第八章　端口扫描器的设计与实现

8.1 本章训练目的与要求

端口扫描器是一种重要的网络安全检测工具。通过端口扫描，不仅可以发现目标主机的开放端口和操作系统的类型，还可以查找系统的安全漏洞，获得弱口令等相关信息。因此，端口扫描技术是网络安全的基本技术之一，对于维护系统的安全性有着十分重要的意义。

本章编程训练的目的如下：

① 掌握端口扫描器的基本设计方法。

② 理解 ping 程序，TCP connect 扫描，TCP SYN 扫描，TCP FIN 扫描以及 UDP 扫描的工作原理。

③ 熟练掌握 Linux 环境下的套接字编程技术。

④ 掌握 Linux 环境下多线程编程的基本方法

本章编程训练的要求如下：

① 编写端口扫描程序，提供 TCP connect 扫描，TCP SYN 扫描，TCP FIN 扫描以及 UDP 扫描 4 种基本扫描方式。

② 设计并实现 ping 程序，探测目标主机是否可达。

8.2 相关背景知识

8.2.1　ping 程序

ping 程序是日常网络管理中经常使用的程序。它用于确定本地主机与网络中其它主机的通信情况。因为只是简单地探测某一 IP 地址所对应的主机是否存在，因此它的原理十分简单。扫描发起主机向目标主机发送一个要求回显（type = 8，即为 ICMP_ECHO）的 ICMP 数据包，目标主机在收到请求后，会返回一个回显（type = 0，即为 ICMP_ECHOREPLY）的 ICMP 数据包。扫描发起主机可以通过是否接收到响应的 ICMP 数据包来判断目标主机是否存在。

在本章编程中，可以在向目标主机发起端口扫描之前使用 ping 程序确定目标主机是否存在。如果 ping 目标主机成功，则继续后面的扫描工作；否则，放弃对目标主机的扫描。

8.2.2 TCP 扫描

1. TCP 连接建立过程

TCP 协议的包头结构如图 2-3 所示。对于 TCP 端口扫描而言，TCP 协议的标志位起着至关重要的作用。各标志位的含义参考本书第 2 章关于 TCP 协议的介绍，此处不再赘述。

虽然 TCP 扫描的种类繁多，但是它们的原理都与 TCP 连接的建立过程密切相关。在介绍各种 TCP 扫描之前，必须深入理解一个 TCP 连接的建立过程。该过程的步骤如下：

(1) 首先客户端（请求方）在连接请求中，向服务器端（接收方）发送 SYN=1，ACK=0 的 TCP 数据包，表示希望与服务器建立一个连接。

(2) 如果服务器端响应这个连接请求，就返回一个 SYN=1，ACK=1 的数据包给客户端，表示服务器端同意建立这个连接，并要求客户端进行确认。

(3) 最后客户端发送一个 SYN=0，ACK=1 的数据包给服务器端，表示确认建立连接。

2. TCP 协议相关的三种扫描

(1) connect 扫描

TCP connect 扫描非常简单。扫描发起主机只需要调用系统 API connect 尝试连接目标主机的指定端口，如果 connect 成功，意味着扫描发起主机与目标主机之间至少经历了一次完整的 TCP 三次握手建立连接过程，被测端口开放；否则，端口关闭。

虽然在编程时不需要程序员手动构造 TCP 数据包，但是 connect 扫描的效率非常低下。由于 TCP 协议是可靠协议，connect 系统调用不会在尝试发送第一个 SYN 包未得到响应的情况下就放弃，而是会经过多次尝试后才彻底放弃，因此需要较长的时间。此外，connect 失败会在系统中造成大量连接失败日志，容易被管理员发现。

(2) SYN 扫描

TCP SYN 扫描是使用最广泛的扫描方式，其原理就是向待扫描端口发送 SYN 数据包。如果扫描发起主机能够收到 ACK|SYN 数据包，则表示端口开放；如果收到 RST 数据包，则表示端口关闭。如果未收到任何数据包，且确定目标主机存在，那么发送给被测端口的 SYN 数据包可能被防火墙等安全设备过滤。因为 SYN 扫描不需要完成 TCP 连接的三次握手过程，所以它又被称为半开放扫描。

SYN 扫描的最大优点就是速度快。在 Internet 中，如果不考虑防火墙的影响，SYN 扫描每秒钟可以扫描数千个端口。但是由于其扫描行为较为明显，SYN 扫描容易被入侵检测系统发现，也容易被防火墙屏蔽。同时构造原始的 TCP 数据包也需要较高的系统权限（在 Linux 中仅限于 root 账户）。

(3) FIN 扫描

TCP FIN 扫描会向目标主机的被测端口发送一个 FIN 数据包。如果目标主机没有任何响应且确定该主机存在，那么表示目标主机正在监听这个端口，端口是开放的；如果目标主机

返回一个 RST 数据包且确定该主机存在，那么表示目标主机没有监听这个端口，端口是关闭的。

FIN 扫描具有良好的隐蔽性，不会留下日志。但是它的应用具有很大的局限性，由于不同系统实现网络协议栈的细节不同，FIN 扫描只能扫描 Linux/UNIX 系统。对于 Windows 系统而言，由于无论端口开放与否，都会返回 RST 数据包，因此对端口的状态无法进行判断。

8.2.3 UDP 扫描

一般情况下，当向一个关闭的 UDP 端口发送数据时，目标主机会返回一个 ICMP 不可达（ICMP port unreachable）的错误。UDP 扫描就是利用了上述原理，向被扫描端口发送 0 字节的 UDP 数据包，如果收到一个 ICMP 不可达响应，那么就认为端口是关闭的；而对于那些长时间没有响应的端口，则认为是开放的。

但是，因为大部分系统都限制了 ICMP 差错报文的产生速度，所以针对特定主机的大范围 UDP 端口扫描的速度非常缓慢。此外，UDP 协议和 ICMP 协议是不可靠协议，没有收到响应的情况也可能是由于数据包丢失造成的，因此扫描程序必须对同一端口进行多次尝试后才能得出正确的结论。

8.2.4 使用原始套接字构造并发送数据包

本书第 5 章已经对原始套接字（SOCK_RAW）进行了简单的介绍。与流套接字和数据报套接字想比，原始套接字的独特之处在于程序员可以创建并填充协议头。在编写端口扫描器程序时，除了 TCP connect 扫描可以直接调用 connect 函数完成数据包的封装，发送和接收工作以外，其它扫描程序（包括 ping 程序）都需要手动构造数据包。因此，使用原始套接字构造数据包并将其发送到目标主机的指定端口是编写端口扫描器程序的关键。下面详细介绍一下使用原始套接字构造并发送数据包的基本流程。

1. 创建原始套接字

与前几章介绍的一样，一般采用 socket 函数创建原始套接字。函数声明如下，其中通讯域（参数 domain）一般选择 AF_INET，表示 IPv4 协议，套接字类型（参数 type）选择 SOCK_RAW，表示建立原始套接字，通信协议（参数 protocol）表明操作的是哪一种协议的数据包，一般有 IPPROTO_IP 、IPPROTO_TCP、IPPROTO_UDP、IPPROTO_ICMP 等。

```
Int socket(int domain, int type, int protocol);
```

2. 设置套接字选项

创建原始套接字后，可以根据需要调用 setsockopt 函数来设置当前套接字的选项。setsocketopt 函数的声明如下：

```
int setsockopt (int sockfd, int level, int optname, const char void *optval, socklen_t* optlen );
```

参数 sockfd 是标识套接字的描述符。参数 level 表示控制套接字的层次。对 TCP/IP 协

议族而言，level 支持 3 种层次 SOL_SOCKET（通用套接字选项），IPPROTO_IP（IP 选项）和 IPPROTO_TCP（TCP 选项）。参数 Optname 表示 sockfd 所标识的套接字需要设置选项的名称。如果需要构建数据包的 IP 头，那么就将 Optname 设为 IP_HDRINCL。参数 optval 是一个指针，指向存放选项值的缓冲区。实践中，optval 需要根据选项不同的数据类型进行转换。参数 optlen 表示 optval 所指向的缓冲区的长度。

3. 创建并填充协议报头

表 8-1　Linux 系统常用协议的报头结构

协议头	数据结构	头文件
IP 协议头	struct iphdr	<netinet/ip.h>
TCP 协议头	struct tcphdr	<netinet/tcp.h>
UDP 协议头	struct udphdr	<netinet/udp.h>
ICMP 协议头	struct icmphdr	<netinet/ip_icmp.h>

如表 8-1 所示，Linux 系统已经定义常用协议的报头结构，比如 iphdr，tcphdr，udphdr，icmphdr 等。在创建协议数据包的时候，只需要根据对应的结构体定义申请一块新的内存空间，并用指针指向这块空间的地址即可。结构 iphdr，tcphdr，udphdr，icmphdr 的声明如下所示。

```
//结构体声明                                    解释
struct iphdr                                    //IP 协议头
{
    #elif defined
    (_LITTLE_ENDIAN_BITFIELD)                   //IP 协议的版本定义
    _u8 version :4,
    #elif defined                               //我国一般使用 BIG 的定义
    (_BIG_ENDIAN_BITFIELD)
    _u8 version:4,                              //IP 报头标长
    ihl:4;
    #else
    #error "Please fix"
    #endif                                      //服务类型
    _u8 tos;                                    //总长度
    _16 tot_len;                                //标识
    _u16 id;                                    //标志+片偏移
    _u16 frag_off;                              //生存时间
    _u8 ttl;                                    //协议
    _u8 protocol;                               //头校验和
    _u16 check;                                 //源 IP 地址
    _u32 saddr;                                 //目的 IP 地址
    _u32 daddr;
};
struct tcphdr                                   //TCP 协议头
{
```

```
        u_int16_t source;                          //源端口号
        u_int16_t dest;                            //目的端口号
        u_int32_t seq;                             //序号
        u_int32_t ack_seq;                         //确认号
        # if _BYTE_ORDER == _LITTLE _ENDIAN        //LITTLE 版本的定义
          …
        #elif _BYTE _ORDER == _BIG _ENDIAN
        u_int16_t doff:4;                          //4 bit 报头长度
        u_int16_t res1:4;                          //6 bit 保留
        u_int16_t res2:2;
        u_int16_t urg:1;                           //UGR 位
        u_int16_t ack:1;                           //ACK 位
        u_int16_t psh:1;                           //PSH 位
        u_int16_t rst:1;                           //RST 位
        u_int16_t syn:1;                           //SYN 位
        u_int16_t fin:1;                           //FIN 位
        #else
        #error "Adjust your defines"
        #endif
        u_int16_t window;                          //窗口大小
        u_int16_t check;                           //校验和
        u_int16_t urg_ptr;                         //紧急指针
};
struct udphdr                                      //UDP 协议头
{
        u_int16_t source;                          //源端口
        u_int16_t dest;                            //目的端口
        u_int16_t len;                             //数据报长度
        u_int16_t check;                           //校验和
};
struct icmphdr                                     //ICMP 协议头
{
        u_int8_t type;                             //类型
        u_int8_t code;                             //代码
        u_int16_t checksum;                        //0 校验和
        union
        {
          struct                                   //echo 数据报
          {
            u_int16_t id;
            u_int16_t sequence;
          } echo;
          u_int32_t gateway;                       //网关地址
          struct                                   //MTU 值
```

```
        {
          u_int16_t_unused;
          u_int16_t mtu;
        } frag;
      } un;
    };
```

