dst 11:11:11:11:11:11
tcpdump: listening on eth0
20:44:11.161198 11:11:11:11:11:11 (1:11:(1:11:11:11 1111 14:
```

其分析结果如图 6-10 所示。

图 6-10　捕获以太网协议数据的分析结果

6.5.2　构造和发送网络数据包

使用 Libdnet 可以构造任意形式的数据包。其中几种比较常用的网络数据包很容易构造，如 IP，TCP，UDP，ICMP 等，在这里着重分析一下这几种数据包的构造过程。

1、构造和发送 IP 数据包

在 6.5.1 节介绍了怎样构造一个简单的 IP 数据包，下面所举的例子，也是构造一个 IP 数据包，但是可以对其参数进行任意设置和配置。其程序源代码如下：

```
#include "dnet.h"
/*Libdnet 头文件*/
int main()
{
    struct ip_hdr *ip;                          /*IP 协议的数据结构*/
    ip_t *handle;                               /*IP 协议操作句柄*/
    u_char buf[IP_LEN_MAX];                     /*缓存，用来存储 IP 协议数据*/
    int length=20;                              /*IP 协议数据长度*/
    struct addr addr;                           /*地址*/
    char *destination_ip="192.168.0.3";         /*IP 目的地址*/
    char *source_ip="192.168.0.2";              /*IP 源地址*/
    ip = (struct ip_hdr *)buf;                  /*把 buf 表示的缓存数据转换为 IP 的协议格式*/
    ip->ip_hl = 5;                              /*IP 首部长度*/
    ip->ip_v = 4;                               /*IP 版本号*/
    ip->ip_tos = 0;                             /*IP 的 TOS*/
    ip->ip_id = 0x11;                           /*IP 标识符号*/
    ip->ip_off = 0;                             /*偏移量*/
    ip->ip_ttl = IP_TTL_MAX;                    /*IP 生存时间*/
        ip->ip_p = 6;                           /*IP 上层协议类型*/
        ip->ip_sum = 0;                         /*校验和*/
    addr_aton(source_ip, &addr);                /*把源 IP 地址的字符串格式转换为二进制形式*/
    ip->ip_src= addr;addr_ip;                   /*源 IP 地址*/
```

```
    addr_aton(destination_ip, &addr);                    /*把目的 IP 地址的字符串格式转换为二进制形式*/
    ip->ip_dst= addr.addr_ip;                            /*目的 IP 地址*/
    ip->ip_len = htons(length);                          /*长度*/
    ip_checksum(buf, length);                            /*计算校验和*/
    handle= ip_open();                                   /*打开 IP 协议操作，返回 IP 协议操作句柄*/
    ip_send(handle, buf, length);                        /*发送 IP 协议数据*/
    ip_close(handle);                                    /*关闭 IP 协议操作*/
}
```

此程序在 Linux 下的编译过程如下。

（1）使用编译命令。

```
gcc -o test test.c -ldnet
```

生成可执行文件 test，然后运行此文件。

（2）使用 Tcpdump 进行分析结果如下：

```
# tcpdump dst 192.168.0.3 -x
tcpdump: listening on eth0
21:09:01.043485 192.168.0.2 > 192.168.0.3: [|tcp]
                        4500 0014 1100 0000 ff06 296e c0a8 0002
                        c0a8 0003
```

其分析结果如图 6-11 所示。可以得出结论，此程序构造的数据包和捕获到的数据包是一样的，证明构造的数据包是正确的。

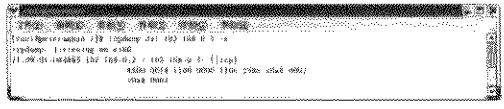

图 6-11　捕获 IP 数据包的分析结果

同样，此程序也可以在 Windows 平台下使用 Visaul C++进行编译。

（3）编译运行后，使用 Ethereal 工具进行捕获，如图 6-12 所示。

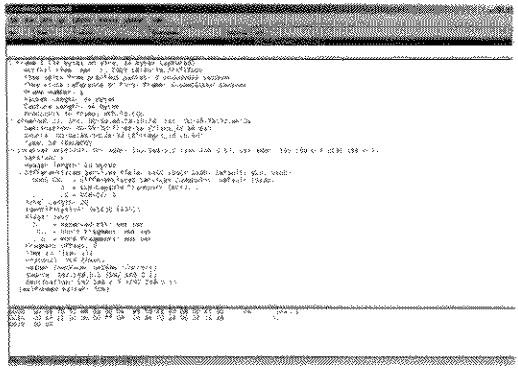

图 6-12　捕获 IP 数据包的分析结果

利用 Ethereal 自动导出此数据包的详细信息，具体如下：

```
Frame 1 (34 bytes on wire, 34 bytes captured)
    Arrival Time: Apr  2, 2005 18:30:58.775737000
    Time delta from previous packet: 0.000000000 seconds
    Time since reference or first frame: 0.000000000 seconds
    Frame Number: 1
    Packet Length: 34 bytes
    Capture Length: 34 bytes
    Protocols in frame: eth:ip:tcp
Ethernet II, Src: 00:0a:e6:5d:2b:3d, Dst: 00:09:7b:52:e4:0a
    Destination: 00:09:7b:52:e4:0a (Cisco_52:e4:0a)
    Source: 00:0a:e6:5d:2b:3d (Elitegro_5d:2b:3d)
    Type: IP (0x0800)
Internet Protocol, Src Addr: 192.168.0.2 (192.168.0.2), Dst Addr: 192.168.0.3 (192.168.0.3)
    Version: 4
    Header length: 20 bytes
    Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
        0000 00.. = Differentiated Services Codepoint: Default (0x00)
        .... ..0. = ECN-Capable Transport (ECT): 0
        .... ...0 = ECN-CE: 0
    Total Length: 20
    Identification: 0x1100 (4352)
    Flags: 0x00
        0... = Reserved bit: Not set
        .0.. = Don't fragment: Not set
        ..0. = More fragments: Not set
    Fragment offset: 0
    Time to live: 255
    Protocol: TCP (0x06)
    Header checksum: 0x298e (correct)
    Source: 192.168.0.2 (192.168.0.2)
    Destination: 192.168.0.3 (192.168.0.3)
[Malformed Packet: TCP]
0000  00 09 7b 52 e4 0a 00 0a e6 5d 2b 3d 08 00 45 00   ..{R.....l+=..E.
0010  00 14 11 00 00 00 ff 06 29 8e c0 a8 00 02 c0 a8   ........).......
0020  00 03                                             ..
```

2. 构造和发送以太网数据包

在 Libdnet 中构造的数据包有两种类型：原始套接字 IP 包和以太网包。上一个例子介绍的是原始套接字 IP 包的构造过程，也可以构造基于 IP 协议的数据包，在后面有介绍。基于以太网协议的数据包是从链路层进行构造的，所有基于以太网的数据包都可以进行构造，如 ARP 数据包等。

下面是构造一个纯以太网数据包的例子，其源代码如下：

```
#include "dnet.h"
    /*Libdnet 头文件*/
int main()
{
    eth_t *handle;                          /*以太网协议操作句柄*/
    struct eth_hdr *eth;                    /*以太网协议结构*/
    struct addr addr;                       /*地址*/
    u_char content[ETH_LEN_MAX];            /*存放以太网协议数据*/
    int length;                             /*长度*/
    length=14;
    eth = (struct eth_hdr *)content;        /*构造以太网协议数据*/
    memset(eth, 0, sizeof(*eth));
    eth->eth_type = htons(ETH_TYPE_IP);     /*以太网协议类型*/
    addr_aton("11:11:11:11:11:11", &addr);  /*把源以太网地址转换为二进制形式*/
    memcpy(&eth->eth_src, &addr.addr_eth, ETH_ADDR_LEN);
                                            /*以太网源地址*/
    addr_aton("22:22:22:22:22:22", &addr);  /*把目的以太网地址转换为二进制形式*/
    memcpy(&eth->eth_dst, &addr.addr_eth, ETH_ADDR_LEN);
                                            /*以太网目的地址*/
    handle= eth_open("eth0");               /*打开以太网协议操作，接口名字为 eth0*/
    eth_send(handle, content, length);      /*发送以太网协议数据*/
    eth_close(handle);                      /*关闭以太网协议操作*/
}
```

在 Linux 下的编译命令如下：

```
gcc -o sendethernet sendethernet.c -ldnet
```

使用 Tcpdump 进行分析，结果如下：

```
# tcpdump ether dst 22:22:22:22:22:22
tcpdump: listening on eth0
21:28:07.017836 [|ip]
```

其分析结果如图 6-13 所示。

图 6-13　捕获以太网数据包的结果

3. 构造和发送 ICMP 数据包

ICMP 协议是基于 IP 协议的，所以构造和发送 ICMP 数据包时是基于原始套接字 IP 来进行构造和发送的，其代码如下。

```
#include "dnet.h"                           /*Libdnet 头文件*/
int main()
{
```

```
    struct ip_hdr *ip;                          /*IP 协议头的数据结构*/
    struct icmp_hdr *icmp;                      /*ICMP 协议的数据结构*/
    ip_t *handle;                               /*IP 协议操作句柄*/
    u_char content[IP_LEN_MAX];                 /*ICMP 数据的内容*/
    int length=20+8;                            /*ICMP 数据包的总长度，应该是 IP 协议头加 ICMP 协议*/
    struct addr addr;                           /*地址*/
    char *destination_ip="192.168.0.3";         /*目的 IP 地址*/
    char *source_ip="192.168.0.2";              /*源 IP 地址*/
    ip = (struct ip_hdr *)content;              /*构造 IP 协议头*/
    ip->ip_hl = 5;                              /*IP 长度*/
    ip->ip_v = 4;                               /*IP 版本号*/
    ip->ip_tos = 0;                             /*服务质量*/
    ip->ip_id = 0x11;                           /*IP 标识*/
    ip->ip_off = 0;                             /*偏移量*/
    ip->ip_ttl = IP_TTL_MAX;                    /*TTL 生存时间*/
    ip->ip_p = 1;                               /*IP 上层协议类型，1 代表 ICMP 协议*/
    ip->ip_sum = 0;                             /*校验和*/
    addr_aton(source_ip, &addr);                /*把源 IP 地址的字符串形式转换为二进制形式*/
    ip->ip_src= addr.addr_ip;                   /*源 IP 地址*/
    addr_aton(destination_ip, &addr);           /*把目的 IP 地址的字符串形式转换为二进制形式*/
    ip->ip_dst= addr.addr_ip;                   /*目的 IP 地址*/
    ip->ip_len = htons(length);                 /*长度*/
    icmp = (struct icmp_hdr *)(content+20);     /*构造 ICMP 协议数据，应该跳过 IP 协议数据，所以应
                                                  该加 20 字节数*/
    memset(icmp, 0, sizeof(*icmp));
    icmp->icmp_type = ICMP_ECHO;                /*ICMP 类型，此时为 ICMP_ECHO，表示回显请求 ICMP
                                                  数据包，如果要构造其他类型的 ICMP 数据包，就需要
                                                  修改这里的类型*/
    icmp->icmp_code = 0;                        /*ICMP 代码，它跟 ICMP 类型一起决定 ICMP 数据的种类*/
    ip_checksum(content, length);               /*计算校验和*/
    handle= ip_open();                          /*打开 IP 协议操作*/
    ip_send(handle, content, length);           /*发送 IP 协议数据包，注意在这里还是调用同一个函数
                                                  来发送 IP 协议数据包，但此时 IP 的负载是一个 ICMP
                                                  协议，所以实际发送的是一个 ICMP 数据包，因为
                                                  ICMP 协议是基于 IP 协议的*/
    ip_close(handle);                           /*关闭 IP 协议操作句柄*/
}
```

分析此程序可知，构造和发送 ICMP 数据包时，是基于原始套接字 IP 种类进行构造的，因为 ICMP 协议是基于 IP 协议的。发送 ICMP 数据包的过程与发送纯 IP 协议数据的过程是一样的，只不过其内容是 ICMP 协议数据而已。

在 Linux 下进行编译，其编译命令如下：

```
gcc -o sendicmp sendicmp.c -ldnet
```

运行此程序，使用工具 Tcpdump 捕获 ICMP 数据包，得到如下结果：

```
[root@peterwuhan root]# tcpdump icmp -Xx
tcpdump: listening on eth0
12:49:27.928191 192.168.0.2 > 192.168.0.3: icmp: echo request
0x0000   4500 001c 1100 0000 ff01 298b c0a8 0002        E.........)).....
0x0010   c0a8 0003 0800 0000 0000 0000                  ............
```

其分析的结果如图 6-14 所示。

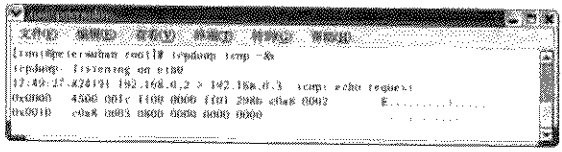

图 6-14　Tcpdump 捕获 ICMP 数据包的结果

可见，所构造的是一个 ICMP 回显请求数据包，其源 IP 地址为 192.168.0.2，目的 IP 地址为 192.168.0.3。

在 Linux 下也可以使用协议分析软件 Ethereal 来进行分析，其界面更方便，显示更清楚。其分析的结果如图 6-15 所示。

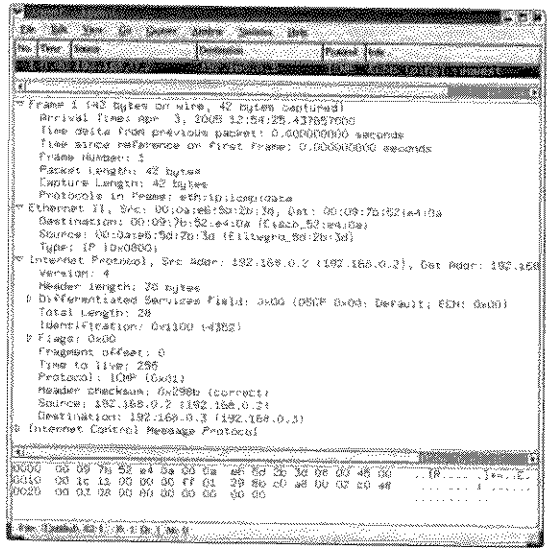

图 6-15　Ethereal 捕获 ICMP 数据包

使用Ethereal自动导出这个数据包的详细信息，具体如下：

```
Frame 1 (42 bytes on wire, 42 bytes captured)
    Arrival Time: Apr  3, 2005 12:54:25.437657000
    Time delta from previous packet: 0.000000000 seconds
```

```
    Time since reference or first frame: 0.000000000 seconds
    Frame Number: 1
    Packet Length: 42 bytes
    Capture Length: 42 bytes
    Protocols in frame: eth:ip:icmp:data
Ethernet II, Src: 00:0a:e6:5d:2b:3d, Dst: 00:09:7b:52:e4:0a
    Destination: 00:09:7b:52:e4:0a (Cisco_52:e4:0a)
    Source: 00:0a:e6:5d:2b:3d (Elitegro_5d:2b:3d)
    Type: IP (0x0800)
Internet Protocol, Src Addr: 192.168.0.2 (192.168.0.2), Dst Addr: 192.168.0.3 (192.168.0.3)
    Version: 4
    Header length: 20 bytes
    Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
        0000 00.. = Differentiated Services Codepoint: Default (0x00)
        .... ..0. = ECN-Capable Transport (ECT): 0
        .... ...0 = ECN-CE: 0
    Total Length: 28
    Identification: 0x1100 (4352)
    Flags: 0x00
        0... = Reserved bit: Not set
        .0.. = Don't fragment: Not set
        ..0. = More fragments: Not set
    Fragment offset: 0
    Time to live: 255
    Protocol: ICMP (0x01)
    Header checksum: 0x298b (correct)
    Source: 192.168.0.2 (192.168.0.2)
    Destination: 192.168.0.3 (192.168.0.3)
Internet Control Message Protocol
    Type: 8 (Echo (ping) request)
    Code: 0
    Checksum: 0x0000 (incorrect, should be 0xf7ff)
    Identifier: 0x0000
    Sequence number: 0x0000
0000  00 09 7b 52 e4 0a 00 0a e6 5d 2b 3d 08 00 45 00   ..{R.....]+=..E.
0010  00 1c 11 00 00 00 ff 01 29 8b c0 a8 00 02 c0 a8   ........).......
0020  00 03 08 00 00 00 00 00 00 00                     ..........
```

4. 构造和发送 TCP 数据包

发送 TCP 数据包的过程与发送 ICMP 数据包的过程相似，都使用基于原始套接字 IP 类型来进行构造和发送。其源代码如下：

```
#include "dnet.h"                          /*Libdnet 头文件*/
int main()
```

```
{
    struct ip_hdr *ip;                          /*IP 协议头的数据结构*/
    struct tcp_hdr *tcp;                        /*TCP 协议头的数据结构*/
    ip_t *handle;                               /*IP 协议操作句柄*/
    u_char content[IP_LEN_MAX];                 /*TCP 数据包内容，包括 IP 协议头内容*/
    int length=20+20;                           /*整个长度，包括 IP 协议头长度和 TCP 协议头长度*/
    struct addr addr;                           /*地址*/
    char *destination_ip="192.168.0.3";         /*目的 IP 地址*/
    char *source_ip="192.168.0.2";              /*源 IP 地址*/
    ip = (struct ip_hdr *)content;              /*构造 IP 协议头*/
    ip->ip_hl = 5;                              /*IP 长度*/
    ip->ip_v = 4;                               /*IP 版本号*/
    ip->ip_tos = 0;                             /*服务质量*/
    ip->ip_id = 0x11;                           /*IP 标识*/
    ip->ip_off = 0;                             /*IP 偏移量*/
    ip->ip_ttl = IP_TTL_MAX;                    /*TTL 生存时间*/
    ip->ip_p = 6;                               /*IP 上层协议类型，6 代表 TCP 协议*/
    ip->ip_sum = 0;                             /*计算校验和*/
    addr_aton(source_ip, &addr);                /*把源 IP 地址的字符串形式转换为二进制形式*/
    ip->ip_src= addr.addr_ip;                   /*源 IP 地址*/
    addr_aton(destination_ip, &addr);           /*把目的 IP 地址的字符串形式转换为二进制形式*/
    ip->ip_dst= addr.addr_ip;                   /*目的 IP 地址*/
    ip->ip_len = htons(length);                 /*长度*/
    tcp = (struct tcp_hdr *)(content+20);       /*构造 TCP 协议数据，跳过 IP 协议数据，所以应该加 20*/
    memset(tcp, 0, sizeof(*tcp));
    tcp->th_sport = htons(81);                  /*源端口号*/
    tcp->th_dport = htons(91);                  /*目的端口号*/
    tcp->th_seq = htonl(111);                   /*序列号*/
    tcp->th_ack = 0;                            /*确认号*/
    tcp->th_off = 5;                            /*偏移*/
    tcp->th_flags = TH_SYN;                     /*控制标记*/
    tcp->th_win = TCP_WIN_MAX;                  /*窗口大小*/
    tcp->th_urp = 0;                            /*紧急指针*/
    ip_checksum(content, length);               /*计算机校验和*/
    handle= ip_open();                          /*打开 IP 协议操作*/
    ip_send(handle, content, length);           /*发送 TCP 数据包，它是基于原始套接字 IP 进行构造的*/
    ip_close(handle);                           /*关闭 IP 协议操作*/
}
```

此程序的功能是构造和发送一个 TCP 数据包，在 Linux 下进行编译，其编译过程如下：

```
gcc -o send_tcp send_tcp_code.c -ldnet
```

运行此程序，然后使用 Tcpdump 工具对结果进行分析，如图 6-16 所示。

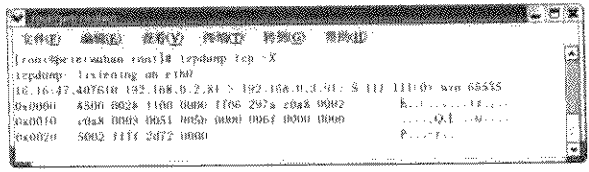

图 6-16　使用 Tcpdump 进行分析

也可以使用 Ethereal 工具对发送的 TCP 数据包进行捕获，其捕获结果如图 6-17 所示：

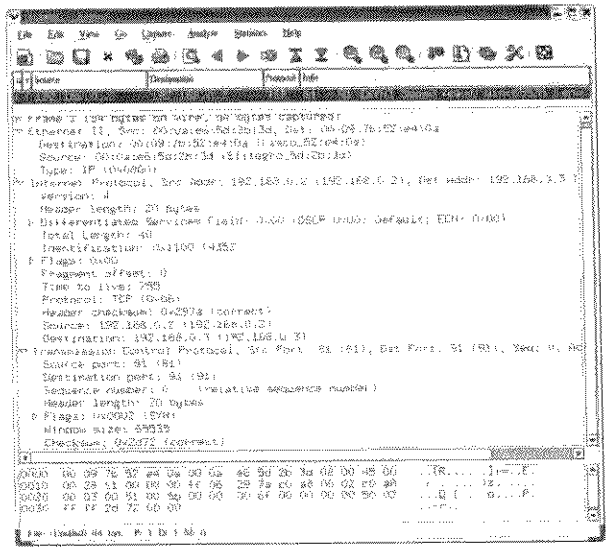

图 6-17　用 Ethereal 捕获所构造的 TCP 数据包

使用 Ethereal 的导出功能，输出此数据包的详细信息如下：

```
Frame 1 (54 bytes on wire, 54 bytes captured)
    Arrival Time: Apr  4, 2005 16:12:56.77[illegible]000
    Time delta from previous packet: 0.000000000 seconds
    Time since reference or first frame: 0.000000000 seconds
    Frame Number: 1
    Packet Length: 54 bytes
    Capture Length: 54 bytes
    Protocols in frame: eth:ip:tcp
Ethernet II, Src: 00:da:e6:5d:3b:3d, Dst: 00:09:7b:52:e4:0a
    Destination: 00:09:7b:52:e4:0a (Cisco_52:e4:0a)
    Source: 00:da:e6:5d:3b:3d (Elitegro_5d:3b:3d)
    Type: IP (0x0800)
```

```
Internet Protocol, Src Addr: 192.168.0.2 (192.168.0.2), Dst Addr: 192.168.0.3 (192.168.0.3)
    Version: 4
    Header length: 20 bytes
    Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
        0000 00.. = Differentiated Services Codepoint: Default (0x00)
        .... ..0. = ECN-Capable Transport (ECT): 0
        .... ...0 = ECN-CE: 0
    Total Length: 40
    Identification: 0x1100 (4352)
    Flags: 0x00
        0... = Reserved bit: Not set
        .0.. = Don't fragment: Not set
        ..0. = More fragments: Not set
    Fragment offset: 0
    Time to live: 255
    Protocol: TCP (0x06)
    Header checksum: 0x297a (correct)
    Source: 192.168.0.2 (192.168.0.2)
    Destination: 192.168.0.3 (192.168.0.3)
Transmission Control Protocol, Src Port: 81 (81), Dst Port: 91 (91), Seq: 0, Ack: 0, Len: 0
    Source port: 81 (81)
    Destination port: 91 (91)
    Sequence number: 0    (relative sequence number)
    Header length: 20 bytes
    Flags: 0x0002 (SYN)
        0... .... = Congestion Window Reduced (CWR): Not set
        .0.. .... = ECN-Echo: Not set
        ..0. .... = Urgent: Not set
        ...0 .... = Acknowledgment: Not set
        .... 0... = Push: Not set
        .... .0.. = Reset: Not set
        .... ..1. = Syn: Set
        .... ...0 = Fin: Not set
    Window size: 65535
        Checksum: 0x2d72 (correct)
0000  00 09 7b 52 e4 0a 00 0a e6 5d 2b 3d 08 00 45 00   ..{R...].+=..E.
0010  00 28 11 00 00 00 ff 06 29 7a c0 a8 00 02 c0 a8   .(......)z......
0020  00 03 00 51 00 5b 00 00 00 6f 00 00 00 00 50 02   ...Q.[...o....P.
0030  ff ff 2d 72 00 00                                 ..-r..
```

可见，其捕获的数据包的内容与程序构造的数据包内容是一模一样的，说明程序构造的数据包完全正确。

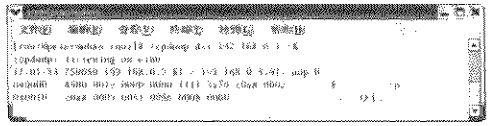

图 6-18　Tcpdump 的分析结果

使用协议分析软件 Ethereal 进行协议分析，其捕获数据包的结果如图 6-19 所示。

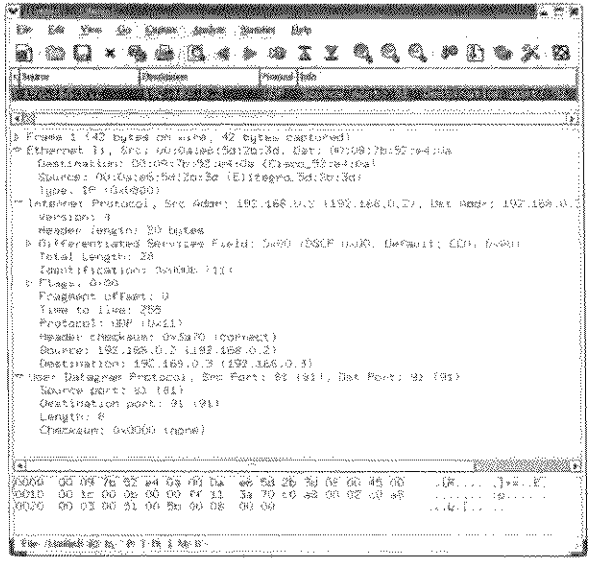

图 6-19　使用 Ethereal 捕获的 UDP 数据包

使用 Ethereal 导出此数据包的详细信息，具体如下。

```
Frame 1 (42 bytes on wire, 42 bytes captured)
    Arrival Time: Apr  4, 2005 17:01:33.750850000
    Time delta from previous packet: 0.000000000 seconds
    Time since reference or first frame: 0.000000000 seconds
    Frame Number: 1
    Packet Length: 42 bytes
    Capture Length: 42 bytes
    Protocols in frame: eth:ip:udp:data
Ethernet II, Src: 00:0a:e6:5d:2b:3d, Dst: 00:09:7b:52:e4:0a
    Destination: 00:09:7b:52:e4:0a (Cisco_52:e4:0a)
    Source: 00:0a:e6:5d:2b:3d (Elitegro_5d:2b:3d)
    Type: IP (0x0800)
Internet Protocol, Src Addr: 192.168.0.2 (192.168.0.2), Dst Addr: 192.168.0.3 (192.168.0.3)
```

```
    Version: 4
    Header length: 20 bytes
    Differentiated Services Field: 0x00 (DSCP 0x00: Default; ECN: 0x00)
        0000 00.. = Differentiated Services Codepoint: Default (0x00)
        .... ..0. = ECN-Capable Transport (ECT): 0
        .... ...0 = ECN-CE: 0
    Total Length: 28
    Identification: 0x000b (11)
    Flags: 0x00
        0... = Reserved bit: Not set
        .0.. = Don't fragment: Not set
        ..0. = More fragments: Not set
    Fragment offset: 0
    Time to live: 255
    Protocol: UDP (0x11)
    Header checksum: 0x3a70 (correct)
    Source: 192.168.0.2 (192.168.0.2)
    Destination: 192.168.0.3 (192.168.0.3)
User Datagram Protocol, Src Port: 81 (81), Dst Port: 91 (91)
    Source port: 81 (81)
    Destination port: 91 (91)
    Length: 8
    Checksum: 0x0000 (none)
0000  00 09 7b 52 e4 0a 00 0a e6 5d 2b 3d 08 00 45 00   ..{R.....}+=..E.
0010  00 1c 00 0b 00 00 ff 11 3a 70 c0 a8 00 02 c0 a8   ........:p......
0020  00 03 00 51 00 5b 00 08 00 00                     ...Q.[....
```

以上是由 Ethereal 自动导出的结果。此数据包包括三部分，一是链路层以太网协议，二是网络层 IP 协议部分，三是传输层 UDP 协议部分。每部分的协议字段都显示得非常清楚，把它与程序构造的数据包进行对比，发现其捕获的数据包与构造的数据包是一样的。说明构造和发送的过程中没有出现错误。

6. 构造和发送 ARP 数据包

ARP 数据包应使用基于以太网协议类型来构造和发送，而不应使用基于原始套接字 IP 类型来进行构造，所以它的构造过程与 ICMP，TCP，UDP 的构造过程是不一样的。

构造 ARP 数据包的例子如下：

```
#include "dnet.h"                                   /*Libdnet 头文件*/
int main( )
{
    struct arp_hdr *arp;                            /*ARP 协议头数据结构*/
    struct arp_ethip *arp_ethernet_ip;              /*ARP 请求的数据结构*/
    eth_t *handle;                                  /*以太网协议操作句柄*/
    struct eth_hdr *ethernet;                       /*以太网协议格式*/
    struct addr addr;                               /*地址*/
```