4. 发送数据包

在填充完数据包之后，需要调用 sendto 函数将数据包发送出去。函数声明如下，参数 s 是标识套接字的描述符，参数 buf 为指向发送数据包的指针，参数 len 表示内存空间的大小，参数 to 为指向接收方地址的指针，参数 tolen 为指向接收方地址大小的指针。sendto 函数既可用于无连接套接字通信，又可用于面向连接的套接字通信。当用于面向连接的通信时（套接字类型为 SOCK_STREAM），将省略参数 to 和 tolen。

```
ssize_t sendto(int s, const void* buf, size_t len, int flags, const struct sockaddr* to,
                socklen_t tolen)
```

需要指出的是，在一般情况下 sendto 函数的 buf 指针所指向的数据包不包含 IP 报头。比如，在 UDP 扫描中，buf 所指向缓冲区内只包含 UDP 头和数据字段。如果要亲自处理 IP 报头，一定要在第二步中调用 setsocketopt 函数设置套接字的选项，代码如下：

```
int flag = 1;
setsockopt(sockfd, IPPROTO_IP, IP_HDRINCL, &flag, sizeof(int));
```

调用 setsockopt 函数后，buf 所指向的缓冲区由 IP 头，TCP 头（或者 UDP 头等其它协议头）和数据字段三部分组成。

5. 接收数据包

在发送完数据包以后，需要调用 recvfrom 函数来接收响应数据包。recvfrom 函数的声明如下，参数 s 是标识套接字的描述符，参数 buf 为指向接收到信息的指针，参数 len 为内存空间的大小，flags 为控制参数，用于控制是否接收带外数据，参数 from 为指向发送方地址的指针，参数 fromlen 为指向发送方地址大小的指针。同样，recvfrom 函数既可以用于无连接的套接字通信，也可以用于面向连接的通信。

```
ssize_t recvfrom(int s, void* buf, size_t len, int flags, struct sockaddr* from,
                  socklen_t* fromlen)
```

一般在接收数据时，recvfrom 函数处于阻塞状态，直到收到一个数据包才会返回。如果希望 recvfrom 在接收数据包时处于非阻塞状态，需要调用函数 fcntl 将套接字设置为非阻塞模式。具体代码如下，参数 Sockfd 表示所设置的套接字描述符。

```
fcntl(int Sockfd, F_SETFL, O_NONBLOCK)
```

另外，在 Linux 中，可以将套接字看成文件描述符，这样在进行数据发送与接收的时候就可以调用 write 和 read 函数对文件描述符进行读写了。write 函数和 read 函数定义如下：

```
ssize_t write(int fd, const void *buf, size_t nbytes)
ssize_t read(int fd, void *buf, size_t nbyte)
```

write 函数将 buf 中的 nbytes 字节内容写入文件描述符 fd，若成功，则返回写的字节数；若失败，则返回-1，并设置 errno 变量。read 函数是负责从 fd 中读取内容。当读取成功时，read 返回实际所读取的字节数；如果返回的值是 0 ，表示已经读到文件末尾；小于 0 表示出现错误。如果错误为 EINTR 说明读是由中断引起的，如果是 ECONNREST 表示网络连接出现问题。

8.3 实例编程练习

8.3.1 编程练习要求

在 Linux 环境下编写一个端口扫描器，利用套接字（socket）正确实现 ping 程序、TCP connect 扫描、TCP SYN 扫描、TCP FIN 扫描、以及 UDP 扫描。ping 程序在用户输入被扫描主机 IP 地址之后探测该主机是否可达。其它四种扫描在指定被扫描主机 IP，起始端口以及终止端口之后，从起始端口到终止端口对被测主机进行扫描。最后将每一个端口的扫描结果正确地显示出来。具体要求如下所示：

1. 程序输入格式

程序为命令行程序，可执行文件名为 Scaner，命令行格式如下：

./Scaner [选项]

其中[选项]是程序为用户提供的多种功能。本程序中，[选项]包括{-h，-c，-s，-f，-u}五项基本功能。-h 表示显示帮助信息；-c 表示进行 TCP connect 扫描；-s 表示进行 TCP SYN 扫描；-f 表示进行 TCP FIN 扫描；-u 表示进行 UDP 扫描。

2. 程序的执行过程

(1) 停用 iptables 服务

首先在 Shell 命令行下输入“service iptables stop”停止 iptables 防火墙的过滤功能，保证端口扫描程序能够正常的接收各种响应数据包。

(2) 打印帮助信息

在控制台命令行中输入 ./Scaner –h，打印端口扫描器程序的帮助信息（如下所示）。帮助信息详细地说明了程序的各个选项和参数。用户可以通过查询帮助信息充分了解程序的各项功能。

```
[root@localhost Scaner]# ./Scaner –h
 Scaner: usage:    [-h]    --help information
                   [-c]    --TCP connect scan
                   [-s]    --TCP syn scan
                   [-f]    --TCP fin scan
                   [-u]    --UDP scan
```

(3) 进行 TCP connect 扫描

在控制台命令行中输入 ./Scaner –c，开始 TCP connect 扫描（如下所示）。程序提示用户输入扫描目标主机的 IP 地址，扫描起始端口与终止端口。在检验 IP 地址和端口的正确性之后，程序首先利用 ping 程序探测目标主机的 IP 地址是否可达，如果不可达，则放弃对该主机的扫描；否则开启扫描线程，依次扫描每个端口并将扫描结果实时地显示出来。

```
[root@localhost Scaner]# ./Scaner –c
Please input IP address of a Host:192.168.1.158
Please input the range of port...
Begin Port:1
End Port:255
Scan Host 192.168.1.158 port 1~10 ...
Ping Host 192.168.1.158 Successfully !
Begin TCP connect scan...
Host: 192.168.1.158 Port: 1 closed !
Host: 192.168.1.158 Port: 2 closed !
…
```

(4) 进行 TCP SYN 扫描

在控制台命令行中输入 ./Scaner –s，开始 TCP SYN 扫描（如下所示）。程序提示用户输入扫描目标主机的 IP 地址，扫描起始端口与终止端口。在检验 IP 地址和端口的正确性之后，程序首先利用 ping 程序探测目标主机的 IP 地址是否可达，如果不可达，则放弃对该主机的扫描；否则开启扫描线程，依次扫描每个端口并将扫描结果实时地显示出来。

```
[root@localhost Scaner]# ./Scaner -s
Please input IP address of a Host:192.168.1.158
Please input the range of port...
Begin Port:1
End Port:255
Scan Host 192.168.1.158 port 1~10 ...
Ping Host 192.168.1.158 Successfully !
Begin TCP SYN scan...
Host: 192.168.1.158 Port: 1 closed !
Host: 192.168.1.158 Port: 2 closed !
…
```

(5) 进行 TCP FIN 扫描

在控制台命令行中输入 ./Scaner –f，开始 TCP FIN 扫描（如下所示）。程序提示用户输

入扫描目标主机的 IP 地址，扫描起始端口与终止端口。在检验 IP 地址和端口的正确性之后，程序首先利用 ping 程序探测目标主机的 IP 地址是否可达，如果不可达，则放弃对该主机的扫描；否则开启扫描线程，依次扫描每个端口并将扫描结果实时地显示出来。

```
[root@localhost Scaner]# ./Scaner -f
Please input IP address of a Host:192.168.1.158
Please input the range of port...
Begin Port:1
End Port:255
Scan Host 192.168.1.158 port 1~10 ...
Ping Host 192.168.1.158 Successfully !
Begin TCP FIN scan...
Host: 192.168.1.158 Port: 1 closed !
Host: 192.168.1.158 Port: 2 closed !
…
```

(6) 进行 TCP UDP 扫描

在控制台命令行中输入 ./Scaner –u，开始 UDP 扫描（如下所示）。程序提示用户输入扫描目标主机的 IP 地址，扫描起始端口与终止端口。在检验 IP 地址和端口的正确性之后，程序首先利用 ping 程序探测目标主机的 IP 地址是否可达，如果不可达，则放弃对该主机的扫描；否则开启扫描线程，依次扫描每个端口并将扫描结果实时地显示出来。

```
[root@localhost Scaner]# ./Scaner -u
Please input IP address of a Host:192.168.1.158
Please input the range of port...
Begin Port:1
End Port:10
Scan Host 192.168.1.158 port 1~10 ...
Ping Host 192.168.1.158 Successfully !
Begin UDP scan...
Host: 192.168.1.158 Port: 1 closed !
Host: 192.168.1.158 Port: 2 closed !
…
```

综上所述，本程序在 Linux 平台下，利用套接字编程实现了 ping 程序，并且在指定扫描端口范围的情况下，实现了对目标主机的 TCP connect 扫描，TCP SYN 扫描，TCP FIN 扫描，以及 UDP 扫描。

8.3.2 编程训练设计与分析

1. 端口扫描器程序 Scaner

端口扫描器程序 Scaner 的流程比较简单。在 main 函数中只需完成与用户进行交互，检测用户输入，以及调用各个扫描功能模块 3 方面的工作。如图 8-1 所示，Scaner 程序的流程