```
tcpdump: listening on eth0
12:15:23.184851 arp who-has 192.168.0.3 (22:22:22:22:22:22) tell 192.168.0.2
                    0001 0800 0604 0001 1111 1111 1111 c0a8
                    0002 2222 2222 2222 c0a8 0003
```

其运行结果如图 6-20 所示。

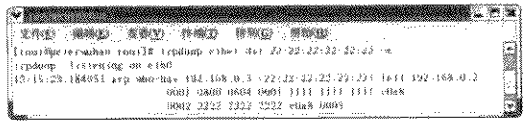

图 6-20　使用 Tcpdump 分析 ARP 的结果

使用协议分析软件 Ethereal 进行分析的结果如图 6-21 所示。

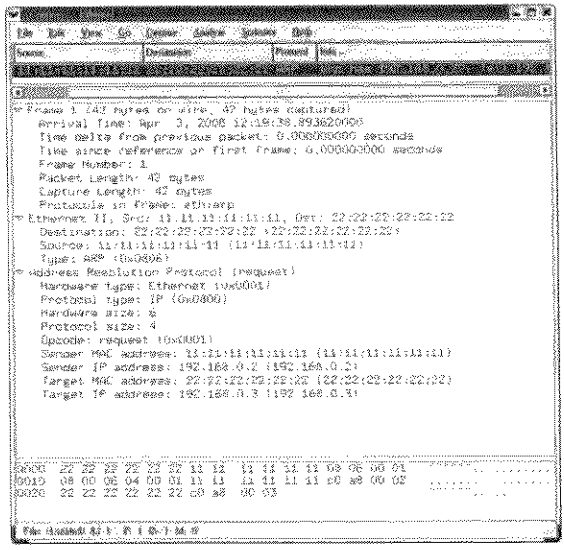

图 6-21　使用 Ethereal 捕获 ARP 数据包

将此数据包的各个协议的详细信息导出如下：

```
Frame 1 (42 bytes on wire, 42 bytes captured)
    Arrival Time: Apr  3, 2005 12:19:38.893620000
    Time delta from previous packet: 0.000000000 seconds
    Time since reference or first frame: 0.000000000 seconds
    Frame Number: 1
    Packet Length: 42 bytes
    Capture Length: 42 bytes
    Protocols in frame: eth:arp
```

```
Ethernet II, Src: 11:11:11:11:11:11, Dst: 22:22:22:22:22:22
    Destination: 22:22:22:22:22:22 (22:22:22:22:22:22)
    Source: 11:11:11:11:11:11 (11:11:11:11:11:11)
    Type: ARP (0x0806)
Address Resolution Protocol (request)
    Hardware type: Ethernet (0x0001)
    Protocol type: IP (0x0800)
    Hardware size: 6
    Protocol size: 4
    Opcode: request (0x0001)
    Sender MAC address: 11:11:11:11:11:11 (11:11:11:11:11:11)
    Sender IP address: 192.168.0.2 (192.168.0.2)
    Target MAC address: 22:22:22:22:22:22 (22:22:22:22:22:22)
    Target IP address: 192.168.0.3 (192.168.0.3)
0000  22 22 22 22 22 22 11 11 11 11 11 11 08 06 00 01   """"""..........
0010  08 00 06 04 00 01 11 11 11 11 11 11 c0 a8 00 02   ................
0020  22 22 22 22 22 22 c0 a8 00 03                     """"""....
```

此数据包是一个ARP请求数据包。可以看到，其以太网源地址是11:11:11:11:11:11，目的地址为22:22:22:22:22:22，源IP地址为192.168.0.2，目的地址为192.168.0.3，与程序构造的ARP协议数据是一致的。

6.5.3 ARP 缓存操作

ARP 缓存存放了 IP 地址到以太网地址之间的映射记录，有了 ARP 缓存，ARP 协议运行的效率就会提高。Libdnet 提供了对 ARP 缓存操作的功能，它可以查看 ARP 缓存的内容，添加 ARP 缓存记录，也可以删除 ARP 缓存记录。

下面通过举例进行讲解，其代码如下：

```
#include "dnet.h"                                                    /*Libdnet 头文件*/
/*下面定义的函数是一个回调函数，在函数 arp_loop()函数中被调用，它的功能就是对 ARP 记录进行操作，在这里
是显示 ARP 记录*/
int show_arp_cache(const struct arp_entry *object, void *p)
{
    printf("IP %s at MAC %s\n ",
    addr_ntoa (&object->arp_pa), addr_ntoa(&object->arp_ha));        /*显示 IP 地址和以太网地址对*/
    return (0);
}
int main( )
{
    struct arp_entry arp_object;                                     /*ARP 缓存的数据结构的变量*/
    arp_t *arp_handle;                                               /*ARP 缓存操作句柄*/
    arp_handle = arp_open();                                         /*打开 ARP 缓存操作*/
    printf("show the arp cache\n");
    arp_loop(arp_handle,show_arp_cache,NULL);
```

```
/*作用回调函数 show_arp_cache，作用是循环调用回调函数对每个 ARP 缓存记录进行操作，此处是显示每个
ARP 缓存记录。*/
printf("Add a arp \n");
addr_pton("192.168.0.4",&arp_object.arp_pa);    /*把一个 IP 地址的字符串形式转化为二进制形式*/
addr_pton("11:11:11:11:11:11",&arp_object.arp_ha);  /*把一个以太网地址的字符串形式转换为二进制形式*/
arp_add(arp_handle,&arp_object);                /*增加一个 ARP 缓存记录*/
printf("a arp added\n");
addr_pton("192.168.0.5",&arp_object.arp_pa);    /*把一个 IP 地址的字符串形式转化为二进制形式*/
addr_pton("22:22:22:22:22:22",&arp_object.arp_ha);  /*把一个以太网地址的字符串形式转换为二进
                                                    制形式*/
arp_add(arp_handle,&arp_object);                /*增加一个 ARP 缓存记录*/
printf("a arp added\n");
printf("show the arp cache again.\n");
arp_loop(arp_handle,show_arp_cache,NULL);       /*显示每个 ARP 缓存记录*/
printf("Delete a arp \n");
addr_pton("192.168.0.4",&arp_object.arp_pa);    /*把一个 IP 地址的字符串形式转化为二进制形式*/
arp_delete(arp_handle,&arp_object);             /*删除一个 ARP 缓存记录*/
printf("a arp deleted\n");
printf("show the arp cache again\n");
arp_loop(arp_handle,show_arp_cache,NULL);       /*显示每个 ARP 缓存记录*/
arp_close(arp_handle);                          /*关闭 ARP 缓存操作*/
}
```

此程序的功能是输出所有的 ARP 缓存信息，并演示了增加和删除 ARP 缓存记录的功能。

在 Linux 下进行编译，其编译命令如下：

```
gcc -o arpop arpop.c -ldnet
```

运行此程序，其结果如图 6-22 所示。

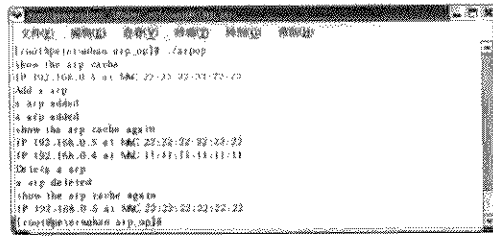

图 6-22　ARP 缓存操作的运行结果

把此程序原封不动地在 Windows 平台下进行编译。使用 Visual C++ 6.0 打开此文件，生成一个工程。同时，要对 Visual C++进行配置，添加 Libdnet 的包含文件路径和静态链接库路径，然后再添加静态库文件，包括 dnet.lib，wsock32.lib 和 iphlpapi.lib 三个文件，如图 6-23 所示。

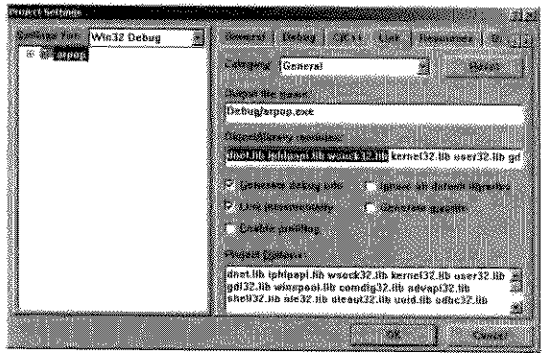

图 6-23　添加静态链接库 dnet.lib，iphlpapi.lib 和 wsock32.lib

修改 Use run-time library 选项为 Debug Multithreaded DLL，如图 6-24 所示。

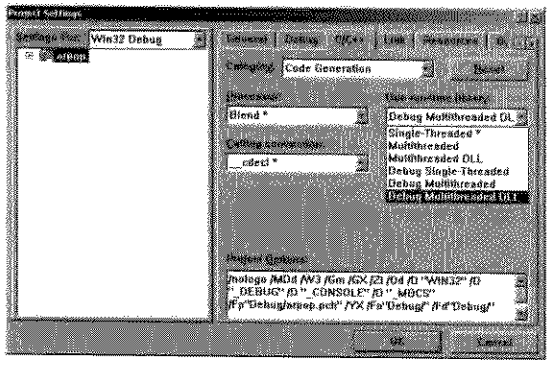

图 6-24　修改 Use run-time library

设置完后，编译、运行此程序，其运行结果如图 6-25 所示。

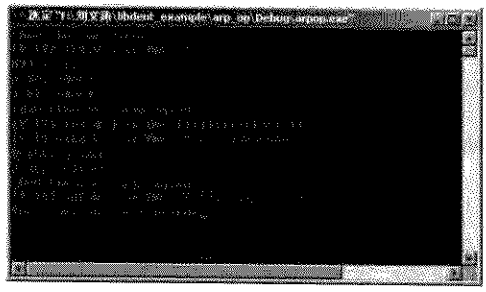

图 6-25　ARP 缓存操作的运行结果

6.5.4 网络接口操作

网络接口操作是很多网络应用程序要涉及的操作，在 Libdnet 中实现了对网络接口操作的功能，它可以显示每个网络接口的信息，修改网络接口的参数。例如，可以使网络接口激活，也可以使网络接口失效；可以使网络接口支持广播，也可以使它不支持广播；可以使网络接口支持 ARP 协议，也可以使它不支持 ARP 协议。

下面通过举一个例子进行讲解，其源代码如下。

```
#include "dnet.h"
/*Libdnet 头文件*/
/*下面的函数是一个回调函数，其在函数 intf_loop () 中调用。它的功能是对每个网络接口进行操作，在这里是输出
每个网络接口的信息*/
int show_interface_content(const struct intf_entry *object, void *p)
{
    int number;
    printf("network interface name is :");
    printf("%s\n",object->intf_name);                    /*输出网络接口的名字*/
    printf("flags: %d\n", object->intf_flags);           /*输出网络接口参数标志*/
    printf("flags including:\n");
    if (object->intf_flags & INTF_FLAG_UP)               /*判断是否是激活状态，UP*/
    printf("UP\n");
    if (object->intf_flags & INTF_FLAG_LOOPBACK)         /*判断是否是回路*/
    printf("LOOPBACK\n");
    if (object->intf_flags & INTF_FLAG_POINTOPOINT)      /*判断是否支持点对点协议*/
    printf("POINTOPOINT\n");
    if (object->intf_flags & INTF_FLAG_NOARP)            /*判断是否不支持 ARP 协议*/
    printf("NOARP\n");
    if (object->intf_flags & INTF_FLAG_BROADCAST)        /*判断是否支持广播*/
    printf("BROADCAST\n");
    if (object->intf_flags & INTF_FLAG_MULTICAST)        /*判断是否支持多播*/
    printf("MULTICAST\n");
    if (object->intf_mtu != 0)
    printf("mtu: %d\n", object->intf_mtu);               /*输出 MTU 值*/
    if (object->intf_addr.addr_type == ADDR_TYPE_IP) {
    if (object->intf_dst_addr.addr_type == ADDR_TYPE_IP) {
        printf("address:%s,desination:%s\n",addr_ntoa(&object->intf_addr),
        addr_ntoa(&object->intf_dst_addr));
        /*输出源地址和目的地址*/
        } else
        printf("address:%s\n", addr_ntoa(&object->intf_addr));
        /*输出源地址*/
}
if (object->intf_link_addr.addr_type == ADDR_TYPE_ETH)
```

```
printf("show the network interface:\n");
intf_loop(handle,show_interface_content,NULL);                    /*显示所有网络接口的信息*/
printf("set the network interface can't use ARP protocol :\n");
entry->intf_flags |= INTF_FLAG_NOARP;      /*设置网络接口参数为不支持 ARP 协议*/
intf_set(handle,entry);                    /*设置指定的网络接口不支持 ARP 协议*/
printf("show the network interface:\n");
intf_loop(handle,show_interface_content,NULL);                    /*显示所有的网络接口的信息*/
printf("set the network interface can use ARP protocol :\n");
entry->intf_flags &= ~INTF_FLAG_NOARP;     /*设置网络接口参数为不支持 ARP 协议*/
intf_set(handle, entry);                   /*设置特定的网络接口不支持 ARP 协议*/
printf("show the network interface:\n");
intf_loop(handle,show_interface_content,NULL);                    /*显示所有的网络接口信息*/
printf("set the network interface alias:\n");
interface_address =&entry->intf_alias_addrs[entry->intf_alias_num++]; /*网络接口的别名*/
addr_pton("192.168.0.6", interface_address);   /*把 IP 地址的字符串形式转换为二进制形式*/
intf_set(handle, entry);                       /*设置网络接口的别名为"192.168.0.6" */
printf("show the network interface:\n");
intf_loop(handle,show_interface_content,NULL); /*显示所有网络接口的信息*/
printf("set the network interface destination:\n");
interface_address = &entry->intf_dst_addr;     /*目的地址*/
addr_pton("192.168.0.7", interface_address);   /*把 IP 地址的字符串形式转换为二进制形式*/
intf_set(handle, entry);                       /*设置网络接口的目的地址为 192.168.0.7*/
printf("show the network interface:\n");
intf_loop(handle,show_interface_content,NULL); /*显示所有网络接口的信息*/
printf("set the network interface address:\n");
interface_address = &entry->intf_addr;         /*网络接口的地址*/
addr_pton("192.168.0.8", interface_address);   /*把 IP 地址的字符串形式转换为二进制形式*/
intf_set(handle, entry);                       /*设置网络接口的地址为 192.168.0.8*/
printf("show the network interface:\n");
intf_loop(handle,show_interface_content,NULL); /*显示所有网络接口的信息*/
printf("set the network interface link:\n");
interface_address = &entry->intf_link_addr;    /*硬件地址*/
addr_pton("192.168.0.9", interface_address);   /*把 IP 地址的字符串形式转换为二进制形式*/
intf_set(handle, entry);                       /*设置网络接口的硬件地址为 192.168.0.8 的硬件地址*/
printf("show the network interface:\n");
intf_loop(handle,show_interface_content,NULL); /*显示所有网络接口的信息*/
intf_close(handle);                            /*关闭网络接口操作*/
}
```

在 Linux 下进行编译，其编译命令如下：

```
gcc -o interfaceop interfaceop.c -ldnet
```

程序的运行结果如下：

```
show the network interface:
network interface name is :lo
```

```
flags: 2
flags including:
LOOPBACK
mtu: 16436
address:127.0.0.1/8
-----------------------------------------------------------
network interface name is :eth0
flags: 49
flags including:
UP
BROADCAST
MULTICAST
mtu: 1500
address:192.168.0.8
Link Address: 00:0a:e6:5d:2b:3d
Alias:192.168.0.6
-----------------------------------------------------------
get the special network interface:
network interface name is :eth0
flags: 49
flags including:
UP
BROADCAST
MULTICAST
mtu: 1500
address:192.168.0.8
Link Address: 00:0a:e6:5d:2b:3d
Alias:192.168.0.6
---------- --- ------------------------------------------
show the network interface about the IP :
network interface name is :eth0
flags: 49
flags including:
UP
BROADCAST
MULTICAST
mtu: 1500
address:192.168.0.3
Link Address: 00:0a:e6:5d:2b:3d
Alias:192.168.0.6
-----------------------------------------------------------
show the network interface about the destination IP:
network interface name is :eth0
```

```
mtu: 1500
address:192.168.0.6
Link Address: 00:0a:e6:5d:2b:3d
-----------------------------------------------------------
set the network interface can't use ARP protocol :
show the network interface:
network interface name is :lo
flags: 2
flags including:
LOOPBACK
mtu: 16436
address:127.0.0.1/8
-----------------------------------------------------------
network interface name is :eth0
flags: 57
flags including:
UP
NOARP
BROADCAST
MULTICAST
mtu: 1500
address:192.168.0.6
Link Address: 00:0a:e6:5d:2b:3d
-----------------------------------------------------------
set the network interface can use ARP protocol :
show the network interface:
network interface name is :lo
flags: 2
flags including:
LOOPBACK
mtu: 16436
address:127.0.0.1/8
-----------------------------------------------------------
network interface name is :eth0
flags: 49
flags including:
UP
BROADCAST
MULTICAST
mtu: 1500
address:192.168.0.6
Link Address: 00:0a:e6:5d:2b:3d
-----------------------------------------------------------
```

```
set the network interface alias:
show the network interface:
network interface name is :lo
flags: 2
flags including:
LOOPBACK
mtu: 16436
address:127.0.0.1/8
-----------------------------------------------------------
network interface name is :eth0
flags: 49
flags including:
UP
BROADCAST
MULTICAST
mtu: 1500
address:192.168.0.6
Link Address: 00:0a:e6:5d:2b:3d
-----------------------------------------------------------
set the network interface destination:
show the network interface:
network interface name is :lo
flags: 2
flags including:
LOOPBACK
mtu: 16436
address:127.0.0.1/8
-----------------------------------------------------------
network interface name is :eth0
flags: 49
flags including:
UP
BROADCAST
MULTICAST
mtu: 1500
address:192.168.0.6
Link Address: 00:0a:e6:5d:2b:3d
-----------------------------------------------------------
set the network interface address:
show the network interface:
network interface name is :lo
flags: 2
flags including:
```

```
LOOPBACK
mtu: 16436
address:127.0.0.1/8
-------------------------------------------------------
network interface name is :eth0
flags: 49
flags including:
UP
BROADCAST
MULTICAST
mtu: 1500
address:192.168.0.8
Link Address: 00:0a:e6:5d:2b:3d
Alias:192.168.0.6
-------------------------------------------------------
set the network interface link:
show the network interface:
network interface name is :lo
flags: 2
flags including:
LOOPBACK
mtu: 16436
address:127.0.0.1/8
-------------------------------------------------------
network interface name is :eth0
flags: 49
flags including:
UP
BROADCAST
MULTICAST
mtu: 1500
address:192.168.0.8
Link Address: 00:0a:e6:5d:2b:3d
Alias:192.168.0.6
-------------------------------------------------------
```

程序在 Linux 下的运行界面如图 6-26 所示。

本程序的功能主要涉及了 Libdnet 中的所有网络接口的操作功能，对它们的使用方式进行了演示，每个函数的用法都进行了详细的注解。其中包括的主要功能有：显示特定的网络接口信息，如根据名字、地址进行显示；修改网络接口的参数，主要是设置接口是否激活、是否支持 ARP 以及接口地址等。

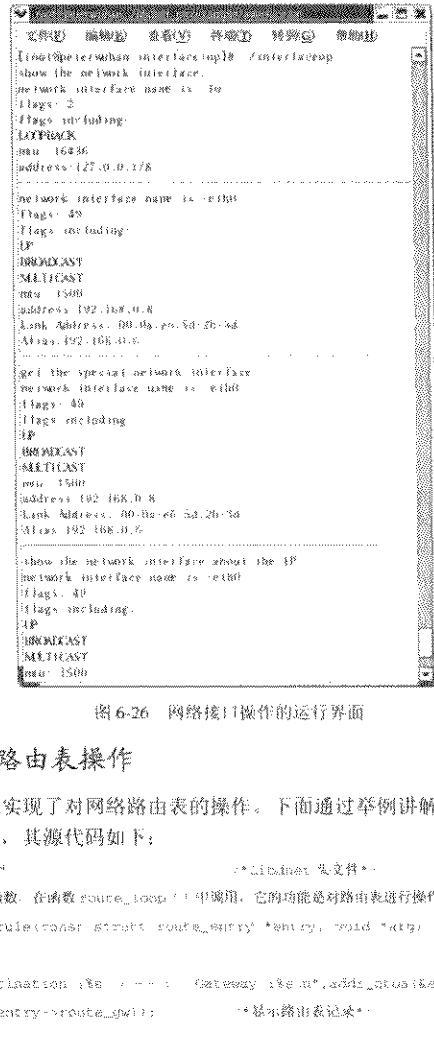

图 6-26 网络接口操作的运行界面

6.5.5 网络路由表操作

在 Libdnet 中也实现了对网络路由表的操作。下面通过举例讲解一下在 Libdnet 中是怎样操作网络路由表的，其源代码如下：

```
#include "dnet.h"                                   /*Libdnet 头文件*/
/*下面的函数是回调函数，在函数route_loop()中调用，它的功能是对路由表进行操作，在这里是显示路由表信息*/
int  show_route_rule(const struct route_entry *entry, void *arg)
{
    printf("Destination %s  ---  Gateway %s\n",addr_ntoa(&entry->route_dst),
    addr_ntoa(&entry->route_gw);                    /*显示路由表记录*/
    return (0);
}
int main(int argc, char *argv[])
{
    struct route_entry entry;                       /*路由表的数据结构的变量*/
    route_t *handle;                                /*路由表操作句柄*/
    handle = route_open();                          /*打开路由表操作*/
```

```
    printf("Show the route rules:\n");
    printf("Route rules :\n");
    printf("Destination IP ---------------------- Gateway IP\n");
    route_loop(handle, show_route_rule, NULL);
    /*注册回调函数show_route_rule，循环调用回调函数对路由表进行操作，这里是显示所有路由表信息*/
    printf("Show the route rules about the Destination IP:\n");
    addr_aton("192.168.0.4", &entry.route_dst);     /*把字符串形式的IP地址转换为二进制形式的IP地址*/
    route_get(handle, &entry);                      /*根据地址获取其路由表信息*/
    printf("Destination is :\n");
    printf("%s\n",addr_ntoa(&entry.route_dst));     /*输出目的地址*/
    printf("Gateway is: \n");
    printf("%s\n",addr_ntoa(&entry.route_gw));      /*输出网关地址*/
    printf("Add a route rule :\n");
    addr_aton("192.168.0.5",&entry.route_dst);      /*把字符串形式的IP地址转换为二进制形式的IP地址*/
    addr_aton("192.168.0.10",&entry.route_gw);      /*把字符串形式的IP地址转换为二进制形式的IP地址*/
    route_add(handle,&entry);                       /*添加路由表记录*/
    printf("Destination :%s Gateway :%s added:\n", addr_ntoa(&entry.route_dst),
    addr_ntoa(&entry.route_gw));
    printf("Add a route rule :\n");
    addr_aton("192.168.0.6",&entry.route_dst);      /*把字符串形式的IP地址转换为二进制形式的IP地址*/
    addr_aton("192.168.0.20",&entry.route_gw);      /*把字符串形式的IP地址转换为二进制形式的IP地址*/
    route_add(handle,&entry);                       /*添加路由表记录*/
printf("Destination :%s Gateway :%s added:\n", addr_ntoa(&entry.route_dst),addr_ntoa
(&entry.route_gw));
    printf("Show the route rules:\n");
    printf("Route rules :\n");
    printf("Destination IP ---------------------- Gateway IP\n");
    route_loop(handle, show_route_rule, NULL);      /*显示所有路由表记录*/
    printf("Delete a route rule :\n");
    addr_aton("192.168.0.5",&entry.route_dst);      /*把字符串形式的IP地址转换为二进制形式的IP地址*/
    route_delete(handle,&entry);                    /*删除路由表记录*/
    printf("Destination :%s deleted\n",addr_ntoa(&entry.route_dst));
    printf("Show the route rules:\n");
    printf("Route rules :\n");
    printf("Destination IP ---------------------- Gateway IP\n");
    route_loop(handle, show_route_rule, NULL);      /*再显示所有路由表记录*/
    route_close(handle);                            /*关闭路由表操作*/
}
```

在Linux下进行编译，其编译的命令如下：

```
Gcc -o routeop routeop.c -ldnet
```

程序的运行结果如下：

```
Show the route rules:
Route rules :
```

```
Destination IP ------------------------ Gateway IP
Destination :0.0.0.0/0  ========  Gateway :192.168.0.1
Show the route rules about the Destination IP:
Destination is :
192.168.0.4
Gateway is:
[null]
Add a route rule :
Destination :192.168.0.5 Gateway :192.168.0.10 added:
Add a route rule :
Destination :192.168.0.6 Gateway :192.168.0.20 added:
Show the route rules:
Route rules :
Destination IP ------------------------ Gateway IP
Destination :192.168.0.5  ========  Gateway :192.168.0.10
Destination :192.168.0.6  ========  Gateway :192.168.0.20
Destination :0.0.0.0/0  ========  Gateway :192.168.0.1
Delete a route rule :
Destination :192.168.0.5 deleted
Show the route rules:
Route rules :
Destination IP ------------------------ Gateway IP
Destination :192.168.0.6  ========  Gateway :192.168.0.20
Destination :0.0.0.0/0  ========  Gateway :192.168.0.1
[root@peterwunan routeop]#
```

运行界面如图 6-27 所示。

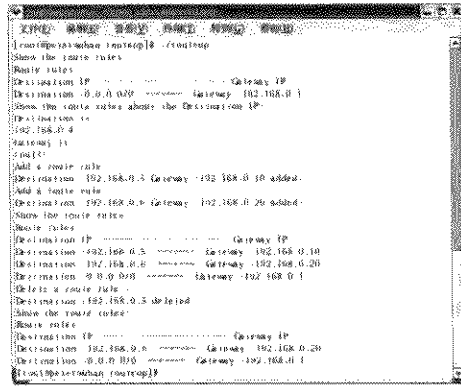

图 6-27　网络路由表操作的运行结果

把此程序原封不动地在 Windows 下进行编译，同样，它使用 Visual C++ 6.0 来编译，用

Visual C++打开此文件，编译生成一个工程，然后对其进行设置，设置的过程与 6.5.2 节设置 ARP 缓存操作程序一模一样，在此不赘述。

设置完后，对其进行编译运行，其结果如图 6-28 所示。

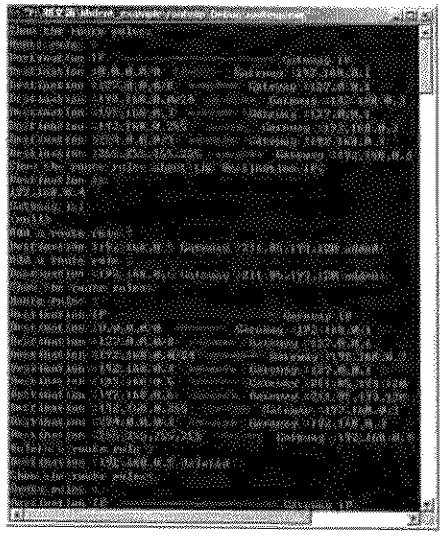

图 6-28　网络路由表操作结果

6.5.6　防火墙操作

使用 Libdnet 可以操作多种防火墙系统，如 Linux 系列的 ipchains，pf，ipfw，还有 Windows 平台的 PktFilter 系统。

在此以 Windows 平台下的 PktFilter 进行举例介绍。PktFilter 是 Windows 平台下的一个包过滤防火墙系统，可从网站上直接下载，下载地址为：http://sourceforge.net/projects/pktfilter/。下载后，直接解压缩就可以了。例如，解压缩在路径 E:\PktFilter 下面，其中有三个文件夹：doc，pktctl 和 pktfltsrv。

配置 PktFilter 系统，由于 PktFilter 是基于命令行的，所以进入 DOS 提示框。进入文件夹 pktfltsrv，首先使用命令：

```
pktfltsrv -i 规则文件路径 日志文件路径
```

它定义了规则文件路径和日志文件路径，如图 6-29 所示。

然后使用命令："net start pktfilter"，表示打开 PktFilter 系统，开始运行 PktFilter 系统，它把数据包信息记录在日志文件中。可以查看此日志文件，如图 6-30 所示。

如果想关闭系统，就使用命令"net stop pktfilter"，如图 6-31 所示。

规则的定义形式与 ipchains 的规则定义形式相同，可以查看相关文档。在此，举一个规则文件的例子，如图 6-32 所示。

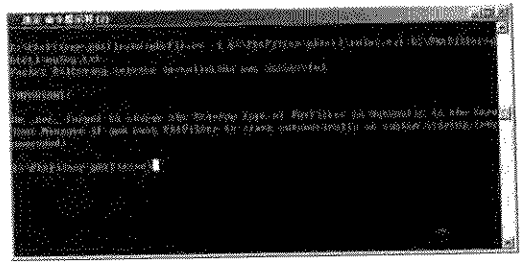

图 6-29　指定规则文件和日志文件

图 6-30　启动 PktFilter 服务

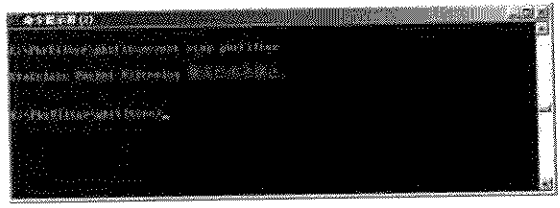

图 6-31　停止 PktFilter 服务

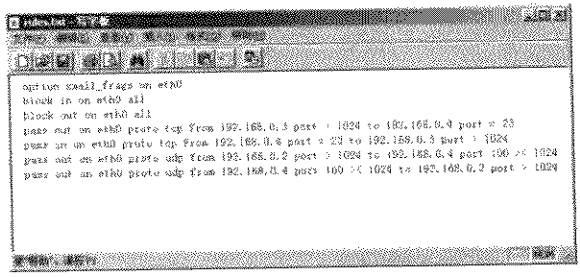

图 6-32　防火墙规则文件举例

规则文件包含的内容如下。

```
option small_frags on eth0
block in on eth0 all
block out on eth0 all
```

```
pass out on eth0 proto tcp from 192.168.0.3 port > 1024 to 192.168.0.4 port = 23
pass in on eth0 proto tcp from 192.168.0.4 port = 23 to 192.168.0.3 port > 1024
pass out on eth0 proto udp from 192.168.0.2 port > 1024 to 192.168.0.4 port 100 >< 1024
pass out on eth0 proto udp from 192.168.0.4 port 100 >< 1024 to 192.168.0.2 port > 1024
```

其中，最开始的是全局选项，后面的是普通规则。第 1 条规则表示阻塞所有进入的网络数据包。第 2 条规则表示阻塞所有出去的网络数据包。第 3 条规则表示允许通过从 192.168.0.3 到 192.168.0.4 的 TCP 协议数据包，源端口号大于 1024，即从 1025 到 65 535；目的端口号为 23。第 4 条规则表示允许通过从 192.168.0.4 到 192.168.0.3 的 TCP 协议数据包，源端口号为 23，目的端口号大于 1024，即从 1025 到 65 535。第 5 条规则表示允许通过从 192.168.0.2 到 192.168.0.4 的 UDP 协议数据包，源端口号大于 1024，即从 1025 到 65 535；目的端口号大于 100 小于 1024；即从 101 到 1023。第 6 条规则表示允许通过从 192.168.0.4 到 192.168.0.2 的 UDP 协议数据包，源端口号大于 100、小于 1024，即从 101 到 1023；目的端口号大于 1024，即从 1025 到 65 535。

下面举一个例子，使用 Libdnet 来操作 PktFilter 系统。实现了显示规则的功能。其源代码如下：

```
#include <dnet.h>                                   /*Libdnet 头文件*/
/*下面是定义的回调函数，它实现了显示防火墙规则的功能*/
int show_firewall_fules(const struct fw_rule *firewall_rule, void *parameter)
{
    char operation[100];                            /*操作，表示是禁止还是允许*/
    char direction[100];                            /*方向，表示是进入还是出去*/
    char net_interface[100];                        /*网络接口*/
    char source_ip[100];                            /*源 IP 地址*/
    char destination_ip[100];                       /*目的 IP 地址*/
    char protocol[16];                              /*规则协议类型，在此只分析 TCP 协议和 UDP 协议*/
    char source_port[16];                           /*源端口号*/
    char destination_port[16];                      /*目的端口号*/
    static int count=1;                             /*规则个数*/
    operation[0]='\0';
    direction[0]='\0';
    net_interface[0]='\0';
    source_ip[0]='\0';
    destination_ip[0]='\0';
    source_port[0]= '\0';
    destination_port[0]='\0';
    if (firewall_rule->fw_proto == 0)
        protocol[0] = '\0';
else
{
        if (firewall_rule->fw_proto==6)             /*表示协议是 TCP 协议*/
            sprintf(protocol, "%s ", "tcp");
        if(firewall_rule->fw_proto==17)             /*表示协议是 UDP 协议*/
```

```
            sprintf(protocol,"%s ","udp");
}
if(firewall_rule->fw_proto==6||firewall_rule->fw_proto==17)
{
/*求源端口号*/
if (firewall_rule->fw_sport[0] == firewall_rule->fw_sport[1])
/*表示端口号只有一个值*/
{
        sprintf(source_port, "port = %d", firewall_rule->fw_sport[0]);
    } else                                          /*表示端口号是一个范围，例如从 1025 到 65535*/
        sprintf(source_port, "port %d-%d",
        firewall_rule->fw_sport[0], firewall_rule->fw_sport[1]);
/*求目的端口号*/
if (firewall_rule->fw_dport[0] == firewall_rule->fw_dport[1])
{
    sprintf(destination_port, "port = %d", firewall_rule->fw_dport[0]);
} else
    sprintf(destination_port, "port %d-%d",
    firewall_rule->fw_dport[0], firewall_rule->fw_dport[1]);
}
if(firewall_rule->fw_op==FW_OP_ALLOW)                          /*判断操作类型*/
    sprintf(operation,"%s","pass");                            /*操作类型是允许*/
else
    sprintf(operation,"%s","block");                           /*操作类型是禁止*/
if(firewall_rule->fw_dir==FW_DIR_IN)                           /*判断方向*/
    sprintf(direction,"%s","in");                              /*表示进入*/
else
    strcpy(direction,"out");                                   /*表示出去*/
sprintf(net_interface ,"%s",firewall_rule->fw_device);         /*获得网络接口*/
if(firewall_rule->fw_src.addr_type)
    sprintf(source_ip,"from %s",addr_ntoa(&firewall_rule->fw_src));
    /*获得源 IP 地址*/
else
    sprintf(source_ip,"%s","");
if(firewall_rule->fw_dst.addr_type)
    sprintf(destination_ip,"to %s",addr_ntoa(&firewall_rule->fw_dst));
    /*获得目的 IP 地址*/
else
    sprintf(destination_ip,"");
printf("第%d 条规则: \n",count);
printf("%s %s %s %s %s %s %s %s %s \n",
        operation,                                         /*操作类型*/
        direction,                                         /*方向*/
```

```
        net_interface,                                  /*网络接口*/
        protocol,                                       /*协议类型*/
        source_ip,                                      /*源 IP 地址*/
        source_port,                                    /*源端口号*/
        destination_ip,                                 /*目的 IP 地址*/
        destination_port,                               /*目的端口号*/
        "");                                            /*输出一个规则*/
count++;                                                /*规则个数加 1*/
return (0);
}
void  main()
{
    struct fw_rule *firewall_rule;                      /*定义防火墙规则变量*/
    fw_t *firewall_handle;                              /*定义防火墙句柄*/
    if ((firewall_handle = fw_open()) == NULL)          /*打开防火墙操作*/
    printf("打开防火墙错误\n");
    fw_loop(firewall_handle,                            /*防火墙句柄*/
            show_firewall_fules,                        /*回调函数的名字*/
            NULL);                                      /*传递给回调函数的参数*/
    /*注册回调函数，对每个规则循环调用回调函数，输出所有规则信息*/
    firewall_rule=(struct fw_rule*)malloc(sizeof(struct fw_rule));
    firewall_rule->fw_op = FW_OP_BLOCK;                 /*操作类型为禁止*/
    firewall_rule->fw_dir =FW_DIR_IN;                   /*方向为进入*/
    strcpy(firewall_rule->fw_device, "eth0");           /*网络接口为 eth0*/
    firewall_rule->fw_proto-6;                          /*协议为 6，表示 TCP 协议*/
    addr_aton("192.168.0.4", &firewall_rule->fw_src);   /*源 IP 地址*/
    addr_aton("192.168.0.5", &firewall_rule->fw_dst);   /*目的 IP 地址* /
    fw_add(firewall_handle, firewall_rule);             /*添加规则*/
    firewall_rule->fw_op = FW_OP_BLOCK;                 /*操作类型为禁止*/
    firewall_rule->fw_dir =FW_DIR_OUT;                  /*方向为出去*/
    strcpy(firewall_rule->fw_device, "eth0");           /*网络接口为 eth0*/
    firewall_rule->fw_proto=6;                          /*协议为 6，表示 TCP 协议*/
    addr_aton("192.168.0.5", &firewall_rule->fw_src);   /*源 IP 地址*/
    addr_aton("192.168.0.4", &firewall_rule->fw_dst);   /*目的 IP 地址*/
    fw_add(firewall_handle, firewall_rule);             /*添加规则*/
    firewall_rule->fw_op = FW_OP_BLOCK;                 /*操作类型为禁止*/
    firewall_rule->fw_dir =FW_DIR_OUT;                  /*方向为出去*/
    strcpy(firewall_rule->fw_device, "eth0");           /*网络接口为 eth0*/
    firewall_rule->fw_proto=6;                          /*协议为 6 表示 TCP 协议*/
    addr_aton("192.168.0.5", &firewall_rule->fw_src);   /*源 IP 地址*/
    addr_aton("192.168.0.4", &firewall_rule->fw_dst);   /*目的 IP 地址*/
    fw_delete(firewall_handle, firewall_rule);          /*删除规则*/
    fw_close(firewall_handle);                          /*关闭防火墙操作*/
```

```
}
```

此程序在 Viusal C++编译，编译过程同编译 ARP 缓存操作程序的步骤一样。

程序的输出内容如下：

```
第 1 条规则：
block in ethb
第 2 条规则：
block out ethb
第 3 条规则：
pass in ethb tcp from 192.168.0.4 port = 23 to 192.168.0.2 port 1025-65535
第 4 条规则：
pass out ethb tcp from 192.168.0.2 port 1025-65535 to 192.168.0.4 port = 23
第 5 条规则：
pass out ethb udp from 192.168.0.2 port 1025-65535 to 192.168.0.4 port 101-1023
第 6 条规则：
pass out ethb udp from 192.168.0.4 port 101-1023 to 192.168.0.2 port 1025-65535
```

程序的编译运行结果如图 6-33 所示。

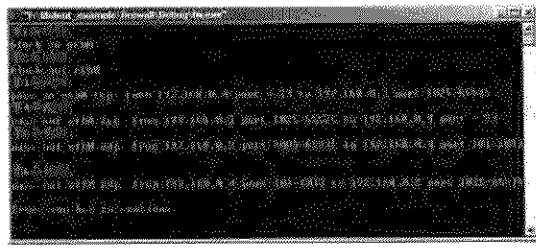

图 6-33　编译运行结果

从运行的结果分析，此程序正确显示了防火墙规则，说明使用 Libdnet 可以操作 PktFilter 防火墙系统。

6.6　小结

Libdnet 是一个通用的网络安全开发包，其中包括一些常用的网络安全操作功能，如地址操作、ARP 缓存操作、路由表操作、网络接口操作、防火墙操作、IP 隧道操作以及网络的数据构造和发送。Libdnet 对这些操作的实现进行了封装，使它们实现起来非常方便，快捷。Libdnet 给我们提供了更高级的控制功能，而且步骤非常简单，即使不懂它的原理，也能轻而易举地实现它们。

由于 Libdnet 具备了非常多有用的功能，所以其应用范围非常广泛。利用它可以构造任何复杂而又功能强大的应用程序。本章对 Libdnet 开发包进行了详细介绍，包括它所具有的功能、各种数据结构定义和输出函数的定义，还通过举例对它们的使用过程进行了详细的阐述。

第 7 章　网络入侵检测开发包 Libnids

7.1　Libnids 概述

Libnids（Library Network Intrusion Detection System）是一个用于网络入侵检测开发的专业编程接口。它实现了基于网络的入侵检测系统的基本框架，并提供了一些基本的功能。使用 Libnids 可以快速地构建基于网络的入侵检测系统，并可以在此基础上进一步扩展开发。Libnids 实现了入侵检测系统的底层功能，使开发者可以专注于高层的功能开发。

Libnids 是基于 Libpcap 和 Libnet 而开发的，所以它具有 Libpcap 和 Libnet 的优点，具有较强的移植性，效率高，使用简单。Libnids 是仿照 Linux 2.0.x 内核中的 TCP/IP 协议部分而实现的，具有高可靠性，并通过了很多测试。

Libnids 的主要功能包括捕获网络数据包、IP 碎片重组、TCP 数据流重组以及端口扫描攻击检测和异常数据包检测等。Libnids 使用了 Libpcap 捕获数据包的功能，可以设定过滤规则，指定捕获感兴趣的数据包。IP 碎片重组是 Libnids 的一个重要内容，它是仿照 Linux 内核中的 IP 重组而实现的，所以非常可靠。Libnids 提供了 TCP 数据流重组功能，这是 Libpcap 所不具备的，利用 TCP 数据流重组，可以分析基于 TCP 协议的各种应用层协议。另外，Libnids 还提供了检测 TCP 端口扫描攻击的功能，检测异常数据包的功能，这是入侵检测系统（IDS）最基本的功能。

Libnids 是由 Rafal Wojtczuk 而开发的，很多其他人员也参与了开发。Libndis 可以在多个平台下运行，包括 Linux，Solaris 以及各种 BSD 类操作系统。在 Windows 平台下有对应的开发包 Libnids-win32，是 Libnids 在 Windows 下的开发版本，其最新稳定版本为 1.19。Libnids 是在 GPL 许可证下发布的，可以从网站上直接下载，其主页为：http://libnids.sourceforge.net/。

Libnids 的最新稳定版本为 1.20，本书以此版本进行介绍。

7.1.1　Libnids 的使用范围

Libndis 使用了 Libpcap，所以它具有捕获数据包的功能，利用 Libnids 可以轻松地实现对数据包的捕获，并对其进行分析。同时，Libnids 提供了 TCP 数据流重组功能，所以对于分析基于 TCP 协议的各种协议 Libnids 都能够胜任。Libnids 还提供了对 IP 分片进行重组的功能，以及端口扫描检测和异常数据包检测功能，所以它对于分析网络异常情况也是非常有效的。总结起来，Libnids 可以用在以下几个方面，但并不局限于此。

1．入侵检测系统

Libnids 的设计是作为入侵检测系统的一个部件来设计的，它实现了入侵检测系统中非常基础的功能，如数据包捕获、协议分析接口等。另外，它还专门针对入侵检测系统的特性，实现了 TCP 数据流重组功能，这对于分析针对 TCP 协议的各种攻击是很有效的。还有，它已经实现了 IP 碎片重组功能、对异常数据包的检测功能以及对 TCP 端口扫描的检测

功能。最重要的是，Libnids 为入侵检测系统更深入的开发提供了开发接口。因此，开发人员可以更关注于对入侵检测技术的研究，而不用考虑底层的实现细节。

2. 网络协议分析

Libnids 是在 Libpcap 基础上开发的，所以它具备了 Libpcap 的功能，可以实现各种协议的分析，并在 Libpcap 的基础上开发了更多有用的功能，如 TCP 数据流重组。这样，在利用 Libnids 分析基于 TCP 协议的各种协议时，不仅可以分析各种单个 TCP 数据包，而且可以分析整个 TCP 连接过程。这对于分析 FTP 协议、HTTP 协议、POP3 协议等基于 TCP 的应用层协议是非常有帮助的。

3. 网络嗅探

网络嗅探也称网络监视，主要是指检测网络信息，查看网络内容。针对不同的目的，有各种各样的网络嗅探。例如，对于网络管理员来说，为了掌握整个网络的运行状况，想检测网络出现的故障，他可以利用网络嗅探技术来实现。对于网络非法攻击者来说，为了获得敏感信息，他也可以利用网路嗅探技术来获得他所需的内容，如密码、用户、账号等。后面，读者将可以看到，利用 Libnids 可以轻松获取很多协议传输的用户和密码。所以说，Libnids 是一把双刃剑。除此之外，利用 Libnids 还可以重现网络内容，还原网络数据，如重现 HTTP 协议中传输的网页、POP3 协议中传输的电子邮件等。

7.1.2 Libnids 的安装

1. Linux 下的安装

由于 Libnids 必须支持库 Libpcap 和 Libnet，所以在安装 Libnids 开发包之前必须先安装 Libpcap 和 Libnet。

（1）安装 Libpcap 开发包。安装 Libpcap 开发包的过程主要分四步，在第 3 章有详细介绍，在这里总结如下：

```
tar -xzvf libpcap-0.8.3.tar.gz
./configure
make
make install
```

（2）安装 Libnet 开发包。安装 Libnet 开发包的过程也是四步骤，在第 5 章有详细介绍，在这里总结如下：

```
tar -xzvf libnet-1.1.2.1.tar.gz
./configure
make
make install
```

（3）安装 Libnids。安装 Libnids 的过程也是分为四步。首先解压缩，执行如下命令，结果如图 7-1 所示。

```
tar -xzvf libnids-1.20.tar.gz
```

然后进入 Libnids 解压缩目录，执行如下配置命令，结果如图 7-2 所示。

```
./configure
```

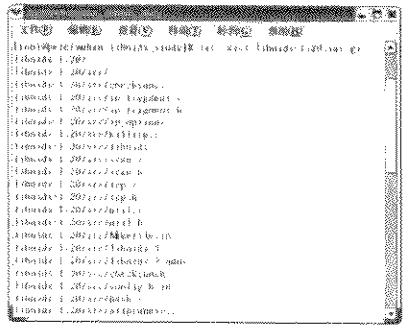

图 7-1　解压缩 Libnids

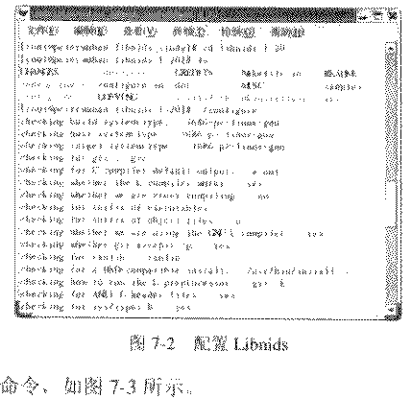

图 7-2　配置 Libnids

然后执行如下编译命令，如图 7-3 所示。

```
make
```

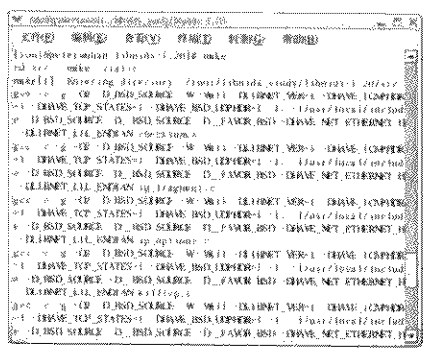

图 7-3　编译 Libnids

最后执行如下安装命令，如图 7-4 所示。

```
make install
```

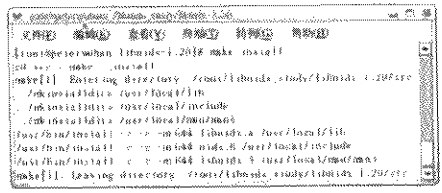

图 7-4　安装 Libnids

2. Windows 平台下的安装

在 Windows 平台下，Libnids 的 Windows 版本是 Libnids-win32，最新版本为 1.19。其安装过程非常简单。

首先从网站上下载其开发包，名称为 Libnids-W32-1.19.tar.gz。

然后用解压缩软件对开发包进行解压缩，存储在一个目录下面，如 F:\Libnids-1.19。打开此文件夹可以发现里面有很多文件。其中，子文件夹 src 下面是 Libnids 的源代码；doc 下面是文档；samples 下面是一些很好的参考例子；WIN32-Code 下面是与 Windows 平台相关的源代码；WIN32-Includes 下面是 Libnids 开发包的包含头文件，这个是开发者所要使用到的；WIN32-Libraries 是 Libnids 开发包的静态链接库文件，也是开发者要用到的；WIN32-PRJ 是 Libndis 开发包本身和参考例子的 Visual C++项目文件。子文件夹 WIN32-Include 和 WIN32-Libraries 是开发者关心的，必须把它们的路径添加到 Visual C++开发环境中，此时就可以使用 Libnids 来开发程序了。

总结起来，在 Windows 下的安装非常简单，只需解压缩就可以了。当然，如果想自己修改 Libnids 的代码，然后进行升级，则可以在 Visual C++ 6.0 中对 Libnids 进行编译连接，生成新的静态链接库文件。具体方法是使用子文件夹下面的项目文件 libnids.dsw，用 Visual C++打开此文件，然后进行编连接就会生成新的 libnids.lib 文件。一般情况下不需要对 Libnids 重新进行编译连接，因为在子文件夹 WIN32-Libraries 下面已经存在 libnids.lib 文件。

7.2　Libnids 数据结构

7.2.1　基本常量

在 Libnids 中定义了一些基本的常量，这些常量都有特定的含义。下面对它们进行介绍。

1. 报警类型

Libnids 提供了对 IP 数据包异常、TCP 数据包异常、UDP 数据包异常以及扫描攻击的检测，对于不同的攻击方式提供不同的报警类型。下面是它们在 Libnids 中的定义。

```
enum
{
  NIDS_WARN_IP = 1,
```

```
  NIDS_WARN_TCP,
  NIDS_WARN_UDP,
  NIDS_WARN_SCAN
};
NIDS_WARN_IP                    /*表示 IP 数据包异常*/
NIDS_WARN_TCP                   /*表示 TCP 数据包异常*/
NIDS_WARN_UDP                   /*表示 UDP 数据包异常*/
NIDS_WARN_SCAN                  /*表示有扫描攻击发生*/
enum
{
  NIDS_WARN_UNDEFINED = 0,
  NIDS_WARN_IP_OVERSIZED,
  NIDS_WARN_IP_INVLIST,
  NIDS_WARN_IP_OVERLAP,
  NIDS_WARN_IP_HDR,
  NIDS_WARN_IP_SRR,
  NIDS_WARN_TCP_TOOMUCH,
  NIDS_WARN_TCP_HDR,
  NIDS_WARN_TCP_BIGQUEUE,
  NIDS_WARN_TCP_BADFLAGS
};
NIDS_WARN_UNDEFINED             /*表示未定义*/
NIDS_WARN_IP_OVERSIZED          /*表示 IP 数据包超长*/
NIDS_WARN_IP_INVLIST            /*表示无效的碎片队列*/
NIDS_WARN_IP_OVERLAP            /*表示发生重叠*/
NIDS_WARN_IP_HDR                /*表示无效 IP 首部，IP 数据包发生异常*/
NIDS_WARN_IP_SRR                /*表示源路由 IP 数据包*/
NIDS_WARN_TCP_TOOMUCH           /*表示 TCP 数据个数太多，因为在 Libnids 中在同一时刻捕获的 TCP 个数
                                  最大值为 TCP 连接参数的哈西表长度的 3/4*/
NIDS_WARN_TCP_HDR               /*表示无效 TCP 首部，TCP 数据包发生异常*/
NIDS_WARN_TCP_BIGQUEUE          /*表示 TCP 接收的队列数据过多*/
NIDS_WARN_TCP_BADFLAGS          /*表示错误标记*/
```

下面是在 Libnids 中定义的警告描述常量：

```
char *nids_warnings[] = {
    "Murphy - you never should see this message !",
    "Oversized IP packet",
    "Invalid IP fragment list: fragment over size",
    "Overlapping IP fragments",
    "Invalid IP header",
    "Source routed IP frame",
    "Max number of TCP streams reached",
    "Invalid TCP header",
    "Too much data in TCP receive queue",
```

```
    "Invalid TCP flags"
};
```

2. Libnids 状态

在对 TCP 数据流进行重组时，必须考虑到 TCP 的连接状态，在 Libnids 中为了方便开发而定义了 Libnids 状态，共有如下 6 种：

```
#define NIDS_JUST_EST 1
#define NIDS_DATA 2
#define NIDS_CLOSE 3
#define NIDS_RESET 4
#define NIDS_TIMED_OUT 5
#define NIDS_EXITING   6
```

Libnids 状态描述的是连接的逻辑状态。真正的 TCP 连接状态有 11 种，它们对应于 TCP 协议的状态变迁图中的各个状态，定义如下：

```
enum {
  TCP_ESTABLISHED = 1,
  TCP_SYN_SENT,
  TCP_SYN_RECV,
  TCP_FIN_WAIT1,
  TCP_FIN_WAIT2,
  TCP_TIME_WAIT,
  TCP_CLOSE,
  TCP_CLOSE_WAIT,
  TCP_LAST_ACK,
  TCP_LISTEN,
  TCP_CLOSING
};
TCP_ESTABLISHED                 /*表示 ESTABLISHED 状态，TCP 连接建立，开始传输数据*/
TCP_SYN_SENT                    /*表示 SYN_SENT 状态，主动打开*/
TCP_SYN_RECV                    /*表示 SYN_RECV 状态，接收 SYN*/
TCP_FIN_WAIT1                   /*表示 FIN_WAIT_1 状态*/
TCP_FIN_WAIT2                   /*表示 FIN_WAIT_2 状态*/
TCP_TIME_WAIT                   /*表示 TIME_WAIT 状态*/
TCP_CLOSE                       /*表示 CLOSED 状态*/
TCP_CLOSE_WAIT                  /*表示 CLOSE_WAIT 状态*/
TCP_LAST_ACK                    /*表示 LAST_ACK 状态*/
TCP_LISTEN                      /*表示 LISTEN 状态*/
TCP_CLOSING                     /*表示 CLOSING 状态*/
```

3. 校验和

在 Libnids 的最新版本中实现了关于是否计算校验和的功能，与此相关的常量定义如下：

```
#define NIDS_DO_CHKSUM  0
```

```
#define NIDS_DONT_CHKSUM   1
NIDS_DO_CHKSUM                    /*表示告诉Libnids要计算校验和*/
NIDS_DON'T_CHKSUM                 /*表示告诉Libnids不需要计算校验和*/
```

7.2.2 tuple4

tuple4 数据结构是 Libnids 中最基本的一种数据结构，其定义如下：

```
struct tuple4
{
u_short source;
u_short dest;
u_int saddr;
u_int daddr;
};
```

tuple4 数据结构描述的是一个地址端口对，它表示发送方 IP 和端口以及接收方 IP 和端口，当它用于 TCP 连接时，其具体意义描述如下：

- 成员 saddr 表示一个 TCP 连接的一端 IP 地址，称为源 IP 地址；
- 成员 daddr 表示一个 TCP 连接的另一端 IP 地址，称为目的 IP 地址，注意此时的源和目的只是相对的；
- 成员 source 表示源 IP 地址的端口号；
- 成员 dest 表示目的 IP 地址的目的端口号。

tuple4 数据结构包括描述 TCP 连接的四个重要参数信息：源 IP 地址、目的 IP 地址，源端口和目的端口。

例如，从 IP 为 192.168.0.3 到 IP 为 192.168.0.5 的一个 TCP 连接，其源端口假设为 1022，目的端口为 21，那么此信息就可以用一个 tuple4 数据结构来描述，代表了一个 TCP 连接的重要信息。

当然，此数据不仅能用在 TCP 协议分析中，也可以用在 UDP 协议分析之中，此时它表示 UDP 发送方的 IP 地址和端口，以及 UDP 接收方的 IP 地址和端口。

7.2.3 half_stream

half_stream 数据结构用来描述在 TCP 连接中一端的所有信息，可以是客户端，也可以是服务器端。half_stream 的数据结构定义如下：

```
struct half_stream
{
char state;
char collect;
char collect_urg;
char *data;
int offset;
int count;
int count_new;
int bufsize;
```

```
int rmem_alloc;
int urg_count;
u_int acked;
u_int seq;
u_int ack_seq;
u_int first_data_seq;
u_char urgdata;
u_char count_new_urg;
u_char urg_seen;
u_int urg_ptr;
u_short window;
u_char ts_on;
u_int curr_ts;
struct skbuff *list;
struct skbuff *listtail;
};
```

half_stream 描述的是一个 TCP 连接的一端的信息。一个 TCP 连接有两端，任何一端可以用此数据结构来进行描述。其中与开发者密切相关的成员有以下几种，其他的可以都不用考虑。

- 成员 state 表示套接字的状态，也就是 TCP 连接状态，如 TCP_CLOSE。
- 成员 collect 可以用来表示有数据到达，此数据存放在 data 成员中，也可以表示不存储此数据到 data 中，此数据忽略。如果大于 0 就存储，否则就忽略。
- 成员 collect_urg 可以表示是否有紧急数据到达，此数据存放在 urgdata 中，也可以表示不存储此数据到 urgdata 中，此数据忽略。如果大于 0 就存储，否则就忽略。
- 成员 data 用来存储正常接收到的数据。
- 成员 offset 表示存储在 data 中数据的第一个字节的偏移量。
- 成员 count 表示从 TCP 连接开始已经存储到 data 中的数据的字节数。
- 成员 count_new 有多少新的数据存储到 data 中，如果为 0，则表示没有新的数据到达。
- 成员 urgdata 用来存储紧急数据。
- 成员 count_new_urg 表示有新的紧急数据到达，如果为 0 表示没有新的紧急数据。

虽然数据结构 half_stream 的成员非常多，但需要开发者特别注意的就那么几个。其中成员 state 非常重要，它表示了当前的 TCP 连接状态，只有知道了 TCP 连接状态之后，才能进行下一步的动作。成员 data 用来存储接收到的正常的网络数据信息，而 urgdata 用来存储接收到的紧急数据。判断是否有新的正常数据到达，是利用成员 count_new 来进行判断的；判断是否有新的紧急数据到达，是用成员 count_new_urg 来进行判断的。如果 count_new 大于 0，就表示有新的正常数据到达；如果 count_new_urg 大于 0，就表示有新的紧急数据达到。然后就可以输出新到达的数据内容，正常数据存储在 data 中，紧急数据存储在 urgdata 中。

7.2.4 tcp_stream

tcp_stream 数据结构描述的是一个 TCP 连接的所有信息，其定义如下。