分为以下 8 个步骤。

(1) 首先程序判断是否需要输出帮助信息，若是，则输出端口扫描器程序的帮助信息，然后退出；否则，继续执行下面的步骤。

(2) 用户输入被扫描主机的 IP 地址，扫描起始端口和终止端口。

(3) 判断 IP 地址与端口号是否错误，若错误，则提示用户并退出；否则，继续下面的步骤。

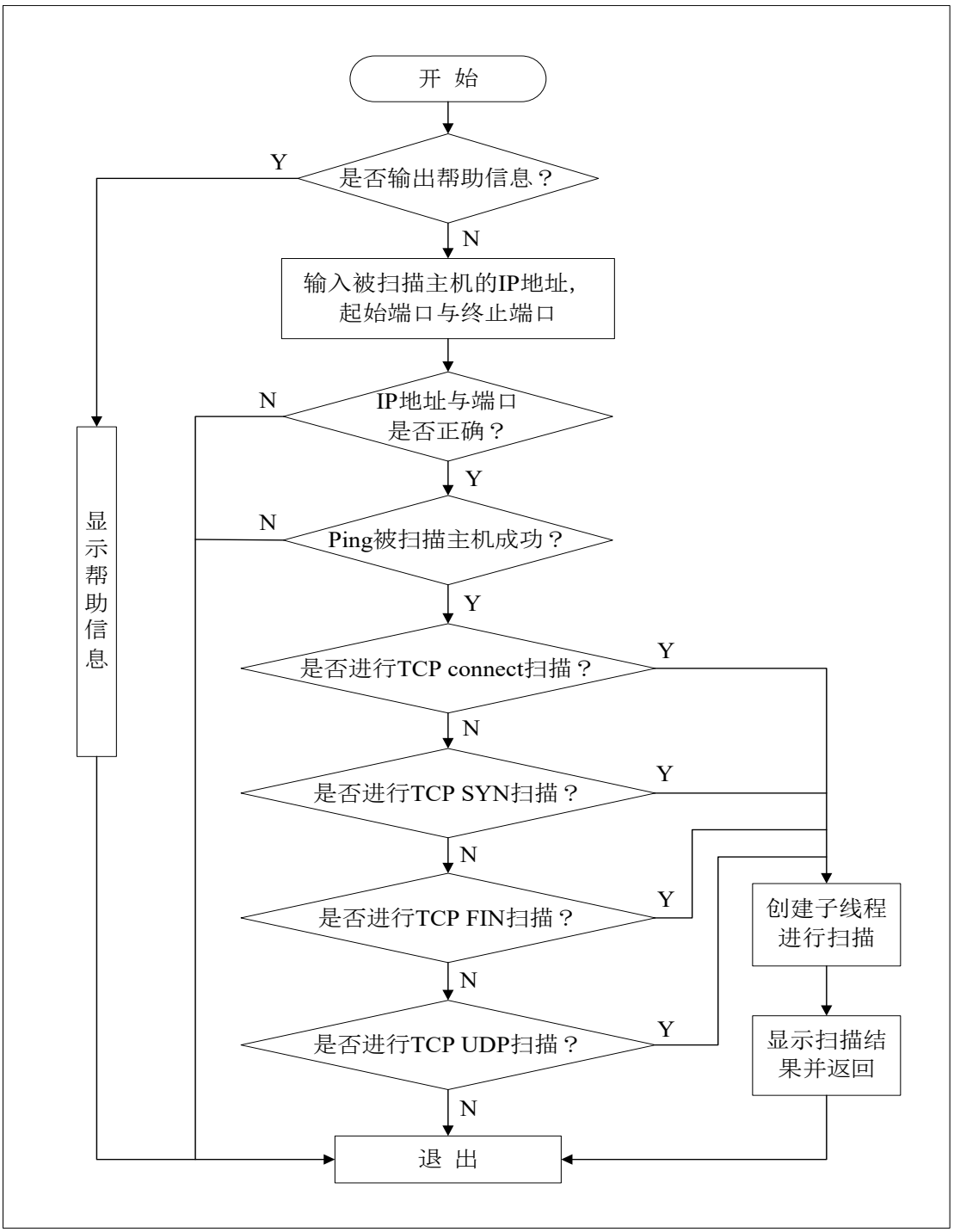

图 8-1 Scaner 程序流程图

(4) 调用 Ping 函数，判断被扫描主机是否可达，若不可达，则提示用户并退出；否则，继续下面的步骤。

(5) 判断是否进行 TCP connect 扫描，若是，则开启 TCP connect 扫描子线程，从起始端口到终止端口对目标主机进行扫描，并把扫描结果显示出来；否则，继续执行下面的步骤。

(6) 判断是否进行 TCP SYN 扫描，若是，则开启 TCP SYN 扫描子线程，从起始端口到终止端口对目标主机进行扫描，并把扫描结果显示出来。否则，继续执行下面的步骤。

(7) 判断是否进行 TCP FIN 扫描，若是，则开启 TCP FIN 扫描子线程，从起始端口到终止端口对目标主机进行扫描，并把扫描结果显示出来。否则，继续执行下面的步骤。

(8) 判断是否进行 UDP 扫描，若是，则开启 UDP 扫描子线程，从起始端口到终止端口对目标主机进行扫描，并把扫描结果显示出来。否则，继续执行下面的步骤。

(9) 等待所有扫描子线程返回后退出。

在端口扫描器的主流程中先后调用了 ping 程序，TCP connect 扫描，TCP SYN 扫描，TCP FIN 扫描，以及 UDP 扫描 5 个功能模块。下面将详细介绍这些模块的设计与实现方法。

2. ICMP 探测指定主机

Ping 程序用于测量本地主机与目标主机之间的网络通信情况。Ping 程序首先发送一个 ICMP 请求数据包给目标主机。如果目标主机返回一个 ICMP 响应数据包，那么表示两台主机之间的通信状况良好，可以继续后面的扫描操作；否则，整个程序将退出。

在端口扫描器程序中，Ping 功能是由函数 bool Ping(string HostIP,unsigned LocalHostIP) 实现的。在 Ping 函数中完成了填充 ICMP 数据包，向目标主机发送请求，以及接收响应等工作。若目标主机可达，则返回 ture，否则，返回 false。Ping 函数的部分代码如下。

```
bool Ping(string HostIP,unsigned LocalHostIP)
{
    /*变量定义*/
    //创建套接字
    PingSock = socket(AF_INET,SOCK_RAW,IPPROTO_ICMP);
    …
    //设置套接字选项
    on = 1;
    ret = setsockopt(PingSock,0,IP_HDRINCL,&on,sizeof(on));
    …
    //创建 ICMP 请求数据包
    SendBufSize = sizeof(struct iphdr)+ sizeof(struct icmphdr) + sizeof(struct timeval);
    SendBuf = (char*)malloc(SendBufSize);
    memset(SendBuf,0,sizeof(SendBuf));

    //填充 IP 头
    ip = (struct iphdr*)SendBuf;
    ip->ihl = 5;
```

```
ip->version = 4;
ip->tos = 0;
ip->tot_len = htons(SendBufSize);
ip->id = rand();
ip->ttl = 64;
ip->frag_off = 0x40;
ip->protocol = IPPROTO_ICMP;
ip->check = 0;
ip->saddr = LocalHostIP;
ip->daddr = inet_addr(&HostIP[0]);

//填充 icmp 头
icmp = (struct icmphdr*)(ip+1);
icmp->type = ICMP_ECHO;
icmp->code = 0;
icmp->un.echo.id = htons(LocalPort);
icmp->un.echo.sequence = 0;

tp = (struct timeval*)&SendBuf[28];
gettimeofday(tp,NULL);
icmp->checksum = in_cksum((u_short *)icmp,
                          sizeof(struct icmphdr)+sizeof(struct timeval));

//设置套接字的发送地址
PingHostAddr.sin_family = AF_INET;
PingHostAddr.sin_addr.s_addr = inet_addr(&HostIP[0]);
Addrlen = sizeof(struct sockaddr_in);

//发送 ICMP 请求
ret = sendto(PingSock,SendBuf,SendBufSize,0,(struct sockaddr*)&PingHostAddr,
        sizeof(PingHostAddr));
 …
//将套接字设置为非阻塞模式
if(fcntl(PingSock, F_SETFL, O_NONBLOCK) == -1)
 …
//循环等待接收 ICMP 响应
gettimeofday(&TpStart,NULL);                    //获得循环起始时刻
flags =false;
do
{
    //接收 ICMP 响应
    ret = recvfrom(PingSock,RecvBuf,1024,0,(struct sockaddr*)&FromAddr,
                   (socklen_t*)&Addrlen);
    if (ret > 0)   //如果接收到一个数据包，对其进行解析
```

```
            {
                Recvip = (struct ip*)RecvBuf;
                Recvicmp = (struct icmp*)(RecvBuf+(Recvip->ip_hl*4));

                SrcIP = inet_ntoa(Recvip->ip_src);     //获得响应数据包 IP 头的源地址
                DstIP = inet_ntoa(Recvip->ip_dst);     //获得响应数据包 IP 头的目的地址

                in_LocalhostIP.s_addr = LocalHostIP;
                LocalIP = inet_ntoa(in_LocalhostIP);   //获得本机 IP 地址
                //判断该数据包的源地址是否等于被测主机的 IP 地址，目的地址是否等于
                //本机 IP 地址，ICMP 头的 type 字段是否为 ICMP_ECHOREPLY
                if (SrcIP == HostIP && DstIP == LocalIP &&
                    Recvicmp->icmp_type == ICMP_ECHOREPLY)
                {   /*ping 成功，退出循环*/                    }
            }
            //获得当前时刻，判断等待相应时间是否超过 3 秒，若是，则退出等待。
            gettimeofday(&TpEnd,NULL);
            TimeUse=(1000000*(TpEnd.tv_sec-TpStart.tv_sec)+
                    (TpEnd.tv_usec-TpStart.tv_usec))/1000000.0;
            if(TimeUse<3)
              …
        } while(true);
         …
    }
```

如图 8-2 所示，Ping 函数的流程分为以下 8 个步骤。

(1) 为 Ping 程序创建原始套接字（SOCK_RAW）PingSock。

(2) 调用函数 setsockopt，设置套接字选项，使其能够重新构造 IP 数据报头部。

(3) 创建 ICMP 请求数据包。该数据包由三部分组成：IP 头，ICMP 头，以及数据字段。在本程序中以当前的时间作为数据字段的内容。

(4) 设置套接字 PingSock 的目标主机地址。

(5) 调用 sendto 函数，发送 ICMP 请求数据包。

(6) 调用函数 fcntl 将套接字设置为非阻塞模式。

(7) 调用函数 recvfrom，循环等待接收 ICMP 响应数据包。如果接收到一个数据包，判断 a）源地址是否等于目标主机的 IP 地址，b）目的地址是否等于本机 IP 地址，c）ICMP 头的 type 字段是否为 ICMP_ECHOREPLY。若上述 3 个条件均满足，则表示成功接收到目标主机发来的 ICMP 响应数据包，Ping 函数返回 true，否则，继续等待。