```
struct tcp_stream
{
struct tuple4 addr;
char nids_state;
struct lurker_node *listeners;
struct half_stream client;
struct half_stream server;
struct tcp_stream *next_node;
struct tcp_stream *prev_node;
int hash_index;
struct tcp_stream *next_time;
struct tcp_stream *prev_time;
int read;
struct tcp_stream *next_free;
};
```

tcp_stream 数据结构描述了一个 TCP 连接的完整信息，包括的内容非常丰富，它是 Libnids 开发包中最重要一个数据结构。其中开发者需要注意的成员有以下几种：

- 成员 addr 是一个 tuple4 类型的成员，它表示了一个 TCP 连接的四个重要信息：源 IP 地址、源端口号、目的 IP 地址和目的端口号。
- 成员 nids_state 表示连接的逻辑状态，共有六种状态：NIDS_JUST_EST，NIDS_DATA，NIDS_CLOSE，NIDS_RESET，NIDS_TIMED_OUT，NIDS_EXITING。这个成员描述了 Libnids 的状态信息。状态信息非常重要，不同的状态代表不同的操作情况，下面分别对它们进行详细阐述。

NIDS_JUST_EST：表示 TCP 连接建立，在此状态下就可以决定是否对此 TCP 连接进行数据分析，可以决定是否捕获 TCP 客户端接收的数据、TCP 服务器端接收的数据、TCP 客户端接收的紧急数据或者 TCP 服务器端接收的紧急数据；

NIDS_CLOSE：表示 TCP 连接正常关闭；

NIDS_RESET：表示 TCP 连接被重置关闭；

NIDS_TIMED_OUT：表示由于超时 TCP 连接被关闭；

NIDS_EXITING：表示 Libnids 正在退出，在这个状态下可以最后一次使用存储在 half_stream 数据结构中的缓存数据；

NIDS_DATA：表示接收数据的状态，在这个状态可以判断是否有新的数据到达，如果有就可以把数据存储起来，可以在这个状态之中来分析 TCP 传输的数据，此数据就存储在 half_stream 数据结构的缓存之中。

- 成员 client 表示客户端信息，成员 server 表示服务器端信息，它们都是 half_stream 类型的。这样 tcp_stream 数据结构就描述了一个完整的 TCP 连接的所有信息。

7.2.5 nids_prm

nids_prm 数据结构描述了 Libnids 的一些全局参数信息，其定义如下：

```
struct nids_prm
{
```

```
int n_tcp_streams;
int n_hosts;
char *device;
char *filename;
int sk_buff_size;
int dev_addon;
void (*syslog) ();
int syslog_level;
int scan_num_hosts;
int scan_delay;
int scan_num_ports;
void (*no_mem) (char *);
int (*ip_filter) ();
char *pcap_filter;
int promisc;
int one_loop_less;
int pcap_timeout;
};
```

此数据结构描述的是 Libnids 的全局参数，利用它可以对 Libnids 的一些环境参数进行设置，设置之后对整个 Libnids 都有效。

- 成员 n_tcp_streams 表示哈西表大小，此哈西表用来存放 tcp_stream 数据结构，默认值为 1040。在同一时刻 Libnids 捕获的 TCP 数据包的最大个数必须是此参数值的 3/4。
- 成员 n_hosts 也是表示哈西表的大小，此哈西表是用来存储 IP 碎片信息的，默认值为 256。
- 成员 filename 表示用来存储网络数据的捕获文件。此文件的类型必须与 Libpcap 类型一致。如果设置了文件，与此同时就应该设置成员 device 为 NULL，默认值为 NULL。
- 成员 device 表示网络接口，Libnids 将在此网络接口上捕获数据，默认值为 NULL。这样 Libnids 将使用 pcap_lookupdev 来查找可用的网络接口。如果其值为 all，表示捕获所有网络接口的数据。
- 成员 sk_buff_size 表示的是数据结构 sk_buff 的大小。数据结构 sk_buff 是 Linux 内核中一个重要的数据结构，是用来进行数据包排队操作的，默认值为 168。
- 成员 dev_addon 表示在数据结构 sk_buff 中用于网络接口上信息的字节数。如果是 -1（默认值），那么 Libnids 会根据不同的网络接口进行修正。
- 成员 syslog 是一个函数指针，默认值为 nids_syslog（）函数。在 syslog 函数中可以检测入侵攻击，如网络扫描攻击，也可以检测一些异常情况，如无效 TCP 标记。
- 成员 syslog_level 表示日志等级，默认值为 LOG_ALERT。
- 成员 scan_num_hosts 表示一个哈西表的大小，（此哈西表用来存储端口扫描信息）表示 Libnids 将要检测的同时扫描的端口数据。如果其值为 0，Libnids 将不提供端口扫描功能。默认值为 256。

- 成员 scan_num_ports 表示相同源地址必须扫描的 TCP 端口数目，默认值为 10。
- 成员 scan_delay 表示在扫描检测中，两端口扫描的间隔时间，以毫秒来计算，缺省值为 3000。
- 成员 no_mem 是一个函数指针，当 Libnids 发生内存溢出时被调用。
- 成员 ip_filter 是一个函数指针，此函数可以用来分析 IP 数据包，当有 IP 数据包到达时，此函数就被调用。如果此函数返回非零值，此数据包就被处理；如果返回零，此 IP 数据包就被丢弃。默认值为 nids_ip_filter 函数，总是返回 1。nids_ip_filter 函数的定义如下：

```
static int nids_ip_filter(struct ip *x, int len)
{
    (void)x;
    (void)len;
    return 1;            /*总是返回 1*/
}
```

- 成员 pcap_filter 表示过滤规则，即 Libpcap 的过滤规则，默认值为 NULL，表示捕获所有数据包。可以在此设置过滤规则，只捕获感兴趣的开发包。
- 成员 promisc 表示网卡模式，如果是非零，就把此网卡设为混杂模式；否则，设为非混杂模式。默认值为 1。
- 成员 one_loop_less 初始值为 0。
- 成员 pcap_timeout 表示捕获数据返回的时间，以毫秒计算。实际上它表示的就是 Libpcap 函数中的 pcap_open_live 函数的 timeout 参数，默认值为 1024。

在 Libnids 中用 nids_prm 数据结构定义了一个全部变量 nids_params，其定义和初始值如下。

```
struct nids_prm nids_params = {
    1040,                       /*n_tcp_streams*/
    256,                        /*n_hosts*/
    NULL,                       /*device*/
    NULL,                       /*filename*/
    168,                        /*sk_buff_size*/
    -1,                         /*dev_addon*/
    nids_syslog,                /*syslog()*/
    LOG_ALERT,                  /*syslog_level*/
    256,                        /*scan_num_hosts*/
    3000,                       /*scan_delay*/
    10,                         /*scan_num_ports*/
    nids_no_mem,                /*no_mem()*/
    nids_ip_filter,             /*ip_filter()*/
    NULL,                       /*pcap_filter*/
    1,                          /*promisc */
    0,                          /*one_loop_less*/
    1024                        /*pcap_timeout*/
};
```

在使用 Libndis 开发程序时，可以首先对 nids_params 全局变量的值进行修改，这样对整个 Libnids 就全部有效。

7.2.6 nids_chksum_ctl

nids_chksum_ctl 数据结构描述的是计算校验和，其定义如下：

```
struct nids_chksum_ctl {
    u_int netaddr;
    u_int mask;
    u_int action;
    u_int reserved;
};
```

此数据结构用于决定是否计算机校验和，其各成员说明如下：

- 成员 netaddr 表示地址。
- 成员 mask 表示掩码。
- 成员 action 表示动作，如果是 NIDS_DO_CHKSUM，表示要计算校验和；如果是 NIDS_DONT_CHKSUM，表示不计算校验和。
- 成员 reserved 保留未用。

7.3 Libnids 函数

1. 基本函数

（1）int nids_init (void);

函数返回值：函数调用成功就返回 1，失败就返回 0。

参数描述：无。

此函数的功能是对 Libnids 进行初始化，这是所有设计基于 Libnids 的程序最开始调用的函数。它的主要内容包括打开网络接口、打开文件、编译过滤规则、设置过滤规则、判断网络链路层类型、进行必要的初始化工作。

（2）void nids_run (void);

函数返回值：无。

参数描述：无。

此函数的功能是运行 Libnids，进入循环捕获数据包状态。它实际上是调用 Libpcap 函数 pcap_loop()来循环捕获数据包。

（3）int nids_getfd (void);

函数返回值：函数操作成功就返回文件描述符，失败就返回－1。

参数描述：无。

此函数的功能是获得文件描述符号。

（4）int nids_dispatch(int cnt)

函数返回值：函数执行成功就返回个数，失败就返回负数。

参数描述：参数 cnt 表示捕获的数据包的个数。

此函数的功能是调用 Libpcap 中的捕获数据包函数 pcap_dispatch（）。

（5）int nids_next (void);

函数返回值：函数执行成功就返回 1，失败就返回 0。

参数描述：无。

此函数的功能是调用 Libpcap 中的捕获数据包函数 pcap_next（）。

（6）void nids_register_chksum_ctl(struct nids_chksum_ctl * ptr, int nr)

函数返回值：无。

参数描述：参数 ptr 表示 nids_chksum_ctl 列表，参数 nr 表示列表中的个数。

此函数的功能是决定是否计算校验和。它是根据数据结构 nids_chksum_ctl 中的 action 进行决定的，如果所要计算的对象不在列表中，则必须都要计算校验和。

2. IP 碎片函数

（1）void nids_register_ip_frag (void (*));

函数返回值：无。

参数描述：参数应该是一个回调函数名字。

此函数的功能是注册一个能够检测所有 IP 数据包的回调函数，包括 IP 碎片。例如，可以使用如下方式调用：

```
nids_register_ip_frag(ip_frag_function);
```

这样就注册了一个回调函数 ip_frag_function 的定义类型如下：

```
void ip_frag_function(struct ip * a_packet, int len)
```

其中参数 a_packet 表示接收的 IP 数据包，参数 len 表示接收的数据包的长度。

此回调函数中可以检测所有的 IP 数据包，包括 IP 碎片。

（2）void nids_register_ip (void (*));

函数返回值：无。

参数描述：参数应该是一个回调函数的名字。

此函数是注册一个回调函数，此回调函数可以接收正常的 IP 数据包。可以使用如下方式调用：

```
nids_register_ip(ip_function);
```

注册一个回调函数 ip_function，此回调函数的定义类型如下：

```
void ip_function(struct ip * a_packet)
```

其中参数 a_packet 表示的是所捕获的 IP 数据包。

此回调函数可以接收正常的 IP 数据包，并在此函数中对捕获到的 IP 数据包进行分析。

3. TCP 数据流重组函数

（1）void nids_register_tcp (void (*));

函数返回值：无。

参数描述：参数应该是一个回调函数。

此函数是注册一个 TCP 连接的回调函数。回调函数的类型定义如下：

```
void tcp_callback(struct tcp_stream * ns, void ** param);
```

其中参数 ns 表示一个 TCP 连接的所有信息，它的类型是 tcp_stream 数据结构；参数 param 表示要传递的连接参数信息，可以指向一个 TCP 连接的私有数据。

此回调函数接收的 TCP 数据存放在 half_stream 的缓存中，应该马上取出来，一旦此回

```
    ascii_string[0] = 0;
    string = ascii_string;
    if (isgraph(ch))                        /*可打印字符*/
    {
        *string++ = ch;
    }
    else if (ch == ' ')                     /*空格*/
    {
        *string++ = ch;
    }
    else if (ch == '\n' || ch == '\r')      /*回车和换行*/
    {
        *string++ = ch;
    }
    else                                    /*其他字符以点 "." 表示*/
    {
        *string++ = '.';
    }
    *string = 0;
    return ascii_string;
}
/*下面的函数是回调函数，用于分析 TCP 连接和 TCP 连接状态，并对 TCP 协议传输的数据进行分析*/
void tcp_protocol_callback(struct tcp_stream* tcp_connection, void** arg)
{
int i;
char address_string[1024];
char content[65535];
char content_urgent[65535];
struct tuple4 ip_and_port = tcp_connection->addr;   /*获取 TCP 连接的地址和端口对*/
strcpy(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.saddr)))); /*获取源地址*/
sprintf(address_string + strlen(address_string), " : %i", ip_and_port.source); /*获取源端口*/
strcat(address_string, " <---> ");
strcat(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.daddr)))); /*获取目的地址*/
sprintf(address_string + strlen(address_string), " : %i", ip_and_port.dest);   /*获取目的端口*/
    strcat(address_string, "\n");
    switch (tcp_connection->nids_state)             /*判断 LIBNIDS 的状态*/
    {
    case NIDS_JUST_EST:                             /*表示 TCP 客户端和 TCP 服务器端建立连接状态*/
        tcp_connection->client.collect++;           /*客户端接收数据*/
        tcp_connection->server.collect++;           /*服务器接收数据*/
        tcp_connection->server.collect_urg++;       /*服务器接收紧急数据*/
        tcp_connection->client.collect_urg++;       /*客户端接收紧急数据*/
        printf("%sTCP 连接建立\n", address_string);
```

```
        return;
    case NIDS_CLOSE:                    /*表示 TCP 连接正常关闭*/
        printf("--------------------------------\n");
        printf("%sTCP 连接正常关闭\n", address_string);
        return;
    case NIDS_RESET:                    /*表示 TCP 连接被 RST 关闭*/
        printf("--------------------------------\n");
        printf("%sTCP 连接被 RST 关闭\n", address_string);
        return;
    case NIDS_DATA:                     /*表示有新的数据到达*/
        {
            struct half_stream* hlf; /*表示 TCP 连接的一端的信息，可以是客户端，也可以是服务器端*/
            if (tcp_connection->server.count_new_urg)
            {                           /*表示 TCP 服务器端接收到新的紧急数据*/
                printf("--------------------------------\n");
                strcpy(address_string,inet_ntoa(*((struct in_addr *) &(ip_and_port.saddr))));
                sprintf(address_string + strlen(address_string), " : %i", ip_and_port.source);
                strcat(address_string, " urgent---> ");
                strcat(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.daddr))));
                sprintf(address_string + strlen(address_string), " : %i", ip_and_port.dest);
                strcat(address_string, "\n");
                address_string[strlen(address_string) + 1] = 0;
                address_string[strlen(address_string)] = tcp_connection->server.urgdata;
                printf("%s", address_string);
                return;
            }
            if (tcp_connection->client.count_new_urg)
            {                           /*表示 TCP 客户端接收到新的紧急数据*/
                printf("--------------------------------\n");
                strcpy(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.saddr))));
                sprintf(address_string + strlen(address_string), " : %i", ip_and_port.source);
                strcat(address_string, " <--- urgent ");
                strcat(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.daddr))));
                sprintf(address_string + strlen(address_string), " : %i", ip_and_port.dest);
                strcat(address_string, "\n");
                address_string[strlen(address_string) + 1] = 0;
                address_string[strlen(address_string)] = tcp_connection->client.urgdata;
                printf("%s", address_string);
                return;
            }
            if (tcp_connection->client.count_new)
            {                           /*表示客户端接收到新的数据*/
                hlf = &tcp_connection->client;   /*此时 hlf 表示的是客户端的 TCP 连接信息*/
```

```
}
void main()
{
    if (nids_init())    /*Libnids 初始化*/
    {
        printf("出现错误: %s\n", nids_errbuf);
        exit(1);
    }
    nids_register_tcp(tcp_protocol_callback);       /*注册回调函数*/
    nids_run();                                     /*Libnids 进入循环捕获数据包状态*/
}
```

此程序的功能是重组 TCP 数据流，显示它们的连接状态以及传输的详细数据。利用此程序可以分析使用 Libnids 的开发流程。首先用函数 nids_init()进行初始化，然后注册相应的回调函数，最后让调用函数 nids_run()进入循环捕获数据包的状态。不同的回调函数实现不同的功能，主要的工作都是在回调函数中实现的。

下面对它们的编译过程进行介绍，在这里主要介绍在 Linux 平台和 Windows 平台下的编译过程。后面所有的例程的编译过程与此相同。

1. Linux 下的编译

在 Linux 下编译此程序非常简单，使用 gcc 编译命令，使用到的开发包是 libnids，libpcap 和 libnet，它们的库名称分别为 nids，pcap 和 net，执行命令如下：

```
gcc -o show_tcp_connection tcp.c -lnids -lpcap -lnet
```

后面所有例子在 Linux 下的编译过程与此相同。

2. Windows 下的编译

首先，用 Visual C++ 6.0 打开此源文件，然后利用菜单 Build 的子菜单 Compile 来编译此程序，会弹出如图 7-5 所示的对话框。点击“是(Y)”按钮，生成一个默认的项目工作空间。

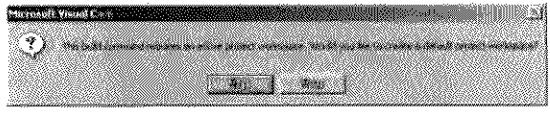

图 7-5　提示生成工作空间

其次，对此项目进行设置。先设置程序的包含头文件路径和静态链接库路径，打开菜单 Tools 的子菜单 Options，弹出 Options 对话框，选择 Directories 页面，再在“Show Derectories for:”中选择 Include files，添加 Libnids 的包含头文件路径。在作者机子中 Libnids 存放在 F:\Libnids-1.19 中，所以其 Include files 路径应该是 F:\Libnids-1.19\WIN32-Includes，如图 7-6 所示。

第三，在“Show Derectories for:”中选择 Library files，添加 Libnids 的静态链接库路径。其在作者的机子里面是路径 F:\Libnids-1.19\WIN32-Libraries，如图 7-7 所示。

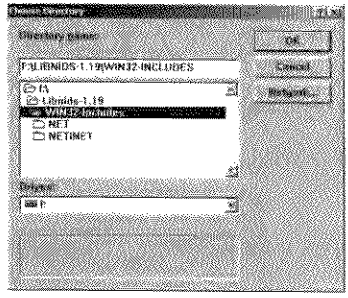

图 7-6　添加 Libnids 的包含头文件路径

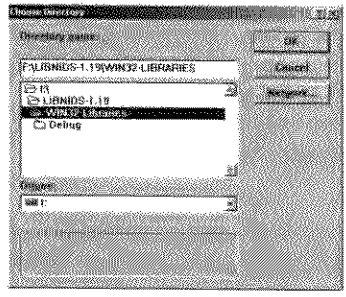

图 7-7　添加 Libnids 的静态链接库路径

由于 Libnids 在 Windows 系统中还需要 WinPcap 的支持，所以必须安装 WinPcap 开发包。在“Show Derectories for:”中选择“Library files”，添加 WinPcap 的静态链接库路径，在作者机子上 WinPcap 开发包安装在 F:\wpdpack\ 下，所以其静态链接库路径为 F:\wpdpack\Lib，如图 7-8 所示。

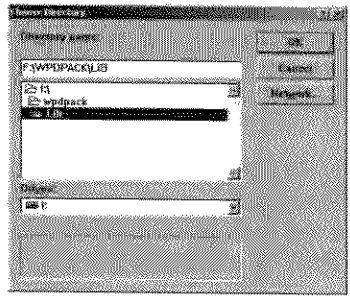

图 7-8　添加 WinPcap 的静态链接库路径

第四，添加程序编译必需的静态链接库，具体步骤是选择菜单“Project”的子菜单“Setting...”，打开对话框“Project Settings”，选择“Link”页面，在“Object/library

modules”中添加三个静态链接库文件 libnids.lib，wpcap.lib 和 wsock32.lib，中间以空格隔开，如图 7-9 所示。

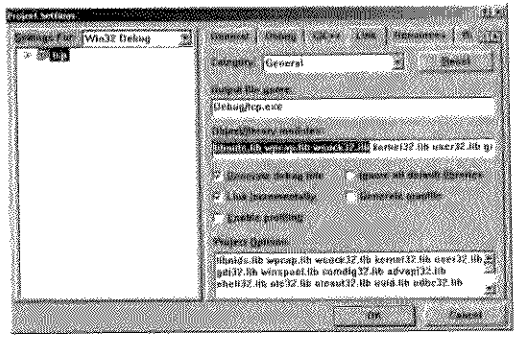

图 7-9　添加静态链接库文件

第五，修改运行时的库类型，在“C/C++”页面，在选项“Category”中选择“Code Generation”，然后在“Use run-time library”中选择“Debug Multithreaded”，如图 7-10 所示。

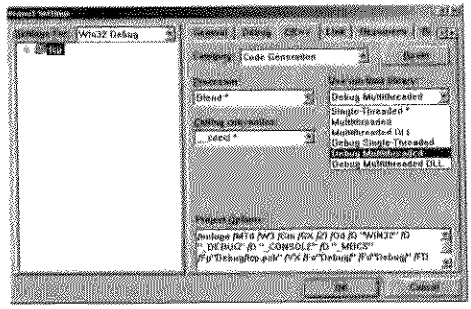

图 7-10　选择运行时的库类型

最后，就可以成功编译了。

下面来分析一下程序的运行结果，此程序的功能是对 TCP 数据流进行重组，显示 TCP 连接状态，并对 TCP 的传输数据进行详细分析并输出。

运行此程序，然后进行测试。作者通过建立 FTP 服务器，然后通过另外一台机子来访问此 FTP 服务器，以产生 TCP 数据流，因为 FTP 是基于 TCP 协议的。运行此程序，输出内容如下。

```
192.168.0.212 : 1615 <---> 192.168.0.211 : 21
TCP 连接建立
-----------------------------------
192.168.0.212:1615 <--- 192.168.0.211:21
客户端接收数据
```

```
220 Serv-U FTP Server v4.0 for WinSock ready...
--------------------------------
192.168.0.212:1615 ---> 192.168.0.211:21
服务器端接收数据
USER peter
--------------------------------
192.168.0.212:1615 <--- 192.168.0.211:21
客户端接收数据
331 User name okay, need password.
--------------------------------
192.168.0.212:1615 ---> 192.168.0.211:21
服务器端接收数据
PASS 1234567890
--------------------------------
192.168.0.212:1615 <--- 192.168.0.211:21
客户端接收数据
230 User logged in, proceed.
--------------------------------
192.168.0.212:1615 ---> 192.168.0.211:21
服务器端接收数据
PORT 192,168,0,212,6,81
--------------------------------
192.168.0.212:1615 <--- 192.168.0.211:21
客户端接收数据
200 PORT Command successful.
--------------------------------
192.168.0.212:1615 ---> 192.168.0.211:21
服务器端接收数据
LIST
192.168.0.211 : 20 <---> 192.168.0.212 : 1617
TCP 连接建立
--------------------------------
192.168.0.212:1615 <--- 192.168.0.211:21
客户端接收数据
150 Opening ASCII mode data connection for /bin/ls.
--------------------------------
192.168.0.211:20 ---> 192.168.0.212:1617
服务器端接收数据
drw-rw-rw-   1 user    group          0 May 26 19:58 .
--------------------------------
192.168.0.211:20 ---> 192.168.0.212:1617
服务器端接收数据
drw-rw-rw-   1 user    group          0 May 26 19:58 ..
```

```
drw-rw-rw-   1 user    group          0 May 26 19:59 company
drw-rw-rw-   1 user    group          0 May 26 19:58 保险
-rw-rw-rw-   1 user    group      10752 May 26 19:57 编号.doc
drw-rw-rw-   1 user    group          0 May 26 19:58 福利
-rw-rw-rw-   1 user    group      10752 May 26 19:57 ?作条例.xls
-rw-rw-rw-   1 user    group      10752 May 26 19:57 规章?度.doc
-rw-rw-rw-   1 user    group      10752 May 26 19:57 员??资.doc
--------------------------------
192.168.0.211 : 20 <---> 192.168.0.212 : 1617
TCP 连接正常关闭
--------------------------------
192.168.0.212:1615 <--- 192.168.0.211:21
客户端接收数据
226 Transfer complete.
--------------------------------
192.168.0.212:1615 ---> 192.168.0.211:21
服务器端接收数据
CWD comany
--------------------------------
192.168.0.212:1615 <--- 192.168.0.211:21
客户端接收数据
550 /d:/ftp/comany: No such file or directory.
--------------------------------
192.168.0.212:1615 ---> 192.168.0.211:21
服务器端接收数据
CWD company
--------------------------------
192.168.0.212:1615 <--- 192.168.0.211:21
客户端接收数据
250 Directory changed to /d:/ftp/company
--------------------------------
192.168.0.212:1615 ---> 192.168.0.211:21
服务器端接收数据
PORT 192,168,0,212,6,83
--------------------------------
192.168.0.212:1615 <--- 192.168.0.211:21
客户端接收数据
200 PORT Command successful.
--------------------------------
192.168.0.212:1615 ---> 192.168.0.211:21
服务器端接收数据
LIST
192.168.0.211 : 20 <---> 192.168.0.212 : 1619
```

TCP 连接建立

192.168.0.212:1615 <--- 192.168.0.211:21

客户端接收数据

150 Opening ASCII mode data connection for /bin/ls.

192.168.0.211:20 ---> 192.168.0.212:1619

服务器端接收数据

drw-rw-rw- 1 user group 0 May 26 19:59 .

192.168.0.211:20 ---> 192.168.0.212:1619

服务器端接收数据

drw-rw-rw- 1 user group 0 May 26 19:59 ..

-rw-rw-rw- 1 user group 44 May 26 19:59 user.txt

192.168.0.211 : 20 <---> 192.168.0.212 : 1619

TCP 连接正常关闭

192.168.0.212:1615 <--- 192.168.0.211:21

客户端接收数据

226 Transfer complete.

192.168.0.212:1615 ---> 192.168.0.211:21

服务器端接收数据

PORT 192,168,0,212,6,84

192.168.0.212:1615 <--- 192.168.0.211:21

客户端接收数据

200 PORT Command successful.

192.168.0.212:1615 ---> 192.168.0.211:21

服务器端接收数据

RETR user.txt

192.168.0.211 : 20 <---> 192.168.0.212 : 1620

TCP 连接建立

192.168.0.212:1615 <--- 192.168.0.211:21

客户端接收数据

150 Opening ASCII mode data connection for user.txt (44 bytes).

192.168.0.211:20 ---> 192.168.0.212:1620

服务器端接收数据

user:peter

```
password:123456
content:test
-------------------------------
192.168.0.211 : 20 <---> 192.168.0.212 : 1620
```

TCP 连接正常关闭

```
-------------------------------
192.168.0.212:1615 <--- 192.168.0.211:21
```

客户端接收数据

```
226 Transfer complete.
-------------------------------
192.168.0.212:1615 ---> 192.168.0.211:21
```

服务器端接收数据

```
QUIT
-------------------------------
192.168.0.212:1615 <--- 192.168.0.211:21
```

客户端接收数据

```
221 Goodbye!
-------------------------------
192.168.0.212 : 1615 <---> 192.168.0.211 : 21
```

TCP 连接正常关闭

程序的运行结果如图 7-11 所示。

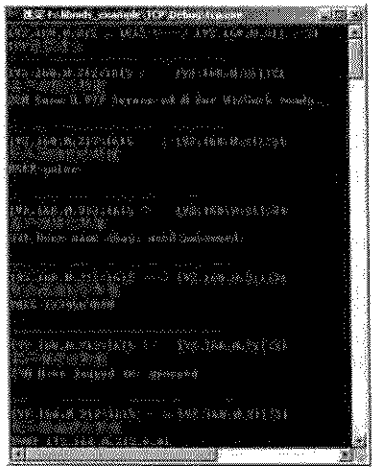

图 7-11　显示 TCP 连接过程的运行结果

对程序的输出内容进行分析，可以发现利用 Libnids 可以轻松对 TCP 连接进行显示，包括它的连接状态，传输的数据内容等。此程序不仅可以分析、显示 FTP 协议的连接状态，还可以显示所有基于 TCP 协议的应用层协议。只要是基于 TCP 协议的协议，都可以利用此

程序来分析其连接状态和运行机理。

但是，本程序只是分析了 TCP 的连接状态和输出的内容，但没有对具体的协议进行分析，没有对传输的数据进行更详细的分析，后面几节分别对 HTTP 协议、FTP 协议、TELENT 协议、POP3 协议、SMTP/ESMTP 协议进行详细分析。利用 Libnids 可以针对基于 TCP 协议的任何协议进行分析，可以分析出协议中传输的重要信息，如用户名、密码及其他重要内容等，也可以对所传输的内容进行重现。例如，重现 HTTP 协议传输的页面图面、POP3 协议中传输的邮件等。那就需要更仔细、更深入的分析，这也就是所谓的嗅探。很多攻击者就是利用此技术来获得敏感信息的。把此项技术用在入侵检测系统之中是非常合适的，也是非常有用的。利用分析的结果与入侵特征进行匹配，就可以发现是否有入侵行为发生。对传输的内容分析得越仔细、越深入，就更容易发现是否有异常行为。所以，Libnids 用在入侵检测系统中是再好不过了，本来它就是为入侵检测系统而设计的一个开发包。

注意，在 Windows 平台下运行基于 Libnids 的程序时，必须先安装 WinPcap。这样程序才能运行，不然会出现如图 7-12 所示的错误。WinPcap 的具体安装步骤请参阅第 4 章的相关内容。

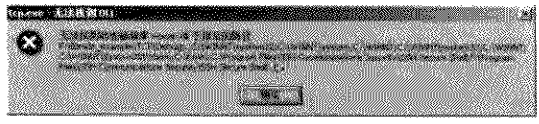

图 7-12　DLL 文件找不到

7.4.2　显示 UDP 数据报

在 7.4.1 节分析了 TCP 协议，对 TCP 数据流进行了重组，可以分析出 TCP 的连接状态以及传输的内容。在本节要利用 Libnids 实现对 UDP 协议的分析，并显示 UDP 数据内容。在 Libnids 中不仅提供了对 TCP 协议的分析，也提供了对 UDP 协议的分析途径。

在 Libnids 中，函数 nids_register_udp（）提供了捕获 UDP 协议的接口，在此函数中注册回调函数，然后在回调函数中对 UDP 协议进行分析。其回调函数类型如下：

```
void udp_callback(struct tuple4* addr, char* buf, int len, struct ip* iph)
```

其中参数 addr 表示的是一对协议地址和端口，分别是 UDP 协议的源 IP 地址，源端口，目的 IP 地址和目的端口；参数 buf 表示的是 UDP 协议的负载内容，即 UDP 协议首部后面紧跟的具体内容，它是 UDP 协议所要传输的内容，也是基于 UDP 协议的应用层协议内容；参数 len 表示参数 buf 的长度，即 UDP 协议负载的长度；参数 iph 表示的是 IP 数据，包括 IP 协议首部、UDP 协议首部以及 UDP 负载内容。显示 UDP 数据报的源代码如下：

```
#include "nids.h"
/*下面是 UDP 协议首部的数据结构*/
struct udp_header
{
    unsigned short udp_source_port;          /*源端口号*/
    unsigned short udp_destination_port;     /*目的端口号*/
    unsigned short udp_length;               /*长度*/
    unsigned short udp_checksum;             /*校验和*/
```

```
};
/*下面是IP协议首部的数据结构*/
struct ip_header
{
#if defined(WORDS_BIGENDIAN)
    unsigned char ip_version : 4,                /*版本*/
        ip_header_length : 4;                    /*首部长度*/
#else
    unsigned char ip_header_length : 4,
        ip_version : 4;
#endif
    unsigned char ip_tos;                        /*服务类型*/
    unsigned short ip_length;                    /*总长度*/
    unsigned short ip_id;                        /*标识*/
    unsigned short ip_off;                       /*标志和偏移*/
    unsigned char ip_ttl;                        /*生存时间/
    unsigned char ip_protocol;                   /*协议*/
    unsigned short ip_checksum;                  /*校验和*/
    struct in_addr ip_soucc_address;             /*源IP地址*/
    struct in_addr ip_destination_address;       /*目的IP地址*/
};
char ascii_string[10000];
char* char_to_ascii(char ch)
{
    char* string;
    ascii_string[0] = 0;
    string = ascii_string;
    if (isgraph(ch))
    {
        *string++ = ch;
    }
    else if (ch == ' ')
    {
        *string++ = ch;
    }
    else if (ch == '\n' || ch == '\r')
    {
        *string++ = ch;
    }
    else
    {
        *string++ = '.';
    }
```

```
    *string = 0;
    return ascii_string;
}
/*下面是分析UDP协议的函数的定义*/
void udp_protocol_packet_callback(u_char* packet_content)
{
    struct udp_header* udp_protocol;
    u_short source_port;
    u_short destination_port;
    u_short length;
    udp_protocol = (struct udp_header *) (packet_content + 20);        /*获取UPD协议数据*/
    source_port = ntohs(udp_protocol->udp_source_port);                /*获取源端口*/
    destination_port = ntohs(udp_protocol->udp_destination_port);      /*获取目的端口*/
    length = ntohs(udp_protocol->udp_length);                          /*获取长度*/
    printf("---------- UDP协议首部  ----------\n");
    printf("源端口:%d\n", source_port);
    printf("目的端口:%d\n", destination_port);
    switch (destination_port)                  /*判断端口号，确定上层协议类型*/
    {
    case 138:
        printf("NETBIOS Datagram Service\n");  break;
    case 137:
        printf("NETBIOS Name Service\n");break;
    case 139:
        printf("NETBIOS session service\n");break;
    case 53:
        printf("name-domain server \n");break;
    default:
        break;
    }
    printf("长度:%d\n", length);
    printf("校验和:%d\n", ntohs(udp_protocol->udp_checksum));      /*获取校验和*/
}                                              /*下面是实现IP协议分析的函数的定义*/
void ip_protocol_packet_callback(u_char* packet_content)
{
    struct ip_header* ip_protocol;
    u_int header_length;
    u_int offset;
    u_char tos;
    unsigned short checksum;
    printf("---------- IP协议首部  ----------\n");
    ip_protocol = (struct ip_header *) (packet_content);           /*获取IP协议数据*/
    checksum = ntohs(ip_protocol->ip_checksum);                    /*获取校验和*/
```

```
    header_length = ip_protocol->ip_header_length * 4;          /*获取长度*/
    tos = ip_protocol->ip_tos;                                  /*获取服务类型*/
    offset = ntohs(ip_protocol->ip_off);                        /*获取标志和偏移*/
    printf("IP 版本:%d\n", ip_protocol->ip_version);
    printf("首部长度:%d\n", header_length);
    printf("TOS:%d\n", tos);
    printf("总长度:%d\n", ntohs(ip_protocol->ip_length));
    printf("标识:%d\n", ntohs(ip_protocol->ip_id));
    printf("偏移:%d\n", (offset & 0x1fff) * 8);
    printf("生存时间:%d\n", ip_protocol->ip_ttl);
    printf("协议:%d\n", ip_protocol->ip_protocol);
    switch (ip_protocol->ip_protocol)                           /*判断上层协议类型*/
    {
    case 6:
        printf("上层协议为 TCP\n");break;
    case 17:
        printf("上层协议为 UDP\n");break;
    case 1:
        printf("上层协议为 ICMP\n");break;
    default:
        break;
    }
    printf("校验和:%d\n", checksum);
    printf("源 IP 地址:%s\n", inet_ntoa(ip_protocol->ip_souce_address));
/*获取源 IP 地址*/
    printf("目的 IP 地址:%s\n", inet_ntoa(ip_protocol->ip_destination_address));
/*获取目的 IP 地址*/
    switch (ip_protocol->ip_protocol)
    {
    case 17:
        udp_protocol_packet_callback(packet_content);break;
        /*调用分析 UDP 协议的函数*/
    default:
        break;
    }
}
/*下面是回调函数，此回调函数将在函数 nids_register_udp () 中被注册*/
void udp_callback(struct tuple4* addr, char* buf, int len, struct ip* iph)
{
    int i;
    char content[65535];
    char content_urgent[65535];
    char tcp_content[65535];
```

```
    char buffer[1024];
    strcpy(buffer, inet_ntoa(*((struct in_addr *) &(addr->saddr))));
    sprintf(buffer + strlen(buffer), " : %i", addr->source);
    strcat(buffer, " -> ");
    strcat(buffer, inet_ntoa(*((struct in_addr *) &(addr->daddr))));
    sprintf(buffer + strlen(buffer), " : %i", addr->dest);
    strcat(buffer, "\n");
    printf("------------------- BEGIN --------------------------\n");
    printf("%s", buffer);
    ip_protocol_packet_callback(iph);                          /*调用分析 IP 协议的函数*/
    printf("----------  UDP 数据包内容  ----------");          /*输出 UDP 负载数据内容*/
    for (i = 0; i < len; i++)
    {
        if (i % 50 == 0)
        {
            printf("\n");
        }
        printf("%s", char_to_ascii(buf[i]));
    }
    printf("\n");
    printf("------------------- END ----------------------------\n");
    printf("\n");
    return ;
}
/*主函数*/
void main()
{
    if (!nids_init())                                          /*Libnids 初始化*/
    {
        printf("%s\n", nids_errbuf);
        exit(1);
    }
    nids_register_udp(udp_callback);                           /*注册回调函数*/
    nids_run();                                                /*进入循环捕获数据包状态*/
}
```

在 Visual C++ 6.0 中编译此程序，其编译过程同 7.4.1 节。

此程序的功能是显示 UDP 协议数据，运行此程序并进行测试，其输出内容节选一部分如下。

```
------------------- BEGIN ----------------------------
192.168.0.2 : 1025 -> 192.168.0.211 : 20
----------  IP 协议首部  ----------
IP 版本:4
首部长度:20
```

```
TOS:0
总长度:38
标识:342
偏移:0
生存时间:64
协议:17
上层协议为 UDP
校验和:63407
源 IP 地址:192.168.0.2
目的 IP 地址:192.168.0.211
-----------  UDP 协议首部  ----------
源端口:1025
目的端口:20
长度:18
校验和:29839
----------  UDP 数据包内容  ----------
0123456789
----------------------- END -----------------------
```

运行结果如图 7-13 所示。

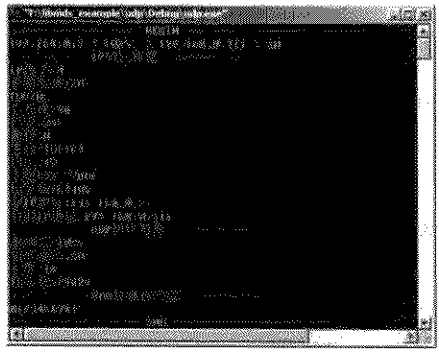

图 7-13　显示 UDP 结果

从运行结果分析，此程序分析了所有的 UDP 协议、IP 协议首部、UDP 首部、UDP 数据内容，并显示了数据包的内容。

7.4.3　HTTP 协议分析

超文本传输协议（HTTP，Hyper Text Transfer Protocol）是基于 TCP 传输协议的，利用 Libnids 的 TCP 数据流重组功能，可以实现对 HTTP 协议的分析。HTTP 协议属于应用层的协议，它包括的内容非常广泛，在此利用 Libnids 开发包实现对 HTTP 协议最常用功能的分析。

HTTP 协议用来实现 Web 服务器和 Web 浏览器之间的通信，其设计简单而灵活，它是

基于客户端/服务器模型的，使用 HTTP 可以传输任何类型的数据，它是一个无连接的协议，每次连接只限处理一个请求。HTTP 协议是一个无状态的协议，其系统运行效率高，服务器应答速度快，但如果后续处理需要用到前面的信息，则必须重传，这样可能导致每次连接传送的数据量增大。HTTP/1.1 由 RFC2626 文档描述。

HTTP 协议的运行基本过程主要有四步：首先浏览器与服务器建立连接；其次浏览器向服务器发送请求信息；然后服务器对请求做出应答，并发送给浏览器；最后数据传送完毕，关闭浏览器和服务器之间的连接。

其中最重要的是浏览器发送请求消息给服务器。请求消息包括的内容非常广泛，其中一般有请求行、普通头、HTTP 请求头、实体头和实体。所有请求消息都由 1 个请求行、0 个或多个头部行和 1 个回车换行组成。其中，头部行可以是普通头、HTTP 请求头或实体头，头部行之后是回车换行，回车换行之后可能有一个实体，也可以没有实体。回车换行表示实体的开始。

HTTP 请求头中包括请求方法，HTTP 请求方法有 GET，HEAD，POST，PUT，DELETE，TRACE，CONNECT。但是最常用的方法是 GET，HEAD 和 POST。

浏览器向服务器发送一个请求，称为请求行，请求行的使用方式如下：

```
Method SP Request-URI SP HTTP-Version CRLF
```

其中 Method 表示请求方法。例如，GET 表示请求指定的页面信息，并返回实体主体；HEAD 表示只请求页面的首部；从客户机向服务器传送数据，在要求服务器和 CGI 做进一步处理时会用到 POST 方法，它主要用于发送 HTML 文本中 FORM 的内容，让 CGI 程序处理。Request-URI 表示资源。HTTP-Version 表示 HTTP 的版本，一般是 HTTP/1.1

请求头（request-header）的内容很多，重要有以下几种。

- Accept：表示浏览器可以接收的数据类型，如果是*/*，则表示能够接收所有类型的数据；
- Accept-Charset：表示可以接收的字符集合；
- Accept-Encoding：表示可以接收的编码；
- Accept-Language：表示浏览器使用的语言；
- Authorization：表示向服务器提供旁路安全保护和加密机制，客户向服务器发认证信息；
- Expect：表示期望值，指出客户期望服务器做什么；
- From：表示浏览器希望服务器提供电子邮件地址；
- Host：表示请求的主机；
- If-Match：用实体标签从一组实体中确定当前的实体；
- If-Modified-Since：表示是否被修改；
- If-None-Match：用于验证文件，与当前的实体都不匹配；
- If-Range：实体标签，它包含一个惟一的标识字符串；
- If-Unmodified-Since：表示是否没有被修改，如果没变，则执行指定的请求操作；
- Max-Forwards：限制请求所能经历的网关和代理数；
- Proxy-Authorization：表示代理-授权。客户向代理服务器发认证信息；
- Range：表示所请求的资源的范围；
- Referer：表示转移地址；

- TE：表示愿意接收的扩展转换编码及是否接收 trailer 域；
- User-Agent：表示浏览器信息。

浏览器发送给服务器请求信息后，服务器及时做出应答。服务器应答向浏览器返回的响应消息一般包括状态行、普通头、响应头、实体头和实体。

状态行的格式如下：

```
HTTP-Version SP Status-Code SP Reason-Phrase CRLF
```

其中 HTTP-Version 表示 HTTP 协议版本；Status-Code 表示状态码。不同的状态码表示不同的含义。状态码是个三位数字码，分为五类，以 1 开头的表示信息响应类，以 2 开头的表示请求被成功处理，以 3 开头的表示请求被重定向，以 4 开头的表示客户的请求有错，以 5 开头的表示服务器不能满足请求。其详细描述如下。

100：表示继续；
101：表示交换协议；
200：表示成功；
201：表示已经创建；
202：表示接收；
203：表示非认证信息；
204：表示无内容；
205：表示重置内容；
206：表示部分内容；
300：表示多路选择；
301：表示永久转移；
302：表示暂时转移；
303：表示参见其他；
304：表示未修改；
305：表示使用代理；
400：表示错误请求；
401：表示未认证；
402：表示需要付费；
403：表示禁止；
404：表示未发现；
405：表示方法不允许；
406：表示不接受；
407：表示需要代理认证；
408：表示请求超时；
409：表示冲突；
410：表示失败；
411：表示需要长度；
412：表示条件失败；
413：表示请求实体太大；
414：表示请求 URI 太长；

415：表示不支持的媒体类型；

500：表示服务器内部错误；

501：表示未实现；

502：表示网关失败；

504：表示网关超时；

505：表示 HTTP 版本不支持。

响应头的内容有如下几种：

- Accept-Ranges：表示接收的范围；
- Age：表示从响应产生以后的时间；
- Etag：实体标签，它包含一个惟一的标识字符串；
- Location：表示重定向接收者到一个新 URI 地址；
- Proxy-Authenticate：表示代理服务器要求的认证方案和参数；
- Retry-After：服务器太忙时返回的字段；
- Server：表示 WEB 服务器类型；
- Vary：对指定的请求标题，沿途的缓存是否可对后续请求直接响应；
- WWW-Authenticate：表示代理-鉴别，通知客户端需要的认证信息；

无论是客户端的请求信息还是服务器端的应答信息，都可能包括普通头、实体头和实体。其中普通头内容如下。

- Cache-Control：表示是否使用缓存；
- Connection：表示连接状态，是持久连接还是关闭连接；
- Date：表示时间，用格林威治时间表示；
- Pragma：包含特定的参数；
- Trailer：表示在 Chunked-Body 中要不要 Trailer；
- Transfer-Encoding：说明采用的编码变换类型，消息体所用的消息转换方法；
- Upgrade：指出可接受的附加通信协议；
- Via：网关和代理指出客户与服务器之间经历的协议和接收者；
- Warning：承载状态或消息变换的附加信息。

其中实体头的内容如下。

- Allow：表示支持的方法类型；
- Content-Encoding：表示支持的编码；
- Content-Language：表示支持的语言；
- Content-Length：表示内容长度；
- Content-Location：表示请求的实体有多个变体时用于提供变体的 URI；
- Content-MD5：MD5 校验值；
- Content-Range：用于指定整个实体中某一部分的插入位置，表示整个实体的长度；
- Content-Type：表示内容类型，为数据设定称为“MIME-TYPE”的内容类型。

其中 MIME，即多任务因特网邮件扩充（Multipurpose Internet Mail Extension），是设定文档内容的一个方法。一些常见的内容类型如下：

- text/html：表示 HTML 文档；
- text/plain：表示纯文本文件；

```
{
    char temp[1024];
    char str1[1024];
    char str2[1024];
    char str3[1024];
    int i;
    int k;
    int j;
    char entity_content[1024];
    if (content[0] != 'H' && content[1] != 'T' && content[2] !- 'T' && content[3] != 'P')
    {
        printf("实体内容为（续）: \n");
        for (i = 0; i < number; i++)
        {
            printf("%s", char_to_ascii(content[i]));
        }
        printf("\n");
    }
    else
    {
        for (i = 0; i < strlen(content); i++)
        {
            if (content[i] != '\n')
            {
                k++;
                continue;
            }
            for (j = 0; j < k; j++)
                temp[j] = content[j + i - k];
            temp[j] = '\0';
            if (strstr(temp, "HTTP"))
            {
                printf("状态行为: ");
                printf("%s\n", temp);
                sscanf(temp, "%s %s %s", str1, str2);
                printf("HTTP 协议为:%s\n", str1);
                printf("状态代码为:%s\n", str2);
            }
            if (strstr(temp, "Date"))
            {
                printf("当前的时间为（Date）:%s\n", temp + strlen("Date:"));
                printf("%s\n", temp);
            }
```

```
if (strstr(temp, "Server"))
{
    printf("服务器为 (Server) :%s\n", temp + strlen("Server:"));
    printf("%s\n", temp);
}
if (strstr(temp, "Cache-Control"))
{
    printf("缓存机制为 (Cache-Control) :%s\n", temp + strlen("Cache-Control:"));
    printf("%s\n", temp);
}
if (strstr(temp, "Expires"))
{
    printf("资源期限为 (Expires) :%s\n", temp + strlen("Expires:"));
    printf("%s\n", temp);
}
if (strstr(temp, "Last-Modified"))
{
    printf("最后一次修改的时间为 (Last-Modified) :%s\n", temp +
strlen("Last-Modified:"));
    printf("%s\n", temp);
}
if (strstr(temp, "ETag"))
{
    printf("Etag 为 (ETag) :%s\n", temp + strlen("Etag:"));
    printf("%s\n", temp);
}
if (strstr(temp, "Accept-Ranges"))
{
    printf("Accept-Ranges (Accept-Ranges) :%s\n", temp + strlen("Accept-Ranges:"));
    printf("%s\n", temp);
}
if (strstr(temp, "Content-Length"))
{
    printf("内容长度为 (Content-Length) :%s\n", temp + strlen("Content-Length:"));
    printf("%s\n", temp);
}
if (strstr(temp, "Connection"))
{
    printf("连接状态为 (Connection) :%s\n", temp + strlen("Connection:"));
    printf("%s\n", temp);
}
if (strstr(temp, "Content-Type"))
{
```

```
                printf("内容类型为 (Content-Type) :%s\n", temp + strlen("Content-Type:"));
                printf("%s\n", temp);
            }
            /*获取实体内容*/
            if ((content[i] == '\n') && (content[i + 1] == '\r'))
            {
                if (i + 3 == strlen(content))
                {
                    printf("无实体内容\n");
                    break;
                }
                for (j = 0; j < number - i - 3; j++)
                    entity_content[j] = content[i + 3 + j];
                entity_content[j] = '\0';
                printf("实体内容为: \n");
                for (i = 0; i < j; i++)
                {
                    printf("%s", char_to_ascii(entity_content[i]));
                }
                printf("\n");
                break;
            }
            k = 0;
        }
    }
}
/*下面的函数是对WEB服务器接收到的数据进行分析*/
void parse_server_data(char content[], int number)
{
    char temp[1024];
    char str1[1024];
    char str2[1024];
    char str3[1024];
    int i;
    int k;
    int j;
    char entity_content[1024];
    for (i = 0; i < strlen(content); i++)
    {
        if (content[i] != '\n')
        {
            k++;
            continue;
```

```
}
for (j = 0; j < k; j++)
    temp[j] = content[j + i - k];
temp[j] = '\0';
if (strstr(temp, "GET"))
{
    printf("请求行为: ");
    printf("%s\n", temp);
    sscanf(temp, "%s %s %s", str1, str2, str3);
    printf("使用的命令为:%s\n", str1);
    printf("获得的资源为:%s\n", str2);
    printf("HTTP 协议类型为:%s\n", str3);
}
if (strstr(temp, "Accept:"))
{
    printf("接收的文件包括 (Accept:) :%s\n", temp + strlen("Accept:"));
    printf("%s\n", temp);
}
if (strstr(temp, "Referer"))
{
    printf("转移地址为 (Referer) :%s\n", temp + strlen("Referer:"));
    printf("%s\n", temp);
}
if (strstr(temp, "Accept-Language"))
{
    printf("使用的语言为 (Accept-Language) :%s\n", temp + strlen("Accept-Language:"));
    printf("%s\n", temp);
}
if (strstr(temp, "Accept-Encoding"))
{
    printf("接收的编码方式为 (Accept-Encoding) :%s\n", temp + strlen("Accept-Encoding:"));
    printf("%s\n", temp);
}
if (strstr(temp, "If-Modified-Since"))
{
    printf("上次修改时间为 (If-Modified-Since) :%s\n", temp + strlen("If-Modified-Since:"));
    printf("%s\n", temp);
}
if (strstr(temp, "If-None-Match"))
{
    printf("If-None-Match 为 (If-Modified-Since) :%s\n", temp + strlen("If-None-Match:"));
    printf("%s\n", temp);
}
```

```
char content[65535];
char content_urgent[65535];
struct tuple4 ip_and_port = tcp_http_connection->addr;
strcpy(address_content, inet_ntoa(*((struct in_addr *) &(ip_and_port.saddr))));
sprintf(address_content + strlen(address_content), " : %i", ip_and_port.source);
strcat(address_content, " <----> ");
strcat(address_content, inet_ntoa(*((struct in_addr *) &(ip_and_port.daddr))));
sprintf(address_content + strlen(address_content), " : %i", ip_and_port.dest);
strcat(address_content, "\n");
if (tcp_http_connection->nids_state == NIDS_JUST_EST)
{
    if (tcp_http_connection->addr.dest != 80)            /*只捕获HTTP协议数据包*/
    {
        return;
    }
    tcp_http_connection->client.collect++;               /*浏览器接收数据*/
    tcp_http_connection->server.collect++;               /*Web服务器端接收数据*/
    printf("\n\n\n==========================================\n");
    printf("%s 建立连接...\n", address_content);
    return;
}
if (tcp_http_connection->nids_state == NIDS_CLOSE)
{
    printf("---------------------------------------\n");
    printf("%s 连接正常关闭...\n", address_content);       /*连接正常关闭*/
    return;
}
if (tcp_http_connection->nids_state == NIDS_RESET)
{
    printf("---------------------------------------\n");
    printf("%s 连接被RST关闭...\n", address_content);     /*连接被RST关闭*/
    return;
}
if (tcp_http_connection->nids_state == NIDS_DATA)
{
    struct half_stream* hlf;
    if (tcp_http_connection->client.count_new)           /*浏览器接收数据*/
    {
        hlf = &tcp_http_connection->client;              /*hlft表示浏览器接收的数据*/
        strcpy(address_content, inet_ntoa(*((struct in_addr *) &(ip_and_port.saddr))));
        sprintf(address_content + strlen(address_content), ":%i", ip_and_port.source);
        strcat(address_content, " <---- ");
        strcat(address_content, inet_ntoa(*((struct in_addr *) &(ip_and_port.daddr))));
```

```
            sprintf(address_content + strlen(address_content), ":%i", ip_and_port.dest);
            strcat(address_content, "\n");
            printf("\n");
            printf("%s", address_content);
            printf("浏览器接收数据...\n");
            printf("\n");
            memcpy(content, hlf->data, hlf->count_new);
            content[hlf->count_new] = '\0';
            parse_client_data(content, hlf->count_new);
            /*分析浏览器接收的数据*/
        }
        else
        {
            hlf = &tcp_http_connection->server;
            /*hlf 表示Web 服务器的TCP 连接端*/
            strcpy(address_content, inet_ntoa(*((struct in_addr *) &(ip_and_port.saddr))));
            sprintf(address_content + strlen(address_content), " : %i", ip_and_port.source);
            strcat(address_content, " ----> ");
            strcat(address_content, inet_ntoa(*((struct in_addr *) &(ip_and_port.daddr))));
            sprintf(address_content + strlen(address_content), ":%i", ip_and_port.dest);
            strcat(address_content, "\n");
            printf("\n");
            printf("%s", address_content);
            printf("服务器接收数据...\n");
            printf("\n");
            memcpy(content, hlf->data, hlf->count_new);
            content[hlf->count_new] = '\0';
            parse_server_data(content, hlf->count_new);     /*分析WEB 服务器接收的数据*/
        }
    }
    return ;
}
/*主函数*/
void main()
{
    if (!nids_init())                                       /*Libnids 初始化*/
    {
        printf("出现错误: %s\n", nids_errbuf);
        exit(1);
    }
    nids_register_tcp(http_protocol_callback);              /*注册回调函数*/
    nids_run();                                             /*进入循环捕获数据包状态*/
}
```

用 Visual C++ 6.0 编译此程序，其编译过程同 7.4.1 节。

运行此程序。为了进行测试，应建立测试环境，为此作者建立了一个 Web 服务器，然后在另一台机子通过浏览器访问此 Web 服务器，其输出内容如下。

```
=================================================
192.168.0.212 : 1864 <----> 192.168.0.211 : 80
 建立连接...
192.168.0.212 : 1864 ----> 192.168.0.211:80
服务器接收数据...
接收的文件包括（Accept:）: */*
Accept: */*
转移地址为（Referer）: http://192.168.0.211/book/
Referer: http://192.168.0.211/book/
使用的语言为（Accept-Language）: zh-cn
Accept-Language: zh-cn
接收的编码方式为（Accept-Encoding）: gzip, deflate
Accept-Encoding: gzip, deflate
上次修改时间为（If-Modified-Since）: Thu, 26 May 2005 11:48:16 GMT
If-Modified-Since: Thu, 26 May 2005 11:48:16 GMT
If-None-Match 为（If-Modified-Since）: "a29c1dcee861c51:8b6"
If-None-Match: "a29c1dcee861c51:8b6"
用户的浏览器信息为（User-Agent）: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT
5.1; SV1)
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1)
访问的主机为（Host）: 192.168.0.211
Host: 192.168.0.211
连接状态为（Connection）: Keep-Alive
Connection: Keep-Alive
Cookie 为（Cookie）: ASPSESSIONIDGGQGGTTC=FFIBJHNCFPDKCKAAGIIEIDGC
Cookie: ASPSESSIONIDGGQGGTTC=FFIBJHNCFPDKCKAAGIIEIDGC
无实体内容
192.168.0.212:1864 <---- 192.168.0.211:80
浏览器接收数据...
服务器为（Server）: Microsoft-IIS/5.0
Server: Microsoft-IIS/5.0
当前的时间为（Date）: Thu, 26 May 2005 13:54:31 GMT
Date: Thu, 26 May 2005 13:54:31 GMT
Etag 为（ETag）: "a29c1dcee861c51:8b6"
ETag: "a29c1dcee861c51:8b6"
内容长度为（Content-Length）: 0
Content-Length: 0
无实体内容
--------------------------------
192.168.0.212 : 1864 <----> 192.168.0.211 : 80
```

程序运行结果如图 7-14 所示。

图 7-14　分析 HTTP 的运行结果

从运行结果进行分析，此程序对 HTTP 协议进行了详细的分析，包括请求访问和应答的内容，实体内容，HTTP 服务的建立，HTTP 协议之间的传输过程，以及 HTTP 服务的关闭。

7.4.4　FTP 协议分析

文件传输协议（FTP，File Transfer Protocol）是 Internet 的传统服务之一。FTP 使用户能在两个连网的计算机之间传输文件，它是 Internet 传递文件最主要的方法。使用匿名 FTP，用户可以免费获取丰富的 Internet 资源。除此之外，FTP 还提供登录、目录查询、文件操作及其他会话控制功能。

在此不详细介绍 FTP 协议的运行机理，而主要讨论 FTP 协议的请求命令和应答状态，这是 FTP 协议操作的主要内容。

FTP 协议的请求命令非常多，大致分为请求控制命令、传输参数命令和 FTP 服务命令。在 FTP 协议中命令是用命令码来表示的，由 FTP 客户端发送给服务器端，不同的命令代表不同的意义，服务器就根据 FTP 命令来进行操作。常用的命令如下：

- USER：表示用户名；
- PASS：表示口令；

- ACCOUNT：表示账户；
- CWD：表示改变工作目录；
- CDUP：表示返回上一层目录；
- SMNT：表示结构加载；
- REIN：表示重新初始化；
- QUIT：表示退出登陆；
- PORT：表示数据端口；
- PASV：表示被动接收请求；
- TYPE：表示类型；
- STRU：表示文件结构，共有三种：F 表示文件结构，R 表示记录结构，P 表示页结构；
- MODE：表示传输模式，共有三种：S 表示流模式，B 表示块模式，C 表示压缩模式；
- RETR：表示从服务器获取文件的命令；
- STOR：表示保存文件；
- STOU：表示惟一保存；
- APPE：表示附加数据到指定文件；
- ALLO：表示分配存储空间；
- REST：表示重新开始的地址；
- RNFR：表示重新命名，需要和 RNTO 命令一起操作；
- RNTO：表示重新命名为一个指定的文件名；
- ABOR：表示终止数据传送；
- DELE：表示删除指定的文件；
- RMD：表示删除指定的目录；
- MKD：表示创建新的目录；
- PWD：表示输出当前目录；
- LIST：表示输出文件列表；
- NLST：表示名字列表；
- SITE：表示服务器信息；
- SYST：表示服务器的操作系统；
- STAT：表示服务器的连接状态；
- HELP：表示帮助命令；
- NOOP：表示空操作。

当 FTP 客户端向 FTP 服务器端发送 FTP 命令之后，服务器端就会对此命令做出响应，然后把响应信息发送给客户端。这样，客户端就可以知道 FTP 命令的操作状态：成功或失败。FTP 应答由一个三位数字来决定。其中第一位表示响应成功，或失败；第二位表示表示具体的信息；第三位保留表示其他信息。下面是 FTP 服务器应答信息的说明。

- 110：表示启动标记应答，在这种情况下文本是确定的，它必须是：MARKyyyy=mmmm，其中 yyyy 是用户进程数据流标记，mmmm 是服务器标记；
- 120：表示服务在 nnn 分钟内可用；