(8) 如果循环等待时间超过 3 秒，则退出等待，Ping 函数返回 false。

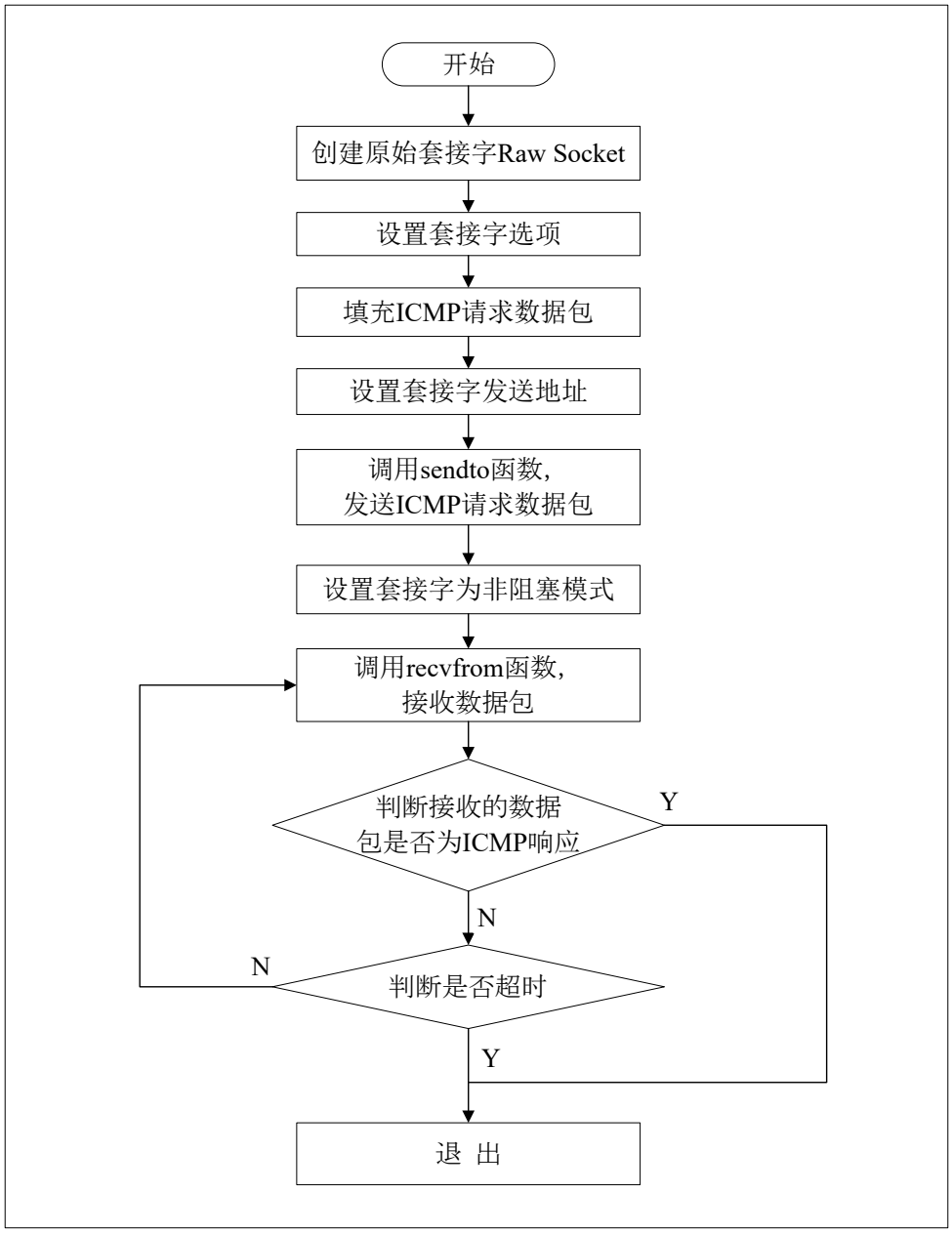

图 8-2　Ping 函数流程图

3. TCP connect 扫描

TCP Connect 扫描时通过调用流套接字（SOCK_STREAM）的 connect 函数实现的。该函数尝试连接被测主机的指定端口，若连接成功，则表示端口开启；否则，表示端口关闭。在编程中为了提高效率，采用了创建子线程同时扫描目标主机多个端口的方法。线程函数的代码如下。

```
void* Thread_TCPconnectHost(void* param)
{
    /*变量定义*/
    //获得目标主机的 IP 地址和扫描端口号
```

```
    p=(struct TCPConHostThrParam*)param;
    HostIP = p->HostIP;
    HostPort = p->HostPort;

    //创建流套接字
    ConSock = socket(AF_INET,SOCK_STREAM,0);
    …
    //设置连接主机地址
    memset(&HostAddr,0,sizeof(HostAddr));
    HostAddr.sin_family = AF_INET;
    HostAddr.sin_addr.s_addr = inet_addr(&HostIP[0]);
    HostAddr.sin_port = htons(HostPort);

    //connect 目标主机
    ret = connect(ConSock,(struct sockaddr*)&HostAddr,sizeof(HostAddr));
    if(ret==-1)
    {/* 连接失败，端口关闭*/}
    else
    {/* 连接成功，端口开启*/}

    //退出线程
    …
    close(ConSock);      //关闭套接字
    //子线程数减 1
    pthread_mutex_lock(&TCPConScanlocker);
        TCPConThrdNum--;
    pthread_mutex_unlock(&TCPConScanlocker);
}
//=====================================================================
void* Thread_TCPconnectScan(void* param)
{
    /*变量定义*/
    //获得扫描的目标主机 IP，启始端口，终止端口
    p=(struct TCPConThrParam*)param;
    HostIP = p->HostIP;
    BeginPort = p->BeginPort;
    EndPort = p->EndPort;

    TCPConThrdNum = 0;    //将线程数设为 0
    //开始从起始端口到终止端口循环扫描目标主机的端口
    for (TempPort=BeginPort;TempPort<=EndPort;TempPort++)
    {
        //设置子线程参数
        TCPConHostThrParam *pConHostParam = new TCPConHostThrParam;
```

```
            pConHostParam->HostIP = HostIP;
            pConHostParam->HostPort = TempPort;

            //将子线程设为分离状态
            pthread_attr_init(&attr);
            pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);

            //创建 connect 目标主机指定的端口子线程
            ret=pthread_create(&subThreadID,&attr,Thread_TCPconnectHost,
                               pConHostParam);
            …
            //线程数加 1
            pthread_mutex_lock(&TCPConScanlocker);
            TCPConThrdNum++;
            pthread_mutex_unlock(&TCPConScanlocker);
            //如果子线程数大于 100，暂时休眠
            while (TCPConThrdNum>100)
            { sleep(3); }
        }

        //等待子线程数为 0，返回
        while (TCPConThrdNum != 0)
        { sleep(1); }
        //返回主流程
        pthread_exit(NULL);
    }
```

上述代码中包含着两个线程函数：Thread_TCPconnectScan 和 Thread_TCPconnectHost。这两个函数共同完成了 TCP connect 扫描的工作。其中 Thread_TCPconnectScan 是扫描的主线程函数，该函数在扫描器的主流程中被调用，用于遍历目标主机的端口，创建负责扫描某一固定端口的子线程。而连接（connect）目标主机指定端口的工作正是由线程函数 Thread_TCPconnectHost 来完成的。端口扫描器主流程，函数 Thread_TCPconnectScan 和函数 Thread_TCPconnectHost 三者之间的关系如图 8-3 所示。为了维护系统中的线程数目，使用变量 TCPConThrdNum 来记录已经创建的子线程数。在函数 Thread_TCPconnectScan 中，每创建一个连接指定端口的子线程，就将 TCPConThrdNum 加 1。而在函数 Thread_TCPconnectHost 退出当前线程之前，将 TCPConThrdNum 减 1。若子线程数大于 100，线程 Thread_TCPconnectScan 就会暂时休眠，等待子线程数降低后再继续工作。当子线程等于 0 时，表示所有的子线程都已经返回，线程 Thread_TCPconnectScan 就返回程序主流程。为了保证多个不同线程对子线程数 TCPConThrdNum 的互斥访问，在修改 TCPConThrdNum 之前需要加互斥锁 TCPConScanlocker，修改完毕后再解锁。这样就保证了 TCPConThrdNum 的正确性。

线程函数 Thread_TCPconnectHost 是整个 TCP connect 扫描的核心部分。线程函数 Thread_TCPconnectScan 通过调用它来创建连接目标主机指定端口的子线程。函数 Thread_TCPconnectHost 的工作流程如下：

(1) 获得线程函数 Thread_TCPconnectScan 传来的目标主机 IP 地址和扫描端口号。

(2) 创建流套接字 ConSock。

(3) 设置连接目标主机的套接字地址 HostAddr。

(4) 调用 connect 函数连接目标主机。若函数返回-1，则连接失败；否则，连接成功。

(5) 关闭套接字 ConSock，子线程数 TCPConThrdNum 减 1，退出线程。

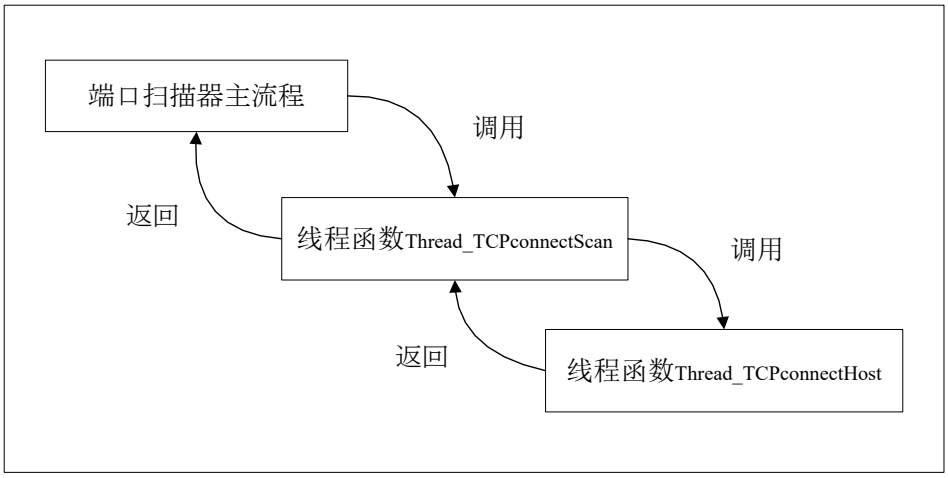

图 8-3 主流程，Thread_TCPconnectScan 和 Thread_TCPconnectHost 之间的关系

4. TCP SYN 扫描

在本章中，TCP SYN 扫描是通过原始套接字（SOCK_RAW）实现的。原始套接字允许程序员构造数据包的 IP 头字段和 TCP 头字段。虽然 Windows XP 系统出于安全的原因禁止原始套接字发送数据的功能，但是在 Linux 网络编程中，它还是给我们带来了很多方便。

和 TCP connect 扫描一样，TCP SYN 扫描也包含了两个线程函数：Thread_TCPSynScan 和 Thread_TCPSYNHost。Thread_TCPSynScan 是主线程函数，负责遍历目标主机的被测端口，并调用 Thread_TCPSYNHost 函数创建多个扫描子线程。Thread_TCPSYNHost 函数用于完成对目标主机指定端口的 TCP SYN 扫描。两个函数的代码如下：

```
Void* Thread_TCPSYNHost(void* param)
{
    /*变量定义*/
    //获得目标主机的 IP 地址和扫描端口号，以及本机的 IP 地址和端口
    p=(struct TCPSYNHostThrParam*)param;
    …
    //设置 TCP SYN 扫描的套接字地址
    memset(&SYNScanHostAddr,0,sizeof(SYNScanHostAddr));
    SYNScanHostAddr.sin_family = AF_INET;
    SYNScanHostAddr.sin_addr.s_addr = inet_addr(&HostIP[0]);
    SYNScanHostAddr.sin_port = htons(HostPort);
```

```
//创建套接字
SynSock=socket(PF_INET, SOCK_RAW, IPPROTO_TCP);
…
//填充 TCP SYN 数据包
struct pseudohdr *ptcph=(struct pseudohdr*)sendbuf;
struct tcphdr *tcph=(struct tcphdr*)(sendbuf+sizeof(struct pseudohdr));

//填充 TCP 伪头部，用于计算校验和
ptcph->saddr = LocalHostIP;
ptcph->daddr = inet_addr(&HostIP[0]);
ptcph->useless = 0;
ptcph->protocol = IPPROTO_TCP;
ptcph->length = htons(sizeof(struct tcphdr));

//填充 TCP 头
tcph->th_sport=htons(LocalPort);
tcph->th_dport=htons(HostPort);
tcph->th_seq=htonl(123456);
tcph->th_ack=0;
tcph->th_x2=0;
tcph->th_off=5;
tcph->th_flags=TH_SYN;                    //TCP 头 flags 字段的 SYN 位置 1
tcph->th_win=htons(65535);
tcph->th_sum=0;
tcph->th_urp=0;
tcph->th_sum=in_cksum((unsigned short*)ptcph, 20+12);

//发送 TCP SYN 数据包
len=sendto(SynSock, tcph, 20, 0, (struct sockaddr *)&SYNScanHostAddr,
                sizeof(SYNScanHostAddr));
…
//接收目标主机的 TCP 响应数据包
len=read(SynSock, recvbuf, 8192);
if(len <= 0)
{   /*接收错误*/ }
else
{      …
    //判断响应数据包的源地址是否等于目标主机地址，目的地址是否等于本机
    //IP 地址，源端口是否等于被扫描端口，目的端口是否等于本机端口号
    if(HostIP == SrcIP && LocalIP == DstIP && SrcPort == HostPort &&
       DstPort == LocalPort)
    {
        if (tcph->th_flags == 0x14)   //判断是否为 SYN|ACK 数据包
        {   /*端口开启*/ }
```

```
                if (tcph->th_flags == 0x12)   //判断是否为 RST 数据包
                {   /*端口关闭*/ }
            }
        }

        //退出子线程
        delete p;
        close(SynSock);

        pthread_mutex_lock(&TCPSynScanlocker);
        TCPSynThrdNum--;
        pthread_mutex_unlock(&TCPSynScanlocker);
    }
    //=================================================================
    void* Thread_TCPSynScan(void* param)
    {
        /*变量定义*/
        //获得目标主机的 IP 地址和扫描的起始端口号，终止端口号，以及本机的 IP 地址
        p=(struct TCPSYNThrParam*)param;
        …
        //循环遍历扫描端口
        TCPSynThrdNum = 0;
        LocalPort = 1024;

        for (TempPort=BeginPort;TempPort<=EndPort;TempPort++)
        {
            //设置子线程参数
            struct TCPSYNHostThrParam *pTCPSYNHostParam =
                                                    new TCPSYNHostThrParam;
            pTCPSYNHostParam->HostIP = HostIP;
            pTCPSYNHostParam->HostPort = TempPort;
            pTCPSYNHostParam->LocalPort = TempPort + LocalPort;
            pTCPSYNHostParam->LocalHostIP = LocalHostIP;

            //将子线程设置为分离状态
            pthread_attr_init(&attr);
            pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);

            //创建子线程
            ret=pthread_create(&subThreadID,&attr,Thread_TCPSYNHost,
                                pTCPSYNHostParam);
            …
            //子线程数加 1
            pthread_mutex_lock(&TCPSynScanlocker);
```

```
            TCPSynThrdNum++;
            pthread_mutex_unlock(&TCPSynScanlocker);
            //子线程数大于 100，休眠
            while (TCPSynThrdNum>100)
            {   sleep(3);   }
        }
        //等待所有子线程返回
        while (TCPSynThrdNum != 0)
        {   sleep(1);   }
        //返回主流程
        pthread_exit(NULL);
    }
```

线程函数 Thread_TCPSYNHost 是整个 TCP SYN 扫描的核心部分。线程函数 Thread_TCPSynScan 通过调用它来创建子线程，向目标主机的指定端口发送 SYN 数据包，并根据目标主机的响应判断端口的状态。函数 Thread_TCPSYNHost 的工作流程如图 8-4 所示：

(1) 获得线程函数 Thread_TCPSynScan 传来的参数，其中包括目标主机 IP 地址和扫描端口号，以及本机 IP 地址和端口号。

(2) 设置 TCP SYN 扫描的套接字地址。

(3) 创建原始套接字 SynSock。

(4) 填充 TCP SYN 数据包。注意 TCP 头 flags 字段的 SYN 位设置为 1。

(5) 调用 sendto 函数向目标主机的指定端口发送 TCP SYN 数据包。

(6) 调用 read 函数接收目标主机的 TCP 响应数据包。若函数的返回值小于 0，则接收错误。否则，继续判断响应数据包的地址是否错误。如果响应数据包的 a）源地址等于目标主机地址，b）目的地址等于本机 IP 地址，c）源端口号等于被扫描端口号，d）目的端口是否等于本机端口号，那么该数据包就是目标主机被扫描端口返回的响应数据包。若该数据包 flags 字段的 ACK 和 SYN 位均置 1，那么表示被扫描端口开启。若 flags 字段的 RST 位置 1，则表示被扫描端口关闭。

(7) 关闭套接字 ConSock，子线程数 TCPSynThrdNum 减 1，退出线程。

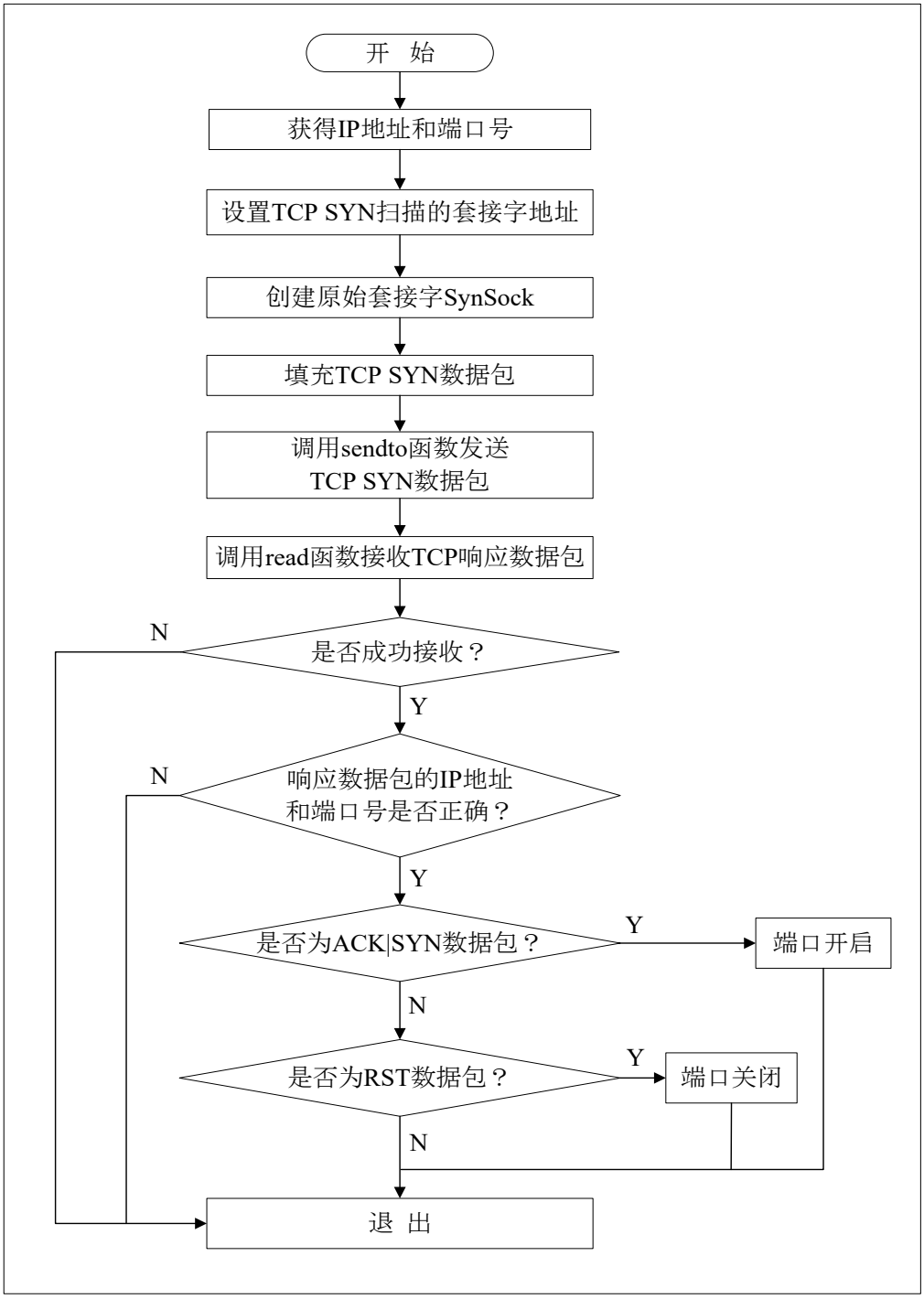

图 8-4 函数 Thread_TCPSYNHost 流程图

需要注意的是，在填充 TCP SYN 数据包的过程中调用了函数 in_cksum 计算校验和。该函数将 TCP 伪头部（12 字节），TCP 报头（20 字节），以及应用层数据（程序中为 0 字节）合在一起作为输入数据，将它们按 16 位进行分组计算校验和。同理，在填充 TCP FIN 数据包和 UDP 数据包时，也需要利用函数 in_cksum 计算校验和，只是输入的数据略有不同。函数 in_cksum 的代码如下：

```
unsigned short in_cksum(unsigned short *ptr, int nbytes)
{
```

```
    register long sum;
    u_short oddbyte;
    register u_short answer;
    sum = 0;
    while(nbytes > 1)
      {
        sum += *ptr++;
        nbytes -= 2;
    }
    if(nbytes == 1)
      {
        oddbyte = 0;
        *((u_char *) &oddbyte) = *(u_char *)ptr;
        sum += oddbyte;
    }
    sum = (sum >> 16) + (sum & 0xffff);
    sum += (sum >> 16);
    answer = ~sum;
    return(answer);
}
```