```
#include "nids.h"
char ascii_string[10000];
char* char_to_ascii(char ch)
{
    char* string;
    ascii_string[0] = 0;
    string = ascii_string;
    if (isgraph(ch))
        *string++ = ch;
    else if (ch == ' ')
        *string++ = ch;
    else if (ch == '\n' || ch == '\r')
        *string++ = ch;
    else
        *string++ = '.';
    *string = 0;
    return ascii_string;
}
/*下面是分析 FTP 协议的回调函数*/
void ftp_protocol_callback(struct tcp_stream* ftp_connection, void** arg)
{
    int i;
    char address_string[1024];
    char content[65535];
    char content_urgent[65535];
    struct tuple4 ip_and_port = ftp_connection->addr;
    strcpy(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.saddr))));
    sprintf(address_string + strlen(address_string), " : %i", ip_and_port.source);
    strcat(address_string, " <---> ");
    strcat(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.daddr))));
    sprintf(address_string + strlen(address_string), " : %i", ip_and_port.dest);
    strcat(address_string, "\n");
    switch (ftp_connection->nids_state)
    {
    case NIDS_JUST_EST:
        if ((ftp_connection->addr.dest == 21) || (ftp_connection->addr.source == 20))
        {
            /* FTP 客户端和 FTP 服务器端建立连接 */
            ftp_connection->client.collect++;          /*FTP 客户端接收数据*/
            ftp_connection->server.collect++;          /*FTP 服务器接收数据*/
            ftp_connection->server.collect_urg++;      /*FTP 服务器接收紧急数据*/
            ftp_connection->client.collect_urg++;      /*FTP 客户端接收紧急数据*/
            if (ftp_connection->addr.dest == 21)
```

```
            printf("%s FTP客户端与FTP服务器建立控制连接\n", address_string);
        if (ftp_connection->addr.source == 20)
            printf("%s FTP服务器与FTP客户端建立数据连接\n", address_string);
    }
    return;
case NIDS_CLOSE:                    /*FTP客户端与FTP服务器端连接正常关闭*/
    printf("--------------------------------\n");
    if (ftp_connection->addr.dest == 21)
        printf("%sFTP客户端与FTP服务器的控制连接正常关闭\n", address_string);
    if (ftp_connection->addr.source == 20)
        printf("%sFTP服务器与FTP客户端的数据连接正常关闭\n", address_string);
    return;
case NIDS_RESET:                    /*FTP客户端与FTP服务器端连接被RST关闭*/
    printf("--------------------------------\n");
    if (ftp_connection->addr.source == 20)
        printf("%sFTP服务器与FTP客户端的数据连接被RESET关闭\n", address_string);
    if (ftp_connection->addr.dest == 21)
        printf("%sFTP客户端与FTP服务器的控制连接被REST关闭\n", address_string);
    return;
case NIDS_DATA:
    {
        /*FTP协议有新的数据达到*/
        struct half_stream* hlf;
        if (ftp_connection->server.count_new_urg)
        {
            /*FTP服务器接收到新的紧急数据*/
            printf("--------------------------------\n");
            strcpy(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.saddr))));
            sprintf(address_string + strlen(address_string), " : %i", ip_and_port.source);
            strcat(address_string, " urgent---> ");
            strcat(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.daddr))));
            sprintf(address_string + strlen(address_string), " : %i", ip_and_port.dest);
            strcat(address_string, "\n");
            address_string[strlen(address_string) + 1] = 0;
            address_string[strlen(address_string)] = ftp_connection->server.urgdata;
            printf("%s", address_string);
            return;
        }
        if (ftp_connection->client.count_new_urg)
        {
            /*FTP客户端接收到新的紧急数据*/
            printf("--------------------------------\n");
            strcpy(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.saddr))));
```

```
        sprintf(address_string + strlen(address_string), " : %i", ip_and_port.source);
        strcat(address_string, " <--- urgent ");
        strcat(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.daddr))));
        sprintf(address_string + strlen(address_string), " : %i", ip_and_port.dest);
        strcat(address_string, "\n");
        address_string[strlen(address_string) + 1] = 0;
        address_string[strlen(address_string)] = ftp_connection->client.urgdata;
        printf("%s", address_string);
        return;
    }
    if (ftp_connection->client.count_new)
    {
        /*FTP 客户端接收到新的数据*/
        hlf = &ftp_connection->client;
        strcpy(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.saddr))));
        sprintf(address_string + strlen(address_string), ":%i", ip_and_port.source);
        strcat(address_string, " <--- ");
        strcat(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.daddr))));
        sprintf(address_string + strlen(address_string), ":%i", ip_and_port.dest);
        strcat(address_string, "\n");
        printf("--------------------------------\n");
        printf("%s", address_string);          /*输出 FTP 客户端接收到的新的数据*/
        memcpy(content, hlf->data, hlf->count_new);
        content[hlf->count_new] = '\0';
        if (ftp_connection->addr.source == 20)
        {
            printf("传输的数据为:\n");
            for (i = 0; i < hlf->count_new; i++)
            {
                printf("%s", char_to_ascii(content[i]));
            }
            printf("\n");
        }
        else
        {
            if (content[0] == '1' || content[0] == '2' ||
                    content[0] == '3' || content[0] == '4' || content[0] == '5')
                printf("FTP 服务器响应状态代码为: %c%c%c\n",
                                content[0], content[1], content[2]);
            if (strncmp(content, "220", 3) == 0)
                printf("新连接的用户的服务已经准备就绪\n");
            if (strncmp(content, "110", 3) == 0)
                printf("启动标记应答\n");
```

```
if (strncmp(content, "120", 3) == 0)
    printf("表示 服务在nnn分钟内可用\n");
if (strncmp(content, "125", 3) == 0)
    printf("表示数据连接已打开，准备传送\n");
if (strncmp(content, "150", 3) == 0)
    printf("表示文件状态正确，正在打开数据连接\n");
if (strncmp(content, "200", 3) == 0)
    printf("表示命令正常执行\n");
if (strncmp(content, "202", 3) == 0)
    printf("表示命令未被执行，此站点不支持此命令\n");
if (strncmp(content, "211", 3) == 0)
    printf("表示系统状态或系统帮助响应\n");
if (strncmp(content, "212", 3) == 0)
    printf("表示目录状态信息\n");
if (strncmp(content, "213", 3) == 0)
    printf("表示文件状态信息\n");
if (strncmp(content, "214", 3) == 0)
    printf("表示帮助信息\n");
if (strncmp(content, "215", 3) == 0)
    printf("表示名字系统类型\n");
if (strncmp(content, "221", 3) == 0)
    printf("表示控制连接已经被关闭\n");
if (strncmp(content, "225", 3) == 0)
    printf("表示数据连接已经打开，没有数据传输\n");
if (strncmp(content, "226", 3) == 0)
    printf("表示数据连接已经关闭，请求文件操作成功完成\n");
if (strncmp(content, "227", 3) == 0)
    printf("表示进入被动模\n");
if (strncmp(content, "230", 3) == 0)
    printf("表示用户已经登录\n");
if (strncmp(content, "250", 3) == 0)
    printf("表示请求文件操作已经成功执行\n");
if (strncmp(content, "257", 3) == 0)
    printf("表示创建路径名字\n");
if (strncmp(content, "331", 3) == 0)
    printf("表示用户名正确，需要输入密码\n");
if (strncmp(content, "332", 3) == 0)
    printf("表示 登录时需要账户信息\n");
if (strncmp(content, "350", 3) == 0)
    printf("表示对请求的文件操作需要更多的指令\n");
if (strncmp(content, "421", 3) == 0)
    printf("表示服务不可用，关闭控制连接\n");
if (strncmp(content, "425", 3) == 0)
```

```
            printf("表示打开数据连接操作失败\n");
        if (strncmp(content, "426", 3) == 0)
            printf("表示关闭连接，终止传输\n");
        if (strncmp(content, "450", 3) == 0)
            printf("表示请求的文件操作未被执行\n");
        if (strncmp(content, "451", 3) == 0)
            printf("表示请求操作终止，有本地错误发生\n");
        if (strncmp(content, "452", 3) == 0)
            printf("表示未执行请求的操作，系统存储空间不足，文件不可用\n");
        if (strncmp(content, "500", 3) == 0)
            printf("表示语法错误，命令错误\n");
        if (strncmp(content, "501", 3) == 0)
            printf("表示参数的语法错误\n");
        if (strncmp(content, "502", 3) == 0)
            printf("表示命令未被执行\n");
        if (strncmp(content, "503", 3) == 0)
            printf("表示命令顺序发生错误\n");
        if (strncmp(content, "504", 3) == 0)
            printf("表示由于参数而发生错误命令\n");
        if (strncmp(content, "530", 3) == 0)
            printf("表示未登录\n");
        if (strncmp(content, "532", 3) == 0)
            printf("表示存储文件需要账户信息\n");
        if (strncmp(content, "550", 3) == 0)
            printf("表示未执行请求的操作，文件不可用\n");
        if (strncmp(content, "551", 3) == 0)
            printf("表示请求操作终止，页面类型未知\n");
        if (strncmp(content, "552", 3) == 0)
            printf("表示请求的文件操作终止，超出存储分配空间\n");
        if (strncmp(content, "553", 3) == 0)
            printf("表示未执行请求的操作，文件名不合法\n");
        for (i = 0; i < hlf->count_new; i++)
        {
            printf("%s", char_to_ascii(content[i]));
        }
        printf("\n");
    }
}
else
{
    /*FTP服务器接收到新的数据*/
    hlf = &ftp_connection->server;
    strcpy(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.saddr))));
```

```
sprintf(address_string + strlen(address_string), ":%i", ip_and_port.source);
strcat(address_string, " ---> ");
strcat(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.daddr))));
sprintf(address_string + strlen(address_string), ":%i", ip_and_port.dest);
strcat(address_string, "\n");
printf("--------------------------------\n");
printf("%s", address_string);
/* 输出FTP服务器端接收到的新的数据 */
memcpy(content, hlf->data, hlf->count_new);
content[hlf->count_new] = '\0';
if (ftp_connection->addr.source == 20)
    printf("FTP服务器向FTP客户端发送数据\n");
else
{
    if (strstr(content, "USER"))
        printf("用户名字为(USER):%s\n", content + strlen("USER"));
    else if (strstr(content, "PASS"))
        printf("用户密码为(PASS):%s\n", content + strlen("PASS"));
    else if (strstr(content, "PORT"))
        printf("端口参数为(PORT):%s\n", content + strlen("PORT"));
    else if (strstr(content, "LIST"))
        printf("显示文件列表(LIST):%s\n", content + strlen("LIST"));
    else if (strstr(content, "CWD"))
        printf("改变工作目录为(CWD):%s\n", content + strlen("CWD"));
    else if (strstr(content, "TYPE"))
        printf("类型为(TYPE):%s\n", content + strlen("TYPE"));
    else if (strstr(content, "RETR"))
        printf("获取文件为(RETR):%s\n", content + strlen("RETR"));
    else if (strstr(content, "STOR"))
        printf("保存文件为(STOR):%s\n", content + strlen("STOR"));
    else if (strstr(content, "XRMD"))
        printf("删除目录(XRMD):%s\n", content + strlen("XRMD"));
    else if (strstr(content, "QUIT"))
        printf("退出登陆(QUIT):%s\n", content + strlen("QUIT"));
    else
        printf("FTP客户端使用的命令为 %c%c%c%c\n",
        content[0], content[1], content[2], content[3]);
}
for (i = 0; i < hlf->count_new; i++)
{
    printf("%s", char_to_ascii(content[i]));
}
printf("\n");
```

```
            }
        }
    default:
        break;
    }
    return;
}                                                   /*主函数*/
void main()
{
    if (!nids_init())                               /*Libnids 初始化*/
    {
        printf("出现错误: %s\n", nids_errbuf);
        exit(1);
    }
    nids_register_tcp(ftp_protocol_callback);       /*注册回调函数*/
    nids_run();                                     /*进入循环捕获数据包状态*/
}
```

在 Viusal C++ 6.0 中进行编译，编译过程同 7.4.1 节相关内容。

为了对此程序的运行结果进行分析，读者建立了一个 FTP 服务器，使用 Serv-U FTP Server 软件来建立 FTP 服务器，然后在另外一台机子上访问此 FTP 服务器。在这之前运行此程序，对访问过程进行监控，输出内容如下：（当然，输出内容会随着 FTP 操作的不容而不同，这只是其中的一种输出情况。）

```
192.168.0.212 : 1644 <---> 192.168.0.211 : 21
 FTP 客户端与 FTP 服务器建立控制连接
--------------------------------
192.168.0.212:1644 <--- 192.168.0.211:21
FTP 服务器响应状态代码为: 220
新连接的用户的服务已经准备就绪
220 Serv-U FTP Server v4.0 for WinSock ready...
--------------------------------
192.168.0.212:1644 ---> 192.168.0.211:21
用户名字为 (USER) : peter
USER peter
--------------------------------
192.168.0.212:1644 <--- 192.168.0.211:21
FTP 服务器响应状态代码为: 331
表示用户名正确，需要输入密码
331 User name okay, need password.
--------------------------------
192.168.0.212:1644 ---> 192.168.0.211:21
用户密码为 (PASS) : 1234567890
PASS 1234567890
--------------------------------
```

```
192.168.0.212:1644 <--- 192.168.0.211:21
FTP 服务器响应状态代码为：230
表示用户已经登录
230 User logged in, proceed.
--------------------------------
192.168.0.212:1644 ---> 192.168.0.211:21
端口参数为（PORT）：192,168,0,212,6,110
PORT 192,168,0,212,6,110
--------------------------------
192.168.0.212:1644 <--- 192.168.0.211:21
FTP 服务器响应状态代码为：200
表示命令正常执行
200 PORT Command successful.
--------------------------------
192.168.0.212:1644 ---> 192.168.0.211:21
显示文件列表（LIST）：
LIST
192.168.0.211 : 20 <---> 192.168.0.212 : 1646
 FTP 服务器与 FTP 客户端建立数据连接
--------------------------------
192.168.0.212:1644 <--- 192.168.0.211:21
FTP 服务器响应状态代码为：150
表示文件状态正确，正在打开数据连接
150 Opening ASCII mode data connection for /bin/ls.
--------------------------------
192.168.0.211:20 ---> 192.168.0.212:1646
FTP 服务器向 FTP 客户端发送数据
drw-rw-rw-   1 user    group          0 May 26 19:58 .
--------------------------------
192.168.0.211:20 ---> 192.168.0.212:1646
FTP 服务器向 FTP 客户端发送数据
drw-rw-rw-   1 user    group          0 May 26 19:58 ..
drw-rw-rw-   1 user    group          0 May 26 19:59 company
drw-rw-rw-   1 user    group          0 May 26 19:58 保险
-rw-rw-rw-   1 user    group      10752 May 26 19:57 ?号.doc
drw-rw-rw-   1 user    group          0 May 26 19:58 福利
-rw-rw-rw-   1 user    group      10752 May 26 19:57 ?作条例.xls
-rw-rw-rw-   1 user    group      10752 May 26 19:57 规章制度.doc
-rw-rw-rw-   1 user    group      10752 May 26 19:57 员??资.doc
--------------------------------
192.168.0.211 : 20 <---> 192.168.0.212 : 1646
FTP 服务器与 FTP 客户端的数据连接正常关闭
--------------------------------
```

```
192.168.0.212:1644 <--- 192.168.0.211:21
FTP 服务器响应状态代码为：226
表示数据连接已经关闭，请求文件操作成功完成
226 Transfer complete.
--------------------------------
192.168.0.212:1644 ---> 192.168.0.211:21
改变工作目录为（CWD）：company
CWD company
--------------------------------
192.168.0.212:1644 <--- 192.168.0.211:21
FTP 服务器响应状态代码为：250
表示请求文件操作已经成功执行
250 Directory changed to /d:/ftp/company
--------------------------------
192.168.0.212:1644 ---> 192.168.0.211:21
端口参数为（PORT）：192,168,0,212,6,111
PORT 192,168,0,212,6,111
--------------------------------
192.168.0.212:1644 <--- 192.168.0.211:21
FTP 服务器响应状态代码为：200
表示命令正常执行
200 PORT Command successful.
--------------------------------
192.168.0.212:1644 ---> 192.168.0.211:21
显示文件列表（LIST）：
LIST
192.168.0.211 : 20 <---> 192.168.0.212 : 1647
 FTP 服务器与 FTP 客户端建立数据连接
--------------------------------
192.168.0.212:1644 <--- 192.168.0.211:21
FTP 服务器响应状态代码为：150
表示文件状态正确，正在打开数据连接
150 Opening ASCII mode data connection for /bin/ls.
--------------------------------
192.168.0.211:20 ---> 192.168.0.212:1647
FTP 服务器向 FTP 客户端发送数据
drw-rw-rw-   1 user    group         0 May 26 19:59 .
--------------------------------
192.168.0.211:20 ---> 192.168.0.212:1647
FTP 服务器向 FTP 客户端发送数据
drw-rw-rw-   1 user    group         0 May 26 19:59 ..
-rw-rw-rw-   1 user    group        44 May 26 19:59 user.txt
--------------------------------
```

```
192.168.0.211 : 20 <---> 192.168.0.212 : 1647
FTP 服务器与 FTP 客户端的数据连接正常关闭
--------------------------------
192.168.0.212:1644 <--- 192.168.0.211:21
FTP 服务器响应状态代码为：226
表示数据连接已经关闭，请求文件操作成功完成
226 Transfer complete.
---------------------------------
192.168.0.212:1644 ---> 192.168.0.211:21
端口参数为（PORT）：192,168,0,212,6,112
PORT 192,168,0,212,6,112
---------------------------------
192.168.0.212:1644 <--- 192.168.0.211:21
FTP 服务器响应状态代码为：200
表示命令正常执行
200 PORT Command successful.
---------------------------------
192.168.0.212:1644 ---> 192.168.0.211:21
获取文件为（RETR）：user.txt
RETR user.txt
192.168.0.211 : 20 <---> 192.168.0.212 : 1648
 FTP 服务器与 FTP 客户端建立数据连接
---------------------------------
192.168.0.212:1644 <--- 192.168.0.211:21
FTP 服务器响应状态代码为：150
表示文件状态正确，正在打开数据连接
150 Opening ASCII mode data connection for user.txt (44 bytes).
---------------------------------
192.168.0.211:20 ---> 192.168.0.212:1648
FTP 服务器向 FTP 客户端发送数据
user:peter
password:123456
content:test
---------------------------------
192.168.0.211 : 20 <---> 192.168.0.212 : 1648
FTP 服务器与 FTP 客户端的数据连接正常关闭
---------------------------------
192.168.0.212:1644 <--- 192.168.0.211:21
FTP 服务器响应状态代码为：226
表示数据连接已经关闭，请求文件操作成功完成
226 Transfer complete.
---------------------------------
192.168.0.212:1644 ---> 192.168.0.211:21
```

```
退出登陆（QUIT）：
QUIT
-----------------------------------------
192.168.0.212:1044 ---- 192.168.0.211:21
FTP 服务器响应状态代码为：221
表示控制连接已经被关闭
221 Goodbye!
-----------------------------------------
192.168.0.212 : 1044 <- -- 192.168.0.211 : 21
FTP 客户端与 FTP 服务器的控制连接正常关闭
```

程序运行结果如图 7-15 所示。

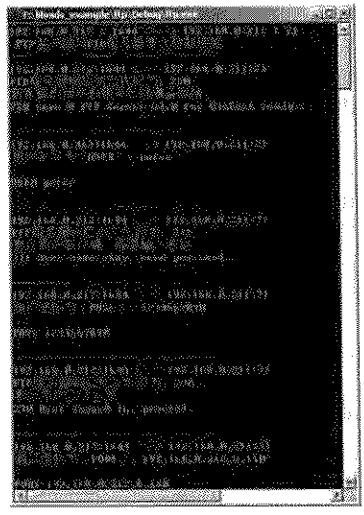

图 7-15　分析 FTP 协议的运行结果

从程序的输出内容来看，使用 Libnids 来分析 FTP 协议，可以显示 FTP 协议的整个连接过程，对于其传输的内容也可以深入分析。通过对内容的解析，可以得到 FTP 的用户名和密码。上述测试例子中传输的用户名为 peter，密码为 1234567890。FTP 客户端通过 RETR 来获得文件内容，然后通过命令 QUIT 来退出 FTP 连接。

7.4.5　Telnet 协议分析

Telnet 协议是 TCP/IP 协议族中的一个应用层协议，是 Internet 远程登录服务的标准协议。应用 Telnet 协议能够把本地用户所使用的计算机变成远程主机系统的一个终端。Telnet 协议的目的是提供一个相对通用的、双向的、面向八位字节的通信方法。

分时系统允许多个用户同时使用一台计算机，为了保证系统的安全和记账方便，系统要求每个用户有单独的账号作为登录标识，还为每个用户指定了一个口令。用户在使用该系统

之前要输入标识和口令，这个过程被称为登录。

远程登录是指用户使用 Telnet 命令，使自己的计算机暂时成为远程主机的一个仿真终端的过程。仿真终端等效于一个非智能的机器，它只负责把用户输入的每个字符传递给主机，再将主机输出的每条信息回显在屏幕上。

为了适应异构环境，Telnet 协议定义了数据和命令在 Internet 上的传输方式，此定义被称为网络虚拟终端（NVT，Network Virtual Terminal）。NVT 是虚拟设备，连接的双方都必须把它们的物理终端和 NVT 进行相互转换。客户端把来自用户终端的按键和命令序列转换为 NVT 格式，并发送到服务器，服务器将收到的数据和命令从 NVT 格式转换为远地系统需要的格式。远程服务器将数据从远地机器的格式转换为 NVT 格式，而本地客户机将接收到的 NVT 格式数据再转换为本地的格式。

Telnet 远程登录服务分为四个阶段：首先，客户端与服务器建立连接；其次，将客户端上输入的用户名和口令及以后输入的任何命令或字符以 NVT 格式传送到远程服务器；再次客户端把远程服务器传回的 NVT 格式的数据转化为本地所接受的格式送回本地终端，其中包括输入命令回显和命令执行结果；最后，客户端与服务器终止连接。

Telnet 通信的两个方向都采用带内信令方式。字节 0xff 称为 IAC（Interpret as Command）。该字节后面的一个字节才是命令字节。如果要发送数据 255，就必须发送两个连续的字节 255。下面是 Telnet 命令的详细介绍。

- EOF：代码 236，表示文件结束符；
- SUSP：代码 237，表示挂起当前进程；
- ABORT：代码 238，表示异常终止进程；
- EOR：代码 239，表示记录结束符；
- SE：代码 240，表示子选项结束；
- NOP：代码 241，表示无操作；
- DM：代码 242，表示数据标记；
- BRK：代码 243，表示中断；
- IP：代码 244，表示中断进程；
- AO：代码 245，表示异常终止输出；
- AYT：代码 246，表示对方是否还在进行；
- EC：代码 247，表示转义字符；
- EL：代码 248，表示删除行；
- GA：代码 249，表示继续进行；
- SB：代码 250，表示子选项开始；
- WILL：代码 251，表示选项协商；
- WONT：代码 252，表示选项协商；
- DO：代码 253，表示选项协商；
- DON'T：代码 254，表示选项协商；
- IAC：代码 255，表示数据字节 255。

Telnet 协议是用于远程登录的，它的过程可以使用 Libnids 来进行分析，利用 Libnids 的 TCP 数据流重组功能，可以分析 Telnet 协议传输的任何数据内容，包括敏感信息，如用户名和密码等。下面举例说明。

```
        }
        return;
    case NIDS_CLOSE:                                        /*Telnet 协议连接正常关闭*/
        printf("--------------------------------\n");
        printf("%sTELNET 客户端与 TELNET 服务器连接正常关闭\n", address_string);
        return;
    case NIDS_RESET:                                        /*Telnet 协议连接被 RST 关闭*/
        printf("--------------------------------\n");
        printf("%sTELNET 客户端与 TELNET 服务器连接被 REST 关闭\n",address_string);
        return;
    case NIDS_DATA:
        {                                                   /*Telnet 协议有新的数据到达*/
            struct half_stream* hlf;
            if (telnet_connection->server.count_new_urg)
            {                                               /*Telnet 服务器接收到新的紧急数据*/
                printf("--------------------------------\n");
                strcpy(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.saddr))));
                sprintf(address_string + strlen(address_string), " : %i", ip_and_port.source);
                strcat(address_string, " urgent---> ");
                strcat(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.daddr))));
                sprintf(address_string + strlen(address_string), " : %i", ip_and_port.dest);
                strcat(address_string, "\n");
                address_string[strlen(address_string) + 1] = 0;
                address_string[strlen(address_string)] = telnet_connection->server.urgdata;
                printf("%s", address_string);
                return;
            }
            if (telnet_connection->client.count_new_urg)
            {                                               /*Telnet 客户端接收到新的紧急数据*/
                printf("--------------------------------\n");
                strcpy(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.saddr))));
                sprintf(address_string + strlen(address_string), " : %i", ip_and_port.source);
                strcat(address_string, " <--- urgent ");
                strcat(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.daddr))));
                sprintf(address_string + strlen(address_string), " : %i", ip_and_port.dest);
                strcat(address_string, "\n");
                address_string[strlen(address_string) + 1] = 0;
                address_string[strlen(address_string)] = telnet_connection->client.urgdata;
                printf("%s", address_string);
                return;
            }
            if (telnet_connection->client.count_new)
            {                                               /*Telnet 客户端接收到新的数据*/
```

```
                hlf = &telnet_connection->client;
                strcpy(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.saddr))));
                sprintf(address_string + strlen(address_string), ":%i", ip_and_port.source);
                strcat(address_string, " <--- ");
                strcat(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.daddr))));
                sprintf(address_string + strlen(address_string), ":%i", ip_and_port.dest);
                strcat(address_string, "\n");
                printf("--------------------------------\n");
                printf("%s", address_string);          /*输出Telnet客户端接收到的新的数据*/
                memcpy(content, hlf->data, hlf->count_new);
                content[hlf->count_new] = '\0';
                for (i = 0; i < hlf->count_new; i++)
                {
                    printf("%s", char_to_ascii(content[i]));
                }
                printf("\n");
            }
            else
            {
                /*Telnet服务器接收到新的数据*/
                hlf = &telnet_connection->server;
                strcpy(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.saddr))));
                sprintf(address_string + strlen(address_string), ":%i", ip_and_port.source);
                strcat(address_string, " ---> ");
                strcat(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.daddr))));
                sprintf(address_string + strlen(address_string), ":%i", ip_and_port.dest);
                strcat(address_string, "\n");
                printf("--------------------------------\n");
                printf("%s", address_string);     /*输出Telnet服务器接收到的新的数据*/
                memcpy(content, hlf->data, hlf->count_new);
                content[hlf->count_new] = '\0';
                for (i = 0; i < hlf->count_new; i++)
                {
                    printf("%s", char_to_ascii(content[i]));
                }
                printf("\n");
            }
        }
    default:
        break;
    }
    return;
}                                                   /*主函数*/
```

```
void main()
{
    if (!nids_init())  /*Libnids 初始化*/
    {
        printf("%s\n", nids_errbuf);
        exit(1);
    }
    nids_register_tcp(telnet_protocol_callback);        /*注册分析 Telnet 协议的回调函数*/
    nids_run();                                         /*进入循环捕获数据包的状态*/
}
```

在 Visual C++ 6.0 中编译此程序，编译过程同 7.4.1 节相关内容。

此程序的功能是显示 Telnet 协议传输的数据内容以及 Telnet 的传输过程。其实，在该程序中，只是对 TCP 数据重组进行了限制，只分析目的端口号是 23 的 TCP 数据流，也就是 Telnet 协议数据流。

运行此程序并进行测试，使用一台 Telnet 客户端计算机连接另一台 Telnet 服务器端计算机，其程序输出内容如下：