5. TCP FIN 扫描

TCP FIN 扫描的代码与 TCP SYN 扫描的代码基本相同。都利用原始套接字构造 TCP 数据包发送给目标主机的被测端口。不同的只是将 TCP 头 flags 字段的 FIN 位置 1。另外在接收 TCP 响应数据包时也略有不同。和前面两种扫描一样，TCP FIN 扫描也由两个线程函数构成。它们分别是 Thread_TCPFinScan 和 Thread_TCPFINHost。线程函数 Thread_TCPFinScan 负责遍历端口号，创建 TCP FIN 扫描子线程。它的流程与 TCP SYN 扫描相同，这里不继续介绍。重点介绍一下线程函数 Thread_TCPFINHost 的代码。

```
void* Thread_TCPFINHost(void* param)
{
    /*------------与 TCP SYN  扫描类似-----------------*/
    …
    //填充 TCP FIN 数据包
    …
    tcph->th_flags=TH_FIN;                    //TCP 头 flags 字段的 FIN 位置 1
    …
    //发送 TCP FIN 数据包
    …
    //将套接字设置为非阻塞模式
    if(fcntl(FinRevSock, F_SETFL, O_NONBLOCK) == -1)
    …
    //接收 TCP 响应数据包循环
```

```
        gettimeofday(&TpStart,NULL);                    //获得开始接收时刻
        do
        {
            //调用 recvfrom 函数接收数据包
            len = recvfrom(FinRevSock,recvbuf,sizeof(recvbuf),0,
            (struct sockaddr*)&FromAddr,(socklen_t*)&FromAddrLen);
            if(len > 0)
            {
                SrcIP = inet_ntoa(FromAddr.sin_addr);
                if(SrcIP == HostIP)                     //响应数据包的源地址等于目标主机地址
                {
                    …
                    //判断响应数据包的源地址是否等于目标主机地址，目的地址是
                    //否等于本机 IP 地址，源端口是否等于被扫描端口，目的端口是
                    //否等于本机端口号
                    if(HostIP == SrcIP && LocalIP == DstIP && SrcPort == HostPort &&
                       DstPort == LocalPort)
                    {
                        if (tcph->th_flags == 0x14)     //判断是否为 RST 数据包
                        {   ...
                            break;
                        }
                    }
                 }
            }
            //判断等待响应数据包时间是否超过 3 秒?
            gettimeofday(&TpEnd,NULL);
            TimeUse=(1000000*(TpEnd.tv_sec-TpStart.tv_sec)+
                        (TpEnd.tv_usec-TpStart.tv_usec))/1000000.0;
            if(TimeUse<3)
               continue;
            else
            {
                //超时，扫描端口开启
                break;
            }
        }
        while(true);

        //退出子线程
        …
    }
```

线程函数 Thread_TCPFINHost 的流程与线程函数 Thread_TCPSYNHost 基本相同。在填充 TCP FIN 数据包的时候，注意将 TCP 头的 flags 字段的 FIN 位设置为 1。调用 sendto 函数

发送完数据包之后，将套接字 FinRevSock 设置为非阻塞模式。这样 recvfrom 函数不会一直阻塞，直到接收到一个数据包为止，而是通过一个外部循环控制等待响应数据包的时间。如果超时，则退出循环。在收到一个数据包以后，如果该数据包的 a）源地址等于目标主机地址，b）目的地址等于本机 IP 地址，c）源端口等于被扫描端口，d）目的端口等于本机端口，那么该数据包为响应数据包。如果该数据包 TCP 头的 flags 字段的 RST 位为 1，那么就表示被扫描端口关闭；否则，继续等待响应数据包。如果等待时间超过 3 秒，那么就认为被扫描端口是开启的。

6. UDP 扫描

在本章中，UDP 扫描是通过线程函数 Thread_UDPScan 和普通函数 UDPScanHost 实现的。与前面介绍的 TCP 扫描不同，UDP 扫描没有采用创建多个子线程同时扫描多个端口的方式。这是因为目标主机返回的 ICMP 不可达数据包没有包含目标主机的源端口号，扫描器无法判断 ICMP 响应是从哪个端口发出的。因此，如果让多个子线程同时扫描端口，会造成无法区分 ICMP 响应数据包与其对应端口的情况。这样，判断被扫描端口是开启还是关闭就显得毫无意义了。为了保证扫描的准确性，必须牺牲程序的运行效率，逐次地扫描目标主机的被测端口。

与前面介绍的 TCP 扫描一样，线程函数 Thread_UDPScan 负责遍历目标主机端口，调用函数 UDPScanHost 对指定端口进行扫描。线程函数 Thread_UDPScan 的代码如下所示，在从启始端口（BeginPort）到终止端口（EndPort）的遍历中，逐次对当前端口（TempPort）进行 UDP 扫描。

```
void* Thread_UDPScan(void* param)
{
    …
    //遍历端口，逐次扫描
    for (TempPort=BeginPort;TempPort<=EndPort;TempPort++)
    {   …
        UDPScanHost(pUDPScanHostParam);
    }
    …
 }
```

函数 UDPScanHost 是 UDP 扫描的核心部分，它负责向被测端口发送 UDP 数据包，并等待接收 ICMP 响应数据包。在发送 UDP 数据包时可以采用两种方法：一种是创建数据报套接字（SOCK_DGRAM）并调用 sendto 函数发送 UDP 数据包；另一种是利用原始套接字（SOCK_RAW）构造一个 UDP 数据包，然后再调用 sendto 函数将该数据包发送给被测端口。本程序采用的是第二种方法，具体代码如下。

```
void UDPScanHost(struct UDPScanHostThrParam *p)
{
    /*变量定义*/
    //获得目标主机 IP 地址
    …
    //创建套接字 UDPSock
```

```
UDPSock=socket(AF_INET,SOCK_RAW,IPPROTO_ICMP);
…
//设置套接字 UDPSock 选项
ret = setsockopt(UDPSock,IPPROTO_IP,IP_HDRINCL,&on,sizeof(on));
…
//设置 UDP 套接字地址
memset(&UDPScanHostAddr,0,sizeof(UDPScanHostAddr));
UDPScanHostAddr.sin_family = AF_INET;
UDPScanHostAddr.sin_addr.s_addr = inet_addr(&HostIP[0]);
UDPScanHostAddr.sin_port = htons(HostPort);

//填充 UDP 数据包
memset(packet, 0x00, sizeof(packet));
ip = (struct iphdr *)packet;
udp = (struct udphdr *)(packet + sizeof(struct iphdr));
pseudo = (struct pseudohdr *)(packet + sizeof(struct iphdr) - sizeof(struct pseudohdr));

//填充 UDP 头
udp->source = htons(LocalPort);
udp->dest = htons(HostPort);
udp->len = htons(sizeof(struct udphdr));
udp->check = 0;

//填充 UDP 伪头部，用于计算校验和
pseudo->saddr = LocalHostIP;
pseudo->daddr = inet_addr(&HostIP[0]);
pseudo->useless = 0;
pseudo->protocol = IPPROTO_UDP;
pseudo->length = udp->len;
udp->check = in_cksum((u_short *)pseudo,
                      sizeof(struct udphdr)+sizeof(struct pseudohdr));

//填充 IP 头
ip->ihl = 5;
ip->version = 4;
ip->tos = 0x10;
ip->tot_len = sizeof(packet);
ip->frag_off = 0;
ip->ttl = 69;
ip->protocol = IPPROTO_UDP;
ip->check = 0;
ip->saddr = LocalHostIP;
ip->daddr = inet_addr(&HostIP[0]);
```

```
        //发送 UDP 数据包
        n = sendto(UDPSock, packet, ip->tot_len, 0,
                    (struct sockaddr *)&UDPScanHostAddr, sizeof(UDPScanHostAddr));
        …
        //设置套接字 UDPSock 为非阻塞模式
        if(fcntl(UDPSock, F_SETFL, O_NONBLOCK) == -1)
        …
        //接收 ICMP 相应数据包循环
        gettimeofday(&TpStart,NULL);                              //获得接收起始时间
        do
        {
            //接收 ICMP 数据包
            n = read(UDPSock, (struct ipicmphdr *)&hdr, sizeof(hdr));
            if(n > 0)
            {
                //判断 ICMP 数据包的源地址是否等于目标主机地址，code 字段和
                //type 字段的值是否是 3
                if((hdr.ip.saddr == inet_addr(&HostIP[0])) && (hdr.icmp.code == 3) &&
                    (hdr.icmp.type == 3))
                {
                    /*UDP 端口关闭*/
                    break;
                }
            }
            //判断等待时间是否超过 3 秒
            gettimeofday(&TpEnd,NULL);
            TimeUse=(1000000*(TpEnd.tv_sec-TpStart.tv_sec)+
                        (TpEnd.tv_usec-TpStart.tv_usec))/1000000.0;
            if(TimeUse<3)
                continue;
            else
            {
                /*UDP 端口开启*/
                break;
            }
        } while(true);

        //关闭套接字
        close(UDPSock);
        delete p;
    }
```