```
192.168.0.212 : 1690 <---> 192.168.0.211 : 23
TELNET 客户端与 TELNET 服务器建立连接
-------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
.?.?.?.?.?.?
-------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
.?
-------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
..%.....
-------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
.?.?.?....P.....?.?
-------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
..%.....
 -----     ----------------------
192.168.0.212:1690 <--- 192.168.0.211:23
Microsoft (R) Windows (TM) Version 5.00 (Build 2195)
Welcome to Microsoft Telnet Service
Telnet Server Build 5.00.99206.1
login:
-------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
j
-------------------------------
```

```
192.168.0.212:1690 <--- 192.168.0.211:23
j
--------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
e
--------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
e
--------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
n
--------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
n
--------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
n
--------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
n
--------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
y
--------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
y
--------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
--------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
password:
--------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
l
--------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
*
--------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
2
--------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
*
```

```
--------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
3
--------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
*
--------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
4
--------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
*
--------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
5
--------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
*
--------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
--------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
.?
--------------------------------
192.168.0.212:1690    > 192.168.0.211:23
.?
--------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
......
--------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
....ANSI..
--------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
.[1;1H*==========================================================================
.[2;1H欢迎使用 Microsoft Telnet 服务器.
.[3;1H*==========================================================================
[4;1HC:\.
--------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
d
--------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
```

```
.[4;5Hd.[4;6H
--------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
:
--------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
.[4;6H:.[4;7H
--------------------------------
192.168.0.212:1690 --> 192.168.0.211:23
--------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
.[6;1HD:\>.[6;5H
--------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
c
--------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
.[6;5Hc.[6;6H
--------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
d
--------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
.[6;6Hd.[6;7H
--------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
--------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
.[6;8H
--------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
f
--------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
.[6;8Hf.[6;9H
--------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
t
--------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
.[6;9Ht.[6;10H
--------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
```

```
p
--------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
.[6;10Hp.[6;11H
--------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
--------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
.[8;1HD:\ftp>.[8;8H
--------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
d
--------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
.[8;8Hd.[8;9H
--------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
i
--------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
.[8;9Hi.[8;10H
--------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
r
--------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
.[8;10Hr.[8;11H
--------------------------------
192.168.0.212:1690 ---> 192.168.0.211:23
--------------------------------
192.168.0.212:1690 <--- 192.168.0.211:23
.[1;1H欢迎使用 Microsoft Telnet 服务器。                    .[2;1H*===
==========================================================.[3;1HC:\>d:
                                                .[4;1H.[K.[5;1HD:\>cd ftp.[6;1
H.[K.[7;1HD:\ftp>dir.[8;1H 驱动器 D 中的卷是 file.[9;2H卷的序列号是 F4E9-90FB.[1
1;2HD:\ftp 的目录.[13;1H2005-05-26  19:58      <DIR>          ..[14;1H2005-05-2
6  19:58      <DIR>          ...[15;1H2005-05-26  19:59      <DIR>          co
mpany.[16;1H2005-05-26  19:58      <DIR>          保险.[17;1H2005-05-26  19:57
            10,752 员??资.doc.[18;1H2005-05-26  19:57            10,752 ?作
条例.xls.[19;1H2005-05-26  19:58      <DIR>          福利.[20;1H2005-05-26  19:
57            10,752 编号.doc.[21;1H2005-05-26  19:57            10,752 规
章制度.doc.[22;16H4 个文件        43,008 字节.[23;16H5 个目录  1,304,907,776 可
用字节.[25;1HD:\ftp>.[25;8H
```

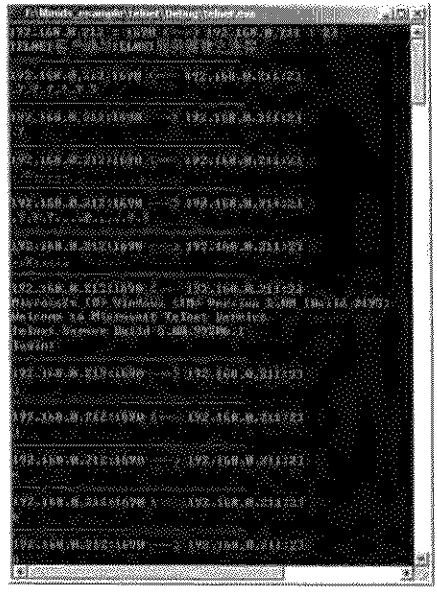

图 7-16　分析 Telnet 协议的运行结果

从程序的运行结果来看，Telnet 客户端 IP 地址为 192.168.0.212，Telnet 服务器端的 IP 地址为 192.168.0.211，用户名为 jenny，密码为 12345；Telnet 客户端发送命令为 cd ftp，服务器发送命令为 dir，退出命令为 exit。整个 Telnet 过程的分析非常清楚。

7.4.6　POP3 协议分析

POP3（Post Office Protocol 3）是适用于客户-服务器结构的脱机模型的电子邮件协议。它规定了怎样将个人计算机连接到 Internet 邮件服务器和下载电子邮件的电子协议，允许用户从服务器上把邮件存储到本地主机上，同时删除保存在邮件服务器上的邮件。POP3 服务器是遵循 POP3 协议的接收邮件服务器，用来接收电子邮件。

在 POP3 协议中有三种状态：认可状态，处理状态和更新状态。

当客户机与服务器建立联系时，一旦客户机提供了自己身份并成功确认，即由认可状态转入处理状态，在完成相应的操作后客户机发出 QUIT 命令，则进入更新状态，更新之后重返认可状态。大多数现有的 POP3 客户机与服务器执行采用 ASCII 明文发送用户名和口令，在认可状态等待客户连接的情况下，客户发出连接，并由命令 user/pass 对在网络上发送明文用户名和口令给服务器进行身份确认。一旦确认成功，便转入处理状态。

POP3 命令由 1 个命令和一些参数组成。所有命令以一个回车换行 CRLF 结束。命令和参数由可打印的 ASCII 字符组成，它们之间由空格隔开。命令一般是 3 到 4 个字母，每个参数却可达 40 个字符长。POP3 响应是由 1 个状态码和 1 个可能跟有附加信息的命令组成。所有响应也是由回车换行 CRLF 结束的。有两种状态码，状态码“+OK”表示成功，状态码

“-ERR”表示失败。

下面对 POP3 命令进行详细介绍。

- USER：参数为 username，在认可状态下有效，表示用户名；
- PASS：参数为 password，在认可状态下有效，表示用户密码，与命令 USER 一起使用，如果操作成功，就导致状态转换；
- APOP：参数为 Name 和 Digest，在认可状态下有效，其中 Digest 是 MD5 消息摘要；
- STAT：无参数，在处理状态下有效，它请求服务器返回关于邮箱的统计资料；
- UIDL：参数为[Msg#]，在处理状态下有效，它返回邮件的惟一标识符；
- LIST：参数为[Msg#]，在处理状态下有效，它返回邮件数量和每个邮件的大小；
- RETR：参数为[Msg#]，在处理状态下有效，它返回由参数标识的邮件的全部文本；
- DELE：参数为[Msg#]，在处理状态下有效，它表示服务器将由参数标识的邮件标记为删除。执行 QUIT 命令后，邮件将删除；
- RSET：无参数，在处理状态下有效，服务器取消所有标记为删除的邮件，用于撤销 DELE 命令；
- TOP：参数为[Msg#]，在处理状态下有效，服务器将返回由参数标识的邮件前 *n* 行内容，*n* 必须是正整数；
- NOOP：无参数，在处理状态下有效，服务器返回一个肯定的响应；
- QUIT：无参数，在更新状态下有效，退出。

本节利用 Libnids 来实现对 POP3 协议的分析，通过分析，可以清楚地查看 POP3 协议的操作过程。其程序源代码如下。

```
#include "nids.h"
char ascii_string[10000];
char* char_to_ascii(char ch)
{
    char* string;
    ascii_string[0] = 0;
    string = ascii_string;
    if (isgraph(ch))
        *string++ = ch;
    else if (ch == ' ')
        *string++ = ch;
    else if (ch == '\n' || ch == '\r')
        *string++ = ch;
    else
        *string++ = '.';
    *string = 0;
    return ascii_string;
}
/*下面是分析 POP3 协议的回调函数*/
void pop3_protocol_callback(struct tcp_stream* pop3_connection, void** arg)
```

```
{
    int i;
    char address_string[1024];
    char content[65535];
    char content_urgent[65535];
    struct tuple4 ip_and_port = pop3_connection->addr;
    strcpy(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.saddr))));
    sprintf(address_string + strlen(address_string), " : %i", ip_and_port.source);
    strcat(address_string, " <---> ");
    strcat(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.daddr))));
    sprintf(address_string + strlen(address_string), " : %i", ip_and_port.dest);
    strcat(address_string, "\n");
    switch (pop3_connection->nids_state)
    {
    case NIDS_JUST_EST:
        if (pop3_connection->addr.dest == 110)
        {                                                   /*POP3 客户端和 POP3 服务器端建立连接*/
            pop3_connection->client.collect++;              /*POP3 客户端接收数据*/
            pop3_connection->server.collect++;              /*POP3 服务器接收数据*/
            pop3_connection->server.collect_urg++;          /*POP3 服务器接收紧急数据*/
            pop3_connection->client.collect_urg++;          /*POP3 客户端接收紧急数据*/
            printf("%sPOP3 客户端与 POP3 服务器建立连接\n", address_string);
        }
        return;
    case NIDS_CLOSE:                                        /*POP3 客户端与 POP3 服务器端连接正常关闭*/
        printf("--------------------------------\n");
        printf("%sPOP3 客户端与 POP3 服务器连接正常关闭\n", address_string);
        return;
    case NIDS_RESET:
        /* POP3 客户端与 POP3 服务器端连接被 RST 关闭 */
        printf("--------------------------------\n");
        printf("%sPOP3 客户端与 POP3 服务器连接被 REST 关闭\n", address_string);
        return;
    case NIDS_DATA:
        {                                                   /*POP3 协议接收到新的数据*/
            char status_code[5];
            struct half_stream* hlf;
            if (pop3_connection->server.count_new_urg)
            {
                /*POP3 服务器接收到新的紧急数据*/
                printf("--------------------------------\n");
                strcpy(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.saddr))));
                sprintf(address_string + strlen(address_string), " : %i", ip_and_port.source);
```

```
        strcat(address_string, " urgent---> ");
        strcat(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.daddr))));
        sprintf(address_string + strlen(address_string), " : %i", ip_and_port.dest);
        strcat(address_string, "\n");
        address_string[strlen(address_string) + 1] = 0;
        address_string[strlen(address_string)] = pop3_connection->server.urgdata;
        printf("%s", address_string);
        return;
    }
    if (pop3_connection->client.count_new_urg)
    {
        /*POP3 服务器接收到新的紧急数据*/
        printf("--------------------------------\n");
        strcpy(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.saddr))));
        sprintf(address_string + strlen(address_string), " : %i", ip_and_port.source);
        strcat(address_string, " <--- urgent ");
        strcat(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.daddr))));
        sprintf(address_string + strlen(address_string), " : %i", ip_and_port.dest);
        strcat(address_string, "\n");
        address_string[strlen(address_string) + 1] = 0;
        address_string[strlen(address_string)] = pop3_connection->client.urgdata;
        printf("%s", address_string);
        return;
    }
    if (pop3_connection->client.count_new)
    {
        /*POP3 客户端接收到新的数据*/
        hlf = &pop3_connection->client;
        strcpy(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.saddr))));
        sprintf(address_string + strlen(address_string), ":%i", ip_and_port.source);
        strcat(address_string, " <--- ");
        strcat(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.daddr))));
        sprintf(address_string + strlen(address_string), ":%i", ip_and_port.dest);
        strcat(address_string, "\n");
        printf("--------------------------------\n");
        printf("%s", address_string);
        memcpy(content, hlf->data, hlf->count_new);
        content[hlf->count_new] = '\0';
        if (strstr(strncpy(status_code, content, 4), "+OK"))
            printf("操作成功\n");
        if (strstr(strncpy(status_code, content, 4), "-ERR"))
            printf("操作失败\n");
        for (i = 0; i < hlf->count_new; i++)
```

```
                {
                    printf("%s", char_to_ascii(content[i]));
                }
                printf("\n");
                if (strstr(content, "\n\r.\n\r"))
                    printf("数据传输结束\n");
            }
            else
            {
                /*POP3 服务器接收到新的数据*/
                hlf = &pop3_connection->server;
                strcpy(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.saddr))));
                sprintf(address_string + strlen(address_string), ":%i", ip_and_port.source);
                strcat(address_string, " ---> ");
                strcat(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.daddr))));
                sprintf(address_string + strlen(address_string), ":%i", ip_and_port.dest);
                strcat(address_string, "\n");
                printf("--------------------------------\n");
                printf("%s", address_string);
                memcpy(content, hlf->data, hlf->count_new);
                content[hlf->count_new] = '\0';
                if (strstr(content, "USER"))
                    printf("邮件用户名为\n");
                if (strstr(content, "PASS"))
                    printf("用户密码为\n");
                if (strstr(content, "STAT"))
                    printf("返回统计资料\n");
                if (strstr(content, "LIST"))
                    printf("返回邮件数量和大小\n");
                if (strstr(content, "RETR"))
                    printf("获取邮件\n");
                if (strstr(content, "DELE"))
                    printf("删除邮件\n");
                if (strstr(content, "QUIT"))
                    printf("退出连接\n");
                for (i = 0; i < hlf->count_new; i++)
                {
                    printf("%s", char_to_ascii(content[i]));
                }
                printf("\n");
            }
        }
    default:
```

```
            break;
        }
        return ;
}                                                   /*主函数*/
void main()
{
        if (!nids_init())                           /*Libnids 初始化*/
        {
            printf("出现错误: %s\n", nids_errbuf);
            exit(1);
        }
        nids_register_tcp(pop3_protocol_callback);  /*注册分析 POP3 协议的回调函数*/
        nids_run();                                 /*进入循环捕获数据包的状态*/
}
```

在 Visual C++ 6.0 中对此程序进行编译，编译过程同 7.4.1 节相关内容。

为了对此程序的运行结果进行分析，需建立测试环境。作者建立了一个邮件服务器，使用软件 CMailServer 建立邮件服务器，然后建立账号 jenny@test.com，密码为 qwertyuiopqwertyuiop，再建立账号 liuwentao@test.com，密码为 12345678901234567890。设置服务器支持 ESMTP 协议。

运行此程序，在另外一台机子上使用 Outlook Express 来接收邮箱 jenny@test.com 中的邮件。程序运行后输出内容如下：

```
192.168.0.212 : 1728 <---> 192.168.0.211 : 110
POP3 客户端与 POP3 服务器建立连接
--------------------------------
192.168.0.212:1728 <--- 192.168.0.211:110
操作成功
+OK CMailServer 5.3 POP3 Service Ready
--------------------------------
192.168.0.212:1728 ---> 192.168.0.211:110
邮件用户名为
USER jenny
--------------------------------
192.168.0.212:1728 <--- 192.168.0.211:110
操作成功
+OK welcome here
--------------------------------
192.168.0.212:1728 ---> 192.168.0.211:110
用户密码为
PASS qwertyuiopqwertyuiop
--------------------------------
192.168.0.212:1728 <--- 192.168.0.211:110
操作成功
+OK
```

```
--------------------------------
192.168.0.212:1728 ---> 192.168.0.211:110
返回统计资料
STAT
--------------------------------
192.168.0.212:1728 <--- 192.168.0.211:110
操作成功
+OK 1 1358
--------------------------------
192.168.0.212:1728 ---> 192.168.0.211:110
返回邮件数量和大小
LIST
--------------------------------
192.168.0.212:1728 <--- 192.168.0.211:110
操作成功
+OK 1 1358
--------------------------------
192.168.0.212:1728 <--- 192.168.0.211:110
1 1358
.
--------------------------------
192.168.0.212:1728 ---> 192.168.0.211:110
获取邮件
RETR 1
--------------------------------
192.168.0.212:1728 <--- 192.168.0.211:110
操作成功
+OK 1358 octets
--------------------------------
192.168.0.212:1728 <--- 192.168.0.211:110
Return-Path: <jenny@test.com>
Received: from xxs (unknown [192.168.0.212])
.by test.com with CMailServer 5.3 SMTP; Thu, 26 May 2005 20:49:20 +0800
Message-ID: <002201c561f2$04a6fe40$d400a8c0@xxs>
From: "jenny" <jenny@test.com>
To: <jenny@test.com>
Subject: test
Date: Thu, 26 May 2005 20:54:13 +0800
MIME-Version: 1.0
Content-Type: multipart/alternative;
.boundary="----=_NextPart_000_001F_01C56235.128A00F0"
X-Priority: 3
X-MSMail-Priority: Normal
```

X-Mailer: Microsoft Outlook Express 6.00.2900.2180
X-MimeOLE: Produced By Microsoft MimeOLE V6.00.2900.2180

This is a multi-part message in MIME format.

------=_NextPart_000_001F_01C56235.128A00F0
Content-Type: text/plain;
.charset="gb2312"
Content-Transfer-Encoding: base64

dGVzdCE=

------=_NextPart_000_001F_01C56235.128A00F0
Content-Type: text/html;
.charset="gb2312"
Content-Transfer-Encoding: base64

PCFET0NUWVBFIEhUTUwgUFVCTElDICItLy9XM0MvL0RURCBIVE1MIDQuMCBUcmFuc210aW9uYWwv
L0VOIj4NCjxIVE1MPjxIRUFEPg0KPE1FVEEgaHR0cCllcXVpdj1Db250ZW50LVR5cGUgY29udGVu
dD0idGV4dC9odG1sOyBjaGFyc2V0PWdlMjMxMlI+DQo8TUVUQSBjb250ZW50PSJNU0hUTUwgNi4w
MC4yOTAwLjI2MjciIG5hbWU9R0VORVJBVE9SPg0KPFNUWUxFPjwvU1RZTEU+DQo8L0hPQUQ+DQo8
Qk9EWSBiZ0NvbG9yPSNmZmZmZmY+DQo8RElWPjxGT05UIHNpemU9Mj50ZXN0ITwvRk9OVD48L0RJ
Vj48L0JPRFk+PC9IVE1MPg0K

------=_NextPart_000_001F_01C56235.128A00F0--

192.168.0.212:1728 ---> 192.168.0.211:110
删除邮件
DELE 1

192.168.0.212:1728 <--- 192.168.0.211:110
操作成功
+OK

192.168.0.212:1728 ---> 192.168.0.211:110
退出连接
QUIT

192.168.0.212:1728 <--- 192.168.0.211:110
操作成功
+OK bye

```
192.168.0.212 : 1728 <-- --> 192.168.0.211 : 110
POP3 客户端与 POP3 服务器连接正常关闭
```

程序的运行结果如图 7-17 所示。

图 7-17　分析 POP3 协议的运行结果

程序的输出内容，显示了 POP3 协议运行的全部过程。首先建立连接，然后 POP3 客户端发送用户名和密码。从分析的结果可以清楚看出，用户名为 jenny，密码为 qwertyuiopqwertyuiop，全部是以明文的方式传递。如果服务器返回成功信息，POP3 客户端就发送命令 STAT 和 LIST，获取服务器信息和邮件信息。如果操作成功，就发送 RETR 命令接收邮件。然后 POP3 服务器传送邮件到 POP3 客户端。传送完毕，POP3 客户端发送 DELE 命令删除服务器上的邮件，最后使用 QUIT 命令退出连接。

7.4.7　SMTP 和 ESMTP 协议分析

简单邮件传输协议（SMTP，Simple Mail Transfer Protocol）是一组用于由源地址到目的地址传送邮件的协议，用以控制信件的中转方式。SMTP 协议属于 TCP / IP 协议族，它帮助每台计算机在发送或中转信件时找到下一个目的地。通过 SMTP 协议所指定的服务器，可以把 E-mail 寄到收信人的服务器上。SMTP 服务器则是遵循 SMTP 协议的邮件发送服务器，用来发送或中转电子邮件。SMTP 工作在两种情况下，一是电子邮件从客户机传输到服务器，二是从某一个服务器传输到另一个服务器。SMTP 提供了一种邮件传输机制，当收件方和发件方都在一个网络上时，可以把邮件直传给对方。当双方不在同一个网络上时，需要通过一个或几个中间服务器转发。SMTP 首先由发件方提出申请，要求与接收方 SMTP 建立双向的通信渠道。收件方可以是最终收件人，也可以是中间转发的服务器。收件方服务器确认可以建立连接后，双发就可以开始通信。

SMTP 协议的基本流程如下：首先，建立连接；其次，客户端发送命令，以标识发件人

自己的身份，然后发送邮件命令，服务器端做出响应，表明是否准备接收；再次，客户端发送邮件，以标识该电子邮件的计划接收人，服务器端则表示是否愿意为收件人接收邮件，如果协商成功，则发送邮件；最后，结束此次发送，退出连接。

在 SMTP 协议中，最重要的内容是 SMTP 命令和响应状态，这是 SMTP 运行的基础。SMTP 命令都以回车换行最为结束标志，下面对它们做详细描述。

- HELLO：参数为<domain>，识别发送方到接收 SMTP 的一个 HELLO 命令。
- MAIL FROM：参数为<reverse-path>，表示发送者地址。此命令告诉接收方一个新邮件发送的开始，并对所有的状态和缓冲区进行初始化。此命令开始一个邮件传输处理，最终完成将邮件数据传送到一个或多个邮箱中。
- RCPT TO：参数为<forward-path>，表示各个邮件接收者的地址。
- DATA：无参数，接收 SMTP 将把其后的行为看做邮件数据去处理，以“回车换行.回车换行”标识数据的结尾。
- REST：无参数，表示重置会话，当前传输被取消。
- NOOP：无参数，表示无操作，服务器应响应 OK。
- QUIT：无参数，表示要求接收 SMTP 返回一个 OK 应答并关闭传输。
- VRFY：参数<string>，表示验证指定的邮箱是否存在。
- EXPN：参数<string>，表示验证给定的邮箱列表是否存在。
- HELP：无参数，查询服务器支持什么命令。

SMTP 客户端向 SMTP 服务器发送命令之后，SMTP 服务器就会返回一个应答码。应答码的每一个数字都有特定含义，第一位数字为 2 时表示命令成功，为 5 时表示失败，3 时表示没有完成。下面详细描述应答码的含义。

- 501：表示参数格式错误；
- 502：表示命令不可实现；
- 503：表示错误的命令序列；
- 504：表示命令参数不可实现；
- 211：表示系统状态或系统帮助响应；
- 214：表示帮助信息；
- 220：表示服务就绪；
- 221：表示服务关闭；
- 421：表示服务未就绪，关闭传输信道；
- 250：表示要求的邮件操作完成；
- 251：表示用户非本地；
- 450：表示要求的邮件操作未完成，邮箱不可用；
- 550：表示要求的邮件操作未完成，邮箱不可用；
- 451：表示放弃要求的操作；
- 551：表示用户非本地；
- 452：表示系统存储不足，要求的操作未执行；
- 552：表示过量的存储分配，要求的操作未执行；
- 553：表示邮箱名不可用，要求的操作未执行；
- 354：表示开始邮件输入，以"."结束；

- 554：表示操作失败。

ESMTP（Extension SMTP）即认证的邮件传输方式，是邮件服务器系统为了限制非本系统的正式用户利用本系统散发垃圾邮件或其他不当行为而开设的一项安全认证服务。它与传统的 SMTP 方式相比，主要有两点不同，一是支持 8-bit MIME 格式的编码，二是支持用户身份的验证。多了一道用户身份的验证手续，验证之后的邮件发送过程与传统的 SMTP 方式一致。为了方便用户的使用，绝大多数的 ESMTP 服务器都继承了 POP3 服务器的账号和密码设置体系，即收发邮件都用同一个账号和密码。

SMTP 的认证功能主要是增加了 AUTH 命令。AUTH 命令有多种用法，而且有多种认证机制。AUTH 支持的认证机制主要有 LOGIN，CRAM-MD5 等，其中 LGGIN 是最常用的认证机制。

发送邮件需要对用户的身份进行验证，如果账号和密码有误，ESMTP 服务器则拒绝发送该邮件，并返回 553 错误代码。

LOGIN 方式的口令-应答过程一般是这样的：首先，为客户端向服务器发送认证指令；其次，服务端返回 base64 编码串，成功码为 334，编码串说明要求客户端发送用户名；第三，客户端发送用 base64 编码的用户名，服务端返回 base64 编码串，成功码为 334，编码串说明要求客户端发送用户口令，第四，客户端发送用 base64 编码的口令，服务器进行验证，如果成功，服务端返回码为 235，表示认证成功，可以发送邮件。

本节利用 Libnids 来分析 SMTP 的操作过程，其源程序如下：

```
#include "nids.h"
char ascii_string[10000];
char* char_to_ascii(char ch)
{
    char* string;
    ascii_string[0] = 0;
    string = ascii_string;
    if (isgraph(ch))
        *string++ = ch;
    else if (ch == ' ')
        *string++ = ch;
    else if (ch == '\n' || ch == '\r')
        *string++ = ch;
    else
        *string++ = '.';
    *string = 0;
    return ascii_string;
}
/*下面是分析 SMTP 协议的回调函数*/
void smtp_protocol_callback(struct tcp_stream* smtp_connection, void** arg)
{
    int i;
    char address_string[1024];
    char content[65535];
```

```
char content_urgent[65535];
struct tuple4 ip_and_port = smtp_connection->addr;
strcpy(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.saddr))));
sprintf(address_string + strlen(address_string), " : %i", ip_and_port.source);
strcat(address_string, " <---> ");
strcat(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.daddr))));
sprintf(address_string + strlen(address_string), " : %i", ip_and_port.dest);
strcat(address_string, "\n");
switch (smtp_connection->nids_state)
{
case NIDS_JUST_EST:
    if (smtp_connection->addr.dest == 25)
    {                                                   /*SMTP 客户端和 SMTP 服务器端建立连接*/
        smtp_connection->client.collect++;              /*SMTP 客户端接收数据*/
        smtp_connection->server.collect++;              /*SMTP 服务器接收数据*/
        smtp_connection->server.collect_urg++;          /*SMTP 服务器接收紧急数据*/
        smtp_connection->client.collect_urg++;          /*SMTP 客户端接收紧急数据*/
        printf("%sSMTP 发送方与 SMTP 接收方建立连接\n", address_string);
    }
    return;
case NIDS_CLOSE:                                        /*SMTP 客户端与 SMTP 服务器连接正常关闭*/
    printf("--------------------------------\n");
    printf("%sSMTP 发送方与 SMTP 接收方连接正常关闭\n", address_string);
    return;
case NIDS_RESET:                                        /*SMTP 客户端与 SMTP 服务器连接被 RST 关闭*/
    printf("--------------------------------\n");
    printf("%sSMTP 发送方与 SMTP 接收方连接被 REST 关闭\n", address_string);
    return;
case NIDS_DATA:
    {                                                   /*SMTP 协议接收到新的数据*/
        char status_code[4];
        struct half_stream* hlf;
        if (smtp_connection->server.count_new_urg)
        {                                               /*SMTP 服务器接收到新的紧急数据*/
            printf("--------------------------------\n");
            strcpy(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.saddr))));
            sprintf(address_string + strlen(address_string), " : %i", ip_and_port.source);
            strcat(address_string, " urgent---> ");
            strcat(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.daddr))));
            sprintf(address_string + strlen(address_string), " : %i", ip_and_port.dest);
            strcat(address_string, "\n");
            address_string[strlen(address_string) + 1] = 0;
            address_string[strlen(address_string)] = smtp_connection->server.urgdata;
```

```
        printf("%s", address_string);
        return;
    }
    if (smtp_connection->client.count_new_urg)
    {                                        /*SMTP 客户端接收到新的紧急数据*/
        printf("--------------------------------\n");
        strcpy(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.saddr))));
        sprintf(address_string + strlen(address_string), " : %i", ip_and_port.source);
        strcat(address_string, " <--- urgent ");
        strcat(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.daddr))));
        sprintf(address_string + strlen(address_string), " : %i", ip_and_port.dest);
        strcat(address_string, "\n");
        address_string[strlen(address_string) + 1] = 0;
        address_string[strlen(address_string)] = smtp_connection->client.urgdata;
        printf("%s", address_string);
        return;
    }
    if (smtp_connection->client.count_new)
    {                                        /*SMTP 客户端接收到新的数据*/
        hlf = &smtp_connection->client;
        strcpy(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.saddr))));
        sprintf(address_string + strlen(address_string), ":%i", ip_and_port.source);
        strcat(address_string, " <--- ");
        strcat(address_string, inet_ntoa(*((struct in_addr *) &(ip_and_port.daddr))));
        sprintf(address_string + strlen(address_string), ":%i", ip_and_port.dest);
        strcat(address_string, "\n");
        printf("--------------------------------\n");
        printf("%s", address_string);
        memcpy(content, hlf->data, hlf->count_new);
        content[hlf->count_new] = '\0';
        if (strstr(strncpy(status_code, content, 3), "221"))
            printf("连接终止\n");
        if (strstr(strncpy(status_code, content, 3), "250"))
            printf("操作成功\n");
        if (strstr(strncpy(status_code, content, 3), "220"))
            printf("表示服务就绪\n");
        if (strstr(strncpy(status_code, content, 3), "354"))
            printf("开始邮件输入，以\".\"结束\n");
        if (strstr(strncpy(status_code, content, 3), "334"))
            printf("服务器响应验证\n");
        if (strstr(strncpy(status_code, content, 3), "235"))
            printf("认证成功可以发送邮件了\n");
        for (i = 0; i < hlf->count_new; i++)
```