如图 8-5 所示， UDPScanHost 函数的流程分为以下 8 个步骤。

(1) 获得目标主机 IP 地址和扫描端口号，以及本机 IP 地址和端口号。

(2) 创建原始套接字 UDPSock。

(3) 设置套接字 UDPSock 的选项。

(4) 设置 UDP 扫描的套接字地址。

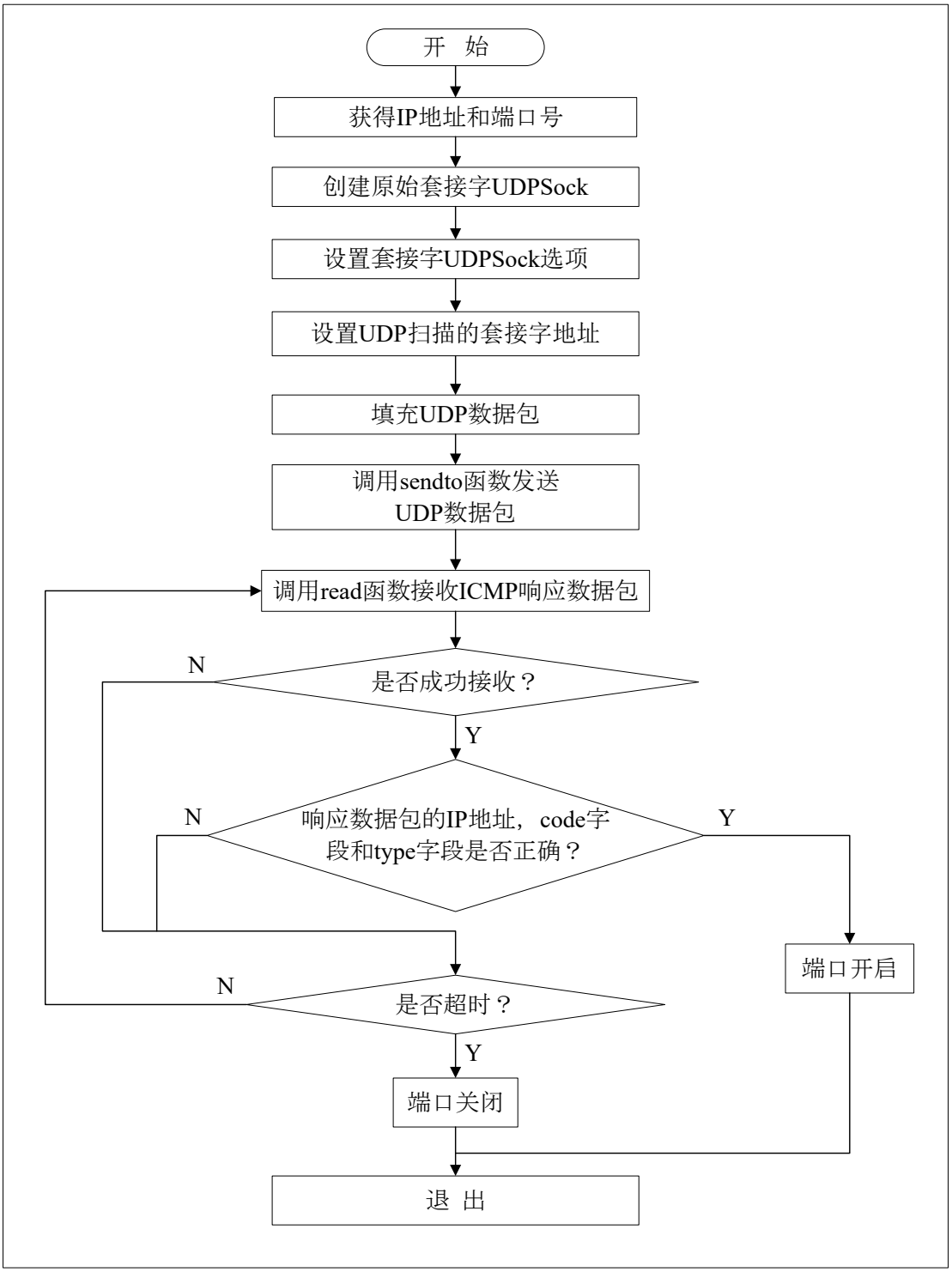

图 8-5 函数 UDPScanHost 流程图

(5) 填充 UDP 数据包。注意 UDP 头的校验和字段调用函数 in_cksum 进行计算。

(6) 调用 sendto 函数向目标主机的指定端口发送 UDP 数据包。

(7) 调用 read 函数接收目标主机的 ICMP 响应数据包。若函数的返回值大于 0，则成功接收一个数据包。如果该响应数据包的源地址等于目标主机地址，code 字段和 type 字段的值都为 3，那么该数据包就是目标主机被扫描端口返回的 ICMP 不可达数据包。因此，可以认为被扫描的 UDP 端口是关闭的。若接收时间超过 3 秒，则认为被扫描的 UDP 端口开启，退出循环。

(8) 关闭套接字 UDPSock，返回。

7. 编写 makefile 文件

本程序包含了多个.cpp 文件，在编译和链接的时候，逐行地输入重复的命令会显得十分繁琐。稍有疏忽，就会产生错误。为了解决这个问题，我们可以在这些代码文件的同一目录下创建一个 makefile 文件，每次编译时，只需在 Shell 命令行中输入 make 命令即可。本程序的 makefile 文件内容如下所示。由于在本程序中使用了多线程编程技术，所以在最后链接生成可执行代码时应该加上参数-pthread 才能通过。在 makefile 文件中还使用了 make clean 命令对中间生成的文件做必要的清理，方便下一次编译。只需在 Shell 命令行中输入 make clean 就可以删除在编译连接时生成的文件。

```
Scaner:Scaner.o TCPConnectScan.o TCPSYNScan.o TCPFINScan.o UDPScan.o
    g++ -lpthread -o Scaner Scaner.o TCPConnectScan.o TCPSYNScan.o TCPFINScan.o
UDPScan.o
Scaner.o:Scaner.cpp
    g++ -c Scaner.cpp
TCPConnectScan.o:TCPConnectScan.cpp
    g++ -c TCPConnectScan.cpp
TCPSYNScan.o:TCPSYNScan.cpp
    g++ -c TCPSYNScan.cpp
TCPFINScan.o:TCPFINScan.cpp
    g++ -c TCPFINScan.cpp
UDPScan.o:UDPScan.cpp
    g++ -c UDPScan.cpp
clean:
    rm Scaner
    rm Scaner.o
    rm TCPConnectScan.o
    rm TCPSYNScan.o
    rm TCPFINScan.o
    rm UDPScan.o
```

8.4 扩展与提高

8.4.1 ICMP 扫描扩展

ICMP 扫描就是利用 8.2.1 小节介绍的 ping 程序判断目标主机是否可达。但是这种传统的 ICMP 扫描方式容易被防火墙过滤。如果才能绕开防火墙探测它后面的主机呢？ICMP 扩展扫描方式就做到了这一点。

ICMP 扩展扫描利用了 ICMP 协议最基本的用途——报错。根据 TCP/IP 协议，如果在通信过程中出现了错误，那么接收端将产生一个 ICMP 的错误报文，用来通告发送端错误的相关信息。这些错误报文是根据 ICMP 协议要求由探测系统自动产生的，一般不会被防火墙拦截。

下面列举出 ICMP 扩展扫描的 4 种实现方式：

(1) 向目标主机发送一个只有 IP 头的 IP 数据报，目标主机将返回 Destination Unreachable 的 ICMP 错误报文。

(2) 向目标主机发送一个错误的 IP 数据报，比如，IP 头长度错误。目标主机将返回 Parameter Problem 的 ICMP 错误报文。

(3) 当数据包分片，但是却没有给接收端足够的分片，接收端分片组装超时会发送分片组装超时的 ICMP 错误报文。

(4) 向目标主机发送一个 IP 数据报，但是协议项是错误的，比如协议项不可用，那么目标将返回 Destination Unreachable 的 ICMP 错误报文。

此外，扩展 ICMP 扫描还用 2 种其他功能。

(1) 探测防火墙的存在：将数据包的 IP 头协议字段填入一个至今无人使用的较大的值，向确定存在的主机发送该数据包，若收不到响应，则证明目标主机有防火墙保护。

(2) 探测目的主机运行协议：因为 IP 头协议字段的长度为 8 位，所以只有 256 种协议的可能。遍历这 256 种可能的协议，探测目标主机，便可穷举出目标主机当前运行的协议。

8.4.2 TCP 扫描扩展

在 8.2.2 小节介绍的 TCP FIN 扫描属于秘密扫描的一种，所谓秘密扫描就是在扫描过程中完全不涉及 TCP 连接建立过程，确保系统不会记录任何日志的扫描技术。下面先阐述秘密扫描一些理论知识，然后再介绍其它几种秘密扫描方式。

1. 理论基础

(1) 当一个 SYN 或者 FIN 数据包到达一个关闭的端口，根据 TCP 协议，该端口在丢弃数据包的同时，发送一个 RST 数据包。

(2) 当一个 RST 数据包到达一个监听端口，RST 数据包被丢弃。

(3) 当一个 RST 数据包到达一个关闭端口，RST 数据包被丢弃。

(4) 当一个包含 ACK 的数据包到达一个监听端口时，数据包被丢弃，同时发送一个 RST 数据包。

(5) 当一个 SYN 数据包到达一个监听端口时，正常的三阶段握手继续，回应一个 ACK|SYN 数据包。

(6) 当一个 FIN 数据包到达一个监听端口时，数据包被丢弃。

(7) 关闭的端口返回 RST，监听端口丢弃包的行为在 URG 和 PSH 标志位时同样要发生。所有的 URG、PSH 和 FIN，或者没有任何标记的 TCP 数据包都会引发该行为。

2. 其它扫描方式

(1) ACK 扫描：发送一个只有 ACK 标志的 TCP 数据包给主机，如果主机反馈一个 TCP RST 数据包来，那么这个主机是存在的。也可以通过这种技术来确定对方防火墙仅仅是简单的分组过滤，还是基于状态的防火墙。

(2) NULL 扫描：发送一个没有任何标志位的 TCP 数据包，根据 RFC793，如果目标主机的相应端口是关闭的话，应该返回一个 RST 数据包。

(3) FIN+URG+PUSH 扫描：向目标主机发送一个 FIN、URG 和 PUSH 分组，根据 RFC793，如果目标主机的相应端口是关闭的话，应该返回一个 RST 数据包。

秘密扫描虽然种类繁多，但是大都依赖操作系统网络协议栈的实现细节。因此，秘密扫描并非对所有操作系统有效。针对不同的操作系统，只有选择合适的扫描方式，才能达到扫描预期的目标。

此外，TCP 扫描还有一种扩展方式叫作间接扫描，也就是扫描发起主机冒充某台不相关的主机，以其 IP 地址填充扫描所需数据包源地址，从而实现隐藏自身的目的。此方式的原理比较简单，不再赘述。

8.4.3 系统漏洞扫描简介

漏洞扫描最初是以黑客攻击技术出现的。黑客通过自己编写或利用程序来检测待攻击目标是否存在特定的漏洞，以便发动有效的攻击。随着扫描技术的发展和漏洞资料库的逐步完善，进而出现了专业的安全评估工具——漏洞扫描器。

漏洞扫描器通常分为主机型和网络型两大类。主机型漏洞扫描有时称被动扫描，它采用非破坏性的方法对系统的文件属性，操作系统的安全补丁，帐户设置，服务配置以及应用程序等进行安全检测。由于是以一定的用户权限进行检测的，所以它能够准确地定位系统的问题，发现漏洞。网络型漏洞扫描器通过模拟黑客攻击方式来进行安全漏洞检测。扫描器先采用定制的脚本模拟攻击系统，然后对结果进行分析，看是否与漏洞库存储的规则匹配。如果匹配，则存在安全漏洞。由此可见，扫描器漏洞资料库的完善与否直接影响到扫描器对漏洞的检测能力。

目前的漏洞扫描产品大部分都属于网络型，同时它们也吸取了主机型扫描器的优点。它

们既可以对网络上的服务器进行远程漏洞检测，又可以对特定主机进行更深层次的安全检测。常见的漏洞扫描产品有 SSS、ISS、eEye、RETINA、NAI CYBERCOP、Nmap 等。

网络漏洞扫描器对目标系统进行漏洞检测时，首先探测目标系统的存活主机，对存活主机进行端口扫描，确定系统开放的端口，同时根据协议指纹技术识别出主机的操作系统类型。然后扫描器对开放的端口进行网络服务类型的识别，确定其提供的网络服务。漏洞扫描器根据目标系统的操作系统平台和提供的网络服务，调用漏洞资料库中已知的各种漏洞进行逐一检测，通过对探测响应数据包的分析判断是否存在漏洞。

现有的网络漏洞扫描器主要是利用特征匹配的原理来识别各种已知的漏洞。扫描器发送含有某一漏洞特征探测码的数据包，根据返回数据包中是否含有该漏洞的响应特征码来判断是否存在漏洞。例如，对于 IIS 中的 Unicode 目录遍历漏洞，扫描器只要发送含有特征代码%c1%1c 的探测包：http://x.x.x.x/scrlpts/..%c1%1c../winnt/system32/cmdexe?/c+dir，如果应答数据包中含有 200 OK 则可以断定该漏洞存在。

由此可见，漏洞扫描是一项基于漏洞数据库进行特征比对的扫描技术，在系统安全检测方面具有广泛的应用前景。

8.4.4 Linux 下 Nmap 的安装与使用

1. Nmap 简介

网络映射器（Network Mapper，Nmap）是一款开放源代码的网络探测和安全审计程序。系统管理员与用户不仅可以使用 Namp 快速地扫描大型网络，发现网络上哪些主机正在运行，而且可以进一步探测这些主机所提供什么服务，操作系统的相关信息，以及报文过滤器或防火墙的类型等等。虽然 Nmap 通常用于安全审计，但是许多系统管理员和网络管理员也用它来做一些日常工作，比如查看整个网络的信息，管理服务升级计划，以及监视主机和服务的运行。

Nmap 支持多种协议的扫描，比如 UDP、TCP connect、TCP SYN (half open)、FIN、ACK sweep、ICMP (ping sweep)、ftp proxy (bounce attack)、Reverse-ident、Xmas Tree 和 Null 扫描。Nmap 还提供了一些高级功能，例如通过 TCP/IP 协议栈特征探测操作系统类型，秘密扫描，动态延时和重传计算，并行扫描，通过并行 ping 扫描探测关闭的主机，诱饵扫描，避开端口过滤检测，直接 RPC 扫描(无须端口映射)，碎片扫描，以及灵活的目标和端口设定。在 Linux 系统中，非 root 用户可以利用 Nmap 完成许多工作，但不幸的是关键的核心功能（比如 raw socket）需要 root 权限才能运行。

2. Nmap 安装

许多操作系统支持 Nmap 的安装，比如 Linux，Microsoft Windows，Mac OS X，Sun Solaris，OpenBSD 等等。根据不同的操作系统，Nmap 提供了多种安装方式。其中，将源代码进行编译并安装是一种传统而有效的安装方式。本章将介绍采用这种方式来安装 Nmap 的过程。

表 8-2　Nmap 与操作系统信息

项目	说明

Nmap 安装文件	nmap-4.85BETA8.tar.bz2
操作系统版本	ubuntu-8.10

(1) 准备工作

如表 8-2 所示，本章选择 Nmap-4.85BETA 版，可以从 http://nmap.org/download.html 下载相关的源代码压缩包。在安装 Nmap 之前，需要检查 ubuntu 是否安装了 g++编译器。如果没有，以 root 身份登陆系统，在终端或 shell 命令行中执行命令“apt-get install g++”，系统就会自动地下载并安装 g++。

(2) 安装步骤

a. 在完成上述准备工作之后，就可以开始 Nmap 的安装过程。具体操作步骤如下：

b. 以 root 身份登陆系统，或者输入“su root”转换为 root 用户。

c. 在终端或 shell 命令行中执行命令“tar xvjf nmap-4.85BETA8.tar.bz2”，将 nmap 软件包的内容解压至同一目录下的文件夹 nmap-4.85BETA8 中。

d. 执行命令“cd nmap-4.85BETA8”进入解压后的文件夹。

e. 执行命令“./configure”对当前 Linux 系统进行配置。如果配置成功，在命令行中会显示一条由二进制字符组成的喷火巨龙，如图 8-6 所示。

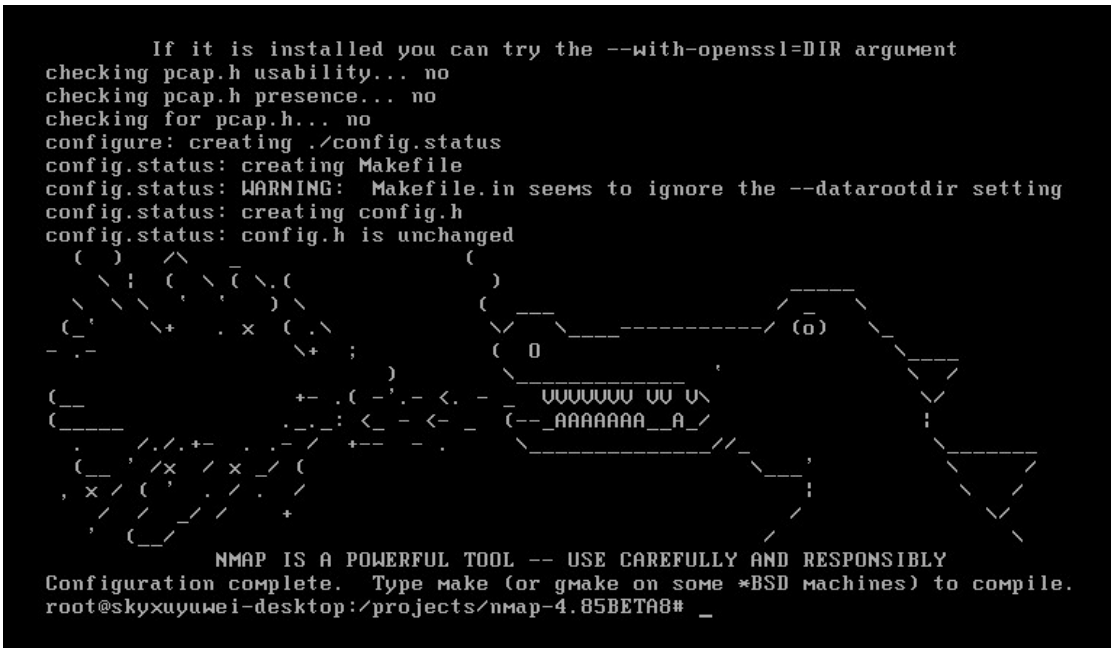

图 8-6 Nmap 配置成功界面

f. 执行命令“make”对 Nmap 进行编译。

g. 执行命令“make install”，将 Nmap 安装到/usr/local/bin/nmap 目录下。

3. Nmap 使用

Nmap 的语法如下：

nmap [扫描类型] [功能选项] [目标说明]

(1) 扫描类型

Nmap 支持的十几种扫描技术。，除了 UDP 扫描(-sU)可能和任何一种 TCP 扫描类型结合使用外，一般一次只使用一种方法进行扫描。扫描类型的格式是-s[*]，其中[*]是一个字符，表示特定的扫描类型。表 8-2 列出了各种扫描类型所对应的格式。其中，deprecated FTP bounce 扫描(-b) 是一个例外。

表 8-2 Nmap 扫描类型列表

扫描类型	说明	扫描类型	说明
-sS	TCP SYN 扫描	-sT	TCP connect 扫描
-sU	UDP 扫描	-sN	TCP NULL 扫描
-sF	TCP FIN 扫描	-sX	TCP XmasTree 扫描
-sA	TCP ACK 扫描	-sW	TCP 窗口扫描
-sM	TCP Maimon 扫描	-sO	IP 协议扫描
-b	FTP 弹跳扫描		

(2) 功能选项

Nmap 的功能选项可以组合使用。一些功能选项只能够在某种特定的扫描模式下使用。Nmap 会自动识别无效或者不支持的功能选项组合，并向用户发出警告信息。因为 Nmap 的功能选项种类繁多，所以本章不再逐一进行详细介绍。请登陆 Nmap 的官方网站查阅相关内容。网址：http://nmap.org/book/man-briefoptions.html。

(3) 目标说明

除扫描类型和功能选项以外，Nmap 命令中剩余的部分都被视为对目标主机的说明。以 TCP SYN 扫描为例，最简单的情况莫过于只指定一个目标 IP 地址或主机名。扫描结果如下所示。

```
root@skyxuyuwei-desktop:~# nmap -sS 192.168.1.1

Starting Nmap 4.85BETA8 ( http://nmap.org ) at 2009-04-27 22:07 CST
Interesting ports on 192.168.1.1:
Not shown: 999 closed ports
PORT      STATE SERVICE
23/tcp open    telnet
MAC Address: 00:E0:FC:04:01:AA (Huawei Technologies CO.)

Nmap done: 1 IP address (1 host up) scanned in 10.92 seconds
```

如果希望扫描整个网络的相邻主机，那么目标说明可以采用一个 CIDR 风格的地址。只要在一个 IP 地址或主机名后面加上“/<numbit>”， Nmap 就会扫描所有与该参考 IP 地址具有 <numbit>位相同比特的所有 IP 地址或主机。<numbit>所允许的最小值是 1，这将会扫描半个互联网；最大值是 32，这将会扫描该主机或 IP 地址。例如，192.168.1.0/24 将会扫描 192.168.1.0 (二进制格式: 11000000 10101000 00000001 00000000)和 192.168.10.255 (二进制格式: 11000000 10101000 00000001 11111111)之间的 256 台主机。以 TCP SYN 扫描为例，扫

描结果如下所示。

```
root@skyxuyuwei-desktop:~# nmap -sS 192.168.1.0/24

Starting Nmap 4.85BETA8 ( http://nmap.org ) at