```
    }
    return ;
}                                                    /*主函数*/
void main()
{
    if (!nids_init())                                /*Libnids 初始化*/
    {
        printf("%s\n", nids_errbuf);
        exit(1);
    }
    nids_register_tcp(smtp_protocol_callback);       /*注册分析 TCP 协议的回调函数*/
    nids_run();                                      /*进入循环捕获数据包状态*/
}
```

在 Visual C++ 6.0 中编译此程序，编译过程同 7.4.1 节相关内容。

在 7.4.6 节建立了邮件服务器，在此同样使用此邮件服务器，在另外一台机子上使用 Outlook Express 来发送邮件。程序的输出内容如下：

```
192.168.0.212 : 1712 <---> 192.168.0.211 : 25
SMTP 发送方与 SMTP 接收方建立连接
--------------------------------
192.168.0.212:1712 <--- 192.168.0.211:25
表示服务就绪
220 ESMTP CMailServer 5.3.2005.05.18 SMTP Service Ready(Unregistered)
--------------------------------
192.168.0.212:1712 ---> 192.168.0.211:25
HELLO 命令
EHLO xxs
--------------------------------
192.168.0.212:1712 <--- 192.168.0.211:25
操作成功
250-AUTH=LOGIN
250 AUTH LOGIN
--------------------------------
192.168.0.212:1712 ---> 192.168.0.211:25
请求认证
认证机制为 LOGIN
AUTH LOGIN
--------------------------------
192.168.0.212:1712 <--- 192.168.0.211:25
服务器响应验证
334 VXNlcm5hbWU6
--------------------------------
192.168.0.212:1712 ---> 192.168.0.211:25
amVubnk=
--------------------------------
```

```
192.168.0.212:1712 <--- 192.168.0.211:25
服务器响应验证
334 UGFzc3dvcmQ6
--------------------------------
192.168.0.212:1712 ---> 192.168.0.211:25
cXdlcnR5dWlvcHF3ZXJ0eXVpb3A=
--------------------------------
192.168.0.212:1712 <--- 192.168.0.211:25
认证成功可以发送邮件了
235 OK
--------------------------------
192.168.0.212:1712 ---> 192.168.0.211:25
发送方邮件地址为
MAIL FROM: <jenny@test.com>
--------------------------------
192.168.0.212:1712 <--- 192.168.0.211:25
操作成功
250 OK
--------------------------------
192.168.0.212:1712 ---> 192.168.0.211:25
接收方邮件地址为
RCPT TO: <liuwentao@test.com>
--------------------------------
192.168.0.212:1712 <--- 192.168.0.211:25
操作成功
250 OK
--------------------------------
192.168.0.212:1712 ---> 192.168.0.211:25
RSET
--------------------------------
192.168.0.212:1712 <--- 192.168.0.211:25
操作成功
250 OK
--------------------------------
192.168.0.212:1712 ---> 192.168.0.211:25
发送方邮件地址为
MAIL FROM: <jenny@test.com>
--------------------------------
192.168.0.212:1712 <--- 192.168.0.211:25
操作成功
250 OK
--------------------------------
192.168.0.212:1712 ---> 192.168.0.211:25
接收方邮件地址为
RCPT TO: <liuwentao@test.com>
```

QkSEWBB1ZOHvbCSyPSNmZmZkZmY+DQoBBBlWVjxUTOSUjHNpemU9M150ZXH0IPbvvik9OVD4RL0RJ
Vj48LOJPRPk+PC8IVEIMPg0K

-------=_NextPart_000_0016_01C56239.AA876AB0

192.168.0.213:1712 ---> 192.168.0.211:25

数据传输结束

192.168.0.212:1712 <--- 192.168.0.211:25

操作成功

250

192.168.0.212:1712 ---> 192.168.0.211:25

退出连接

QUIT

192.168.0.212:1712 <--- 192.168.0.211:25

连接终止

221 SMTP SERVICE CLOSED

192.168.0.212 : 1712 <---> 192.168.0.211 : 25

SMTP 发送方与 SMTP 接收方连接被 RESET 关闭

程序的运行结果如图 7-18 所示。

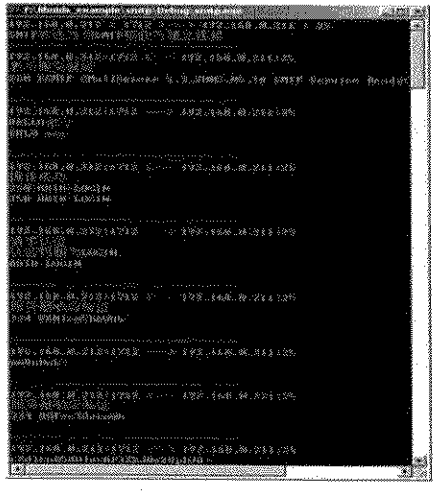

图 7-18　分析 SMTP 协议的运行结果

从程序的输出结果分析，在发送邮件时，使用了 ESMTP 协议，需要进行用户和密码认证。其传输的内容是 base64 编码的字符串。如果认证成功，就使用命令 MAIL FROM 表示发送方地址，使用命令 RCPT TO 表示接收方地址。然后使用命令 DATA 来传输邮件内容。数据传递完毕后，使用 QUIT 命令退出 SMTP 服务。

7.4.8 IP 数据包的捕获和分析

使用 Libnids 不仅可以捕获正常的 IP 数据包，也可以捕获异常的 IP 数据包以及 IP 碎片。Libnids 提供了捕获 IP 碎片的接口，可对 IP 碎片进行重组。下面举一个例子，利用 Libnids 来捕获所有的 IP 数据包（包括 IP 碎片），然后对它们进行分析。

```
#include "nids.h"                                   /*UDP 协议首部的数据结构*/
struct udp_header
{
    unsigned short udp source port;
    unsigned short udp_destination_port;
    unsigned short udp_length;
    unsigned short udp_checksum;
};                                                  /*ICMP 协议首部的数据结构*/
struct icmp_header
{
    unsigned int icmp_type;
    unsigned int icmp_code;
    unsigned char icmp_checksum;
    unsigned char icmp_id;
    unsigned char icmp_sequence;
};                                                  /*IP 协议首部的数据结构*/
struct ip_header
{
#if defined(WORDS_BIGENDIAN)
    unsigned char ip_version : 4,                   /*版本*/
          ip_header_length : 4;                     /*首部长度*/
#else
    unsigned char ip_header_length : 4,
          ip_version : 4;
#endif
    unsigned char ip_tos;                           /*服务类型*/
    unsigned short ip_length;                       /*总长度*/
    unsigned short ip_id;                           /*标识*/
    unsigned short ip_off;                          /*标志和偏移*/
    unsigned char ip_ttl;                           /*生存时间*/
    unsigned char ip_protocol;                      /*协议类型*/
    unsigned short ip_checksum;                     /*校验和*/
    struct in_addr ip_souce_address;                /*源 IP 地址*/
```

```
    printf("----------  ICMP 协议   ----------\n");
    printf("ICMP 类型:%d\n", icmp_protocol->icmp_type);
    switch (icmp_protocol->icmp_type)                                   /*ICMP 类型*/
    {
    case 8:
        printf("ICMP 回显请求协议\n");
        printf("ICMP 代码:%d\n", icmp_protocol->icmp_code);
        printf("标识符:%d\n", icmp_protocol->icmp_id);
        printf("序列码:%d\n", icmp_protocol->icmp_sequence);
        break;
    case 0:
        printf("ICMP 回显应答协议\n");
        printf("ICMP 代码:%d\n", icmp_protocol->icmp_code);
        printf("标识符:%d\n", icmp_protocol->icmp_id);
        printf("序列码:%d\n", icmp_protocol->icmp_sequence);
        break;
    default :
        break;
    }
    printf("ICMP 校验和:%d\n", ntohs(icmp_protocol->icmp_checksum));   /*获取校验和*/
    return;
}
/*下面是分析 TCP 协议的函数*/
void tcp_protocol_packet_callback(const u_char* packet_content)
{
    struct tcp_header* tcp_protocol;
    u_char flags;
    int header_length;
    u_short source_port;
    u_short destination_port;
    u_short windows;
    u_short urgent_pointer;
    u_int sequence;
    u_int acknowledgement;
    unsigned char  checksum;
    tcp_protocol = (struct tcp_header *) (packet_content + 14 + 20);
    source_port = ntohs(tcp_protocol->tcp_source_port);                 /*获取源端口号*/
    destination_port = ntohs(tcp_protocol->tcp_destination_port);       /*获取目的端口号*/
    header_length = tcp_protocol->tcp_offset * 4;                       /*获取首部长度*/
    sequence = ntohl(tcp_protocol->tcp_sequence);                       /*获取序列码*/
    acknowledgement = ntohl(tcp_protocol->tcp_acknowledgement);         /*获取确认号*/
    windows = ntohs(tcp_protocol->tcp_windows);                         /*获取窗口大小*/
    urgent_pointer = ntohs(tcp_protocol->tcp_urgent_pointer);           /*获取紧急指针*/
```

```
    flags = tcp_protocol->tcp_flags;
    checksum = ntohs(tcp_protocol->tcp_checksum);
    printf("-------  TCP 协议   -------\n");
    printf("源端口号:%d\n", source_port);
    printf("目的端口号:%d\n", destination_port);
    switch (destination_port)
    {
    case 80:
        printf("上层协议为 HTTP 协议\n");   break;
    case 21:
        printf("上层协议为 FTP 协议\n");break;
    case 23:
        printf("上层协议为 Telnet 协议\n");break;
    case 25:
        printf("上层协议为 SMTP 协议\n");break;
    case 110:
        printf("上层协议 POP3 协议\n");break;
    default:
        break;
    }
    printf("序列码:%u\n", sequence);
    printf("确认号:%u\n", acknowledgement);
    printf("首部长度:%d\n", header_length);
    printf("保留:%d\n", tcp_protocol->tcp_reserved);
    printf("标记:");
    if (flags & 0x08)
        printf("PSH ");
    if (flags & 0x10)
        printf("ACK ");
    if (flags & 0x02)
        printf("SYN ");
    if (flags & 0x20)
        printf("URG ");
    if (flags & 0x01)
        printf("FIN ");
    if (flags & 0x04)
        printf("RST ");
    printf("\n");
    printf("窗口大小:%d\n", windows);
    printf("校验和:%d\n", checksum);
    printf("紧急指针:%d\n", urgent_pointer);
}
/*下面是分析 UPD 协议的函数*/
```

```
void udp_protocol_packet_callback(u_char* packet_content)
{
    struct udp_header* udp_protocol;
    u_short source_port;
    u_short destination_port;
    u_short length;
    udp_protocol = (struct udp_header *) (packet_content + 20);
    source_port = ntohs(udp_protocol->udp_source_port);                /*获取源端口号*/
    destination_port = ntohs(udp_protocol->udp_destination_port);      /*获取目的端口号*/
    length = ntohs(udp_protocol->udp_length);
    printf("----------  UDP 协议首部   ----------\n");
    printf("源端口:%d\n", source_port);
    printf("目的端口:%d\n", destination_port);
    switch (destination_port)
    {
    case 138:
        printf("NETBIOS Datagram Service\n");  break;
    case 137:
        printf("NETBIOS Name Service\n");break;
    case 139:
        printf("NETBIOS session service\n");break;
    case 53:
        printf("name-domain server \n");break;
    default:
        break;
    }
    printf("长度:%d\n", length);
    printf("校验和:%d\n", ntohs(udp_protocol->udp_checksum));
}
/*下面是分析 IP 协议的函数*/
void ip_protocol_packet_callback(u_char* packet_content)
{
    struct ip_header* ip_protocol;
    u_int header_length;
    u_int offset;
    u_char tos;
    unsigned short checksum;
    printf("----------  IP 协议首部  ----------\n");
    ip_protocol = (struct ip_header *) (packet_content);
    checksum = ntohs(ip_protocol->ip_checksum);                    /*获取校验和*/
    header_length = ip_protocol->ip_header_length * 4;             /*获取首部长度*/
    tos = ip_protocol->ip_tos;
    offset = ntohs(ip_protocol->ip_off);
```

```
    printf("IP版本:%d\n", ip_protocol->ip_version);
    printf("首部长度:%d\n", header_length);
    printf("TOS:%d\n", tos);
    printf("总长度:%d\n", ntohs(ip_protocol->ip_length));
    printf("标识:%d\n", ntohs(ip_protocol->ip_id));
    printf("偏移:%d\n", (offset & 0x1fff) * 8);
    printf("生存时间:%d\n", ip_protocol->ip_ttl);
    printf("协议:%d\n", ip_protocol->ip_protocol);
    switch (ip_protocol->ip_protocol)                          /*判断上层协议类型*/
    {
    case 6:
        printf("上层协议为TCP\n");break;
    case 17:
        printf("上层协议为UDP\n");break;
    case 1:
        printf("上层协议为ICMP\n");break;
    default:
        break;
    }
    printf("校验和:%d\n", checksum);
    printf("源IP地址:%s\n", inet_ntoa(ip_protocol->ip_souce_address));
    printf("目的IP地址:%s\n", inet_ntoa(ip_protocol->ip_destination_address));
    switch (ip_protocol->ip_protocol)
    {
    case 17:
        udp_protocol_packet_callback(packet_content);break;
        /*上层协议为UDP协议，调用分析UDP协议的函数 */
    case 6:
        tcp_protocol_packet_callback(packet_content);break;
        /*上层协议为TCP协议，调用分析TCP协议的函数 */
    case 1:
        icmp_protocol_packet_callback(packet_content);break;
        /*上层协议为ICMP协议，调用分析ICMP协议的函数 */
    default:
        break;
    }
}
/*下面是回调函数*/
void ip_callback (struct ip* a_packet, int len)
{
    ip_protocol_packet_callback(a_packet);              /*调用分析IP协议的函数*/
}
/*主函数*/
```

```
void main()
{
    if (!nids_init())                                   /*Libnids 初始化*/
    {
        printf("出现错误: %s\n", nids_errbuf);
        exit(1);
    }
    nids_register_ip_frag(ip_callback);                 /*注册分析 IP 协议的回调函数*/
    nids_run();                                         /*进入循环捕获数据包的状态*/
}
```

在 Visual C++ 6.0 中编译此程序，编译过程同 7.4.1 节相关内容。

运行此程序，捕获 IP 数据包，然后对其进行分析。程序输出的内容如下：

```
---------- IP 协议首部 ----------
IP 版本:4
首部长度:20
TOS:0
总长度:48
标识:8429
偏移:0
生存时间:128
协议:6
上层协议为 TCP
校验和:22243
源 IP 地址:192.168.0.212
目的 IP 地址:192.168.0.211
------- TCP 协议 -------
源端口号:65280
目的端口号:65280
序列码:3171090432
确认号:0
首部长度:0
保留:5
标记:RST
窗口大小:18176
校验和:0
紧急指针:38144
```

程序的运行结果如图 7-19 所示。

此程序的功能是捕获所有的 IP 数据包，然后对捕获到的 IP 数据包进行分析。本程序分析了基于 IP 协议的 TCP 协议、UDP 协议和 ICMP 协议。

图 7-19　分析 IP 协议的运行结果

7.4.9　检测攻击的一个例子

使用 Libnids 可以检测网络攻击，它是入侵检测系统的开发包。在 Libnids 中提供了一些基本的检测技术，如对网络扫描攻击的检测，以及对异常 IP 数据包、异常 TCP 数据包和异常 UDP 数据包的检测。

在 Libnids 提供的所有功能的基础上还可以进行更深入的开发，以检测更多的入侵行为，它为进一步的开发提供了最基本的功能，其中最重要的就是 TCP 数据流重组功能。在此基础上，可以检测更多的网络攻击行为。另外 Libnids 已经为开发者提供了一些检测攻击的功能，如上所述的端口扫描检测功能，数据包异常检测功能。

本节利用 Libnids 来实现检测端口扫描攻击，整个程序的源代码如下，其主要功能来自 Libnids 中提供的默认函数 sys_log（）：

```
#include "nids.h"
/*下面是检测扫描用的扫描信息数据结构*/
struct scan
{
    u_int addr;                     /*地址*/
    unsigned short port;            /*端口号*/
    u_char flags;                   /*标记*/
};
/*下面是检测扫描时用到的扫描主机数据结构*/
struct host
{
    struct host* next;              /*下一个主机结点*/
    struct host* prev;              /*前一个主机结点*/
    u_int addr;                     /*地址*/
    int modtime;                    /*时间*/
    int n_packets;                  /*个数*/
    struct scan* packets;           /*扫描信息*/
};
```

```
/*下面是IP协议首部的数据结构*/
struct ip_header
{
#if defined(WORDS_BIGENDIAN)
    unsigned int ip_v : 4,
          ip_hl : 4;
#else
    unsigned int ip_hl : 4,
         ip_v : 4;
#endif
    unsigned int ip_tos;
    unsigned char ip_len;
    unsigned char ip_id;
    unsigned char ip_off;
    unsigned int ip_ttl;
    unsigned int ip_p;
    unsigned char ip_csum;
    struct in_addr ip_src;
    struct in_addr ip_dst;
} ;
/* 下面是TCP协议首部的数据结构 */
struct tcp_header0
{
    unsigned char th_sport;                    /*源端口号*/
    unsigned char th_dport;                    /*目的端口号*/
    unsigned short th_seq;                     /*序列号*/
    unsigned short th_ack;                     /*确认号*/
#ifdef WORDS_BIGENDIAN
    unsigned int th_off : 4,                   /*数据偏移*/
             th_x2 : 4;                        /*保留*/
#else
    unsigned int th_x2 : 4,                    /*保留*/
             th_off : 4;                       /*数据偏移*/
#endif
    unsigned int th_flags;
    unsigned char th_win;                      /*窗口大小*/
    unsigned char th_sum;                      /*校验和*/
    unsigned char th_urp;                      /*紧急指针*/
} ;
/*下面是检测扫描攻击和异常数据包的函数*/
static void my_nids_syslog(int type, int errnum, struct  ip_header* iph, void* data)
{
    static int scan_number = 0;
```

```
        printf("%s", string_content);
        sprintf(string_content, "%s\n", inet_ntoa(*((struct in_addr *) &(host_information->addr))));
        printf("%s", string_content);
        sprintf(string_content, "被扫描者的 IP 地址和端口号为:\n");
        printf("%s", string_content);
        sprintf(string_content, "");
        for (i = 0; i < host_information->n_packets; i++)
        {
            strcat(string_content,
                inet_ntoa(*((struct in_addr *) &(host_information->packets[i].addr))));
            sprintf(string_content + strlen(string_content), ":%hi\n", host_information->packets
            [i].port);
            flagsand &= host_information->packets[i].flags;
            flagsor |= host_information->packets[i].flags;
        }
        printf("%s", string_content);
        sprintf(string_content, "");
        if (flagsand == flagsor)
        {
            i = flagsand;
            switch (flagsand)
            {
            case 2:
                strcat(string_content, "扫描类型为: SYN\n");
                break;
            case 0:
                strcat(string_content, "扫描类型为: NULL\n");
                break;
            case 1:
                strcat(string_content, "扫描类型为: FIN\n");
                break;
            default:
                sprintf(string_content + strlen(string_content), "标志=0x%x\n", i);
            }
        }
        else
        {
            strcat(string_content, "标志异常\n");
        }
        printf("%s", string_content);
        break;
    default:
        sprintf(content, "未知");
```

```
            printf("%s", string_content);
            break;
        }
}
/* 主函数 */
void main()
{
    nids_params.syslog = my_nids_syslog;                    /*注册检测攻击的函数*/
    nids_params.pcap_filter = "ip";
    if (!nids_init())                                       /*Libnids 初始化*/
    {
        printf("出现错误: %s\n", nids_errbuf);
        exit(1);
    }
    nids_run();                                             /*进入循环捕获数据包的状态*/
}
```

在 Visual C++ 6.0 中编译此程序，编译过程同 7.4.1 节相关内容。

此程序的主要功能是检测端口扫描攻击和异常数据包。为了分析程序运行结果，使用著名的端口扫描工具 Nmap 来进行测试。运行此例程后，使用 Nmap 工具来扫描另外一台机子，如图 7-20 所示。

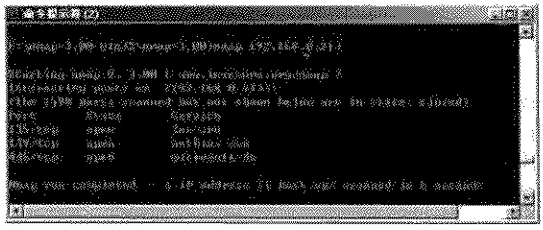

图 7-20　使用 Nmap 工具扫描

程序的输出内容如下，这里只节选了其一部分：

```
-------------- 1 --------------
---- 发现扫描攻击 -----
扫描者的 IP 地址为:
192.168.0.241
被扫描者的 IP 地址和端口号为:
192.168.0.213:166
192.168.0.213:1416
192.168.0.213:604
192.168.0.213:123
192.168.0.213:241
192.168.0.213:826
```

192.168.0.213:5803

192.168.0.213:412

192.168.0.213:994

192.168.0.213:5550

192.168.0.213:6050

扫描类型为：SYN

---------- 2 ----------

----- 发现扫描攻击 -----

扫描者的 IP 地址为：

192.168.0.211

被扫描者的 IP 地址和端口号为：

192.168.0.213:1377

192.168.0.213:746

192.168.0.213:1472

192.168.0.213:1027

192.168.0.213:48

192.168.0.213:830

192.168.0.213:3986

192.168.0.213:1670

192.168.0.213:1538

192.168.0.213:986

192.168.0.213:822

扫描类型为：SYN

---------- 3 ----------

----- 发现扫描攻击 -----

扫描者的 IP 地址为：

192.168.0.211

被扫描者的 IP 地址和端口号为：

192.168.0.213:399

192.168.0.213:948

192.168.0.213:202

192.168.0.213:3372

192.168.0.213:295

192.168.0.213:463

192.168.0.213:2638

192.168.0.213:484

192.168.0.213:195

192.168.0.213:2046

192.168.0.213:878

扫描类型为：SYN

---------- 4 ----------

----- 发现扫描攻击 -----

扫描者的 IP 地址为：

```
192.168.0.211
被扫描者的 IP 地址和端口号为：
192.168.0.213:972
192.168.0.213:111
192.168.0.213:627
192.168.0.213:4662
192.168.0.213:5999
192.168.0.213:309
192.168.0.213:1507
192.168.0.213:83
192.168.0.213:561
192.168.0.213:824
192.168.0.213:1389
扫描类型为：SYN
```

程序的运行结果如图 7-21 所示。

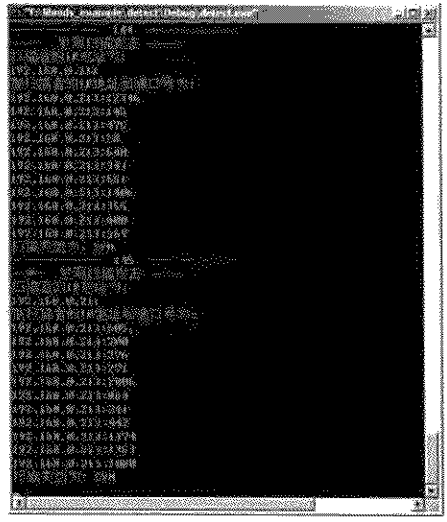

图 7-21　检测扫描攻击的运行结果

从程序的输出内容可以看出，程序检测到了扫描攻击行为，并对攻击方的 IP 地址进行显示，对被扫描的 IP 地址和端口进行了分析，且分析了扫描的类型。在此测试例子中，攻击方 IP 地址为 192.168.0.211，被攻击方 IP 地址为 192.168.0.213，扫描类型为 TCP SYN 扫描，其扫描的端口都一一列出。

7.5　小结

本章详细阐述了 Libnids 开发包，对它的数据结构、输出函数以及使用方法进行了详细

介绍。Libnids 是一个入侵检测系统开发包，可以构造入侵检测系统的基本框架，为开发者进一步开发入侵检测系统提供了最基本的功能。

利用 Libnids 提供的 TCP 数据流重组功能，可以非常方便地对基于 TCP 协议的应用层协议进行分析，对它们的操作过程以及传输的数据内容进行分析，可以获取敏感信息，如账号和密码等。也可以在此基础上分析应用层协议传输的内容，重现和还原网络数据，如邮件、网页等。

Libnids 不仅提供了分析 TCP 协议的接口，也提供了分析 UDP 协议和 IP 协议的接口。利用 Libnids 可以捕获所有的 IP 数据包，包括 IP 碎片数据包，可以方便地对基于 IP 协议、UDP 协议以及 TCP 协议的各种上层协议进行分析。Libnids 还提供了简单的入侵攻击检测功能，如端口扫描攻击检测、异常数据包检测等。

本章最后通过举例对 Libnids 的使用方法进行详细阐述，具体分析了 HTTP 协议、FTP 协议、Telnet 协议、FTP 协议、POP3 协议以及 SMTP/ESMTP 协议，通过 Libnids 对这些协议的运行机理进行了深入阐述。通过这些例子可以深入理解 Libnids 的使用方法，并且对这些应用层协议的运行过程以及传输的数据内容有更深入的了解，这对于理解这些协议的运行机理是很有意义的。

Libnids 的强大功能使它应用广泛，特别适合于入侵检测系统的设计，还可用于网络监视系统、网络数据包分析系统等。

参考文献

1 W.Richard Stevens, Bill Fenner, Andrew M.Rudoff. UNIX 网络编程 卷 1：套接字联网 API（英文版第三版）. 北京：机械工业出版社，2004

2 W. Richard Stevens. TCP/IP 详解 卷 1：协议. 范建华等译. 北京：机械工业出版社，2000

3 W. Richard Stevens. UNIX 环境高级编程. 尤晋元等译. 北京：机械工业出版社，2000

4 （美）Brian Caswel 等. Snort 2.0 入侵检测. 宋劲松译. 北京：国防工业出版社，2004

5 （美）Douglas E. Comer. 用 TCP/IP 进行网际互联第一卷：原理、协议与结构（第四版）. 林瑶等译. 北京：电子工业出版社，2001

6 （美）Douglas E. Comer David L. Stevens. 用 TCP/IP 进行网际互联第二卷：设计、实现与内核（第三版）. 张娟等译. 北京：电子工业出版社，2001

7 http://libdnet.sourceforge.net/

8 http://libnids.sourceforge.net/

9 http://www.packetfactory.net/Projects/Libnet/

10 http://www.tcpdump.org/

11 http://www.winpcap.org/

12 （美）Douglas E. Comer, David L. Stevens. 用 TCP/IP 进行网际互联第三卷：客户-服务器编程与应用（Linux/POSIX 套接字版）. 赵刚等译. 北京：电子工业出版社，2001

13 （美）Douglas E. Comer,David L. Stevens. TCP/IP 网络互联技术（卷 3）：客户-服务器编程与应（Windows 套接字版）. 张卫等译. 北京：清华大学出版社，2004

14 Rebecca Gurley Bace. 入侵检测. 陈明奇等译. 北京：人民邮电出版社，2001

15 冯登国. 网络安全原理与技术. 北京：科学出版社，2003

16 卿斯汉. 密码学与计算机网络安全. 北京：清华大学出版社，2001

17 Andrew S. Tanenbaum. 计算机网络（第 4 版）（中文版）. 潘爱民译. 北京：清华大学出版社，2004

18 冯登国. 计算机通信网络安全. 北京：清华大学出版社，2001

19 戴英侠等. 系统安全与入侵检测. 北京：清华大学出版社，2002

20 http://www.openssl.org

21 http://analyzer.polito.it

22 http://www.ethereal.com/

23 http://www.datanerds.net/~mike/dsniff.html

24 http://netresearch.ics.uci.edu/kfujii/jpcap/doc/index.html

25 http://www.bribes.org/perl/wnetpcap.html

26 http://ngrep.datasurge.net/

27 http://packetvb.sourceforge.net/

28 http://www.snort.org/

29 http://www.winpcap.org/windump

30 Anthony Jones. Windows 网络编程技术. 京京工作室译. 北京：机械工业出版社，2000

31 毛德操，胡希明．Linux 内核源代码情景分析（下册）．杭州：浙江大学出版社，2001
32 （美）Tom Swan．深入学习：GNU C++ for Linux 编程技．邱仲潘等译．北京：电子工业出版社，2000
33 （美）John Shapley Gray．UNIX 进程间通信（第二版）．张宁等译．北京：电子工业出版社，2001
34 朱雁辉．Windows 防火墙与网络封包截获技术．北京：电子工业出版社，2002
35 （美）Greg Holden．防火墙与网络安全——入侵检测和 VPNs．王斌，孔璐译．北京：清华大学出版社，2004
36 （美）William Stallings．密码编码学与网络安全：原理与实践（第三版）．刘玉珍等译．北京：电子工业出版社，2004
37 鲁士文．计算机网络协议和实现技术．北京：清华大学出版社，2000
38 （美）Kenneth D. Reed．协议分析（第 7 版）．孙坦等译．北京：电子工业出版社，2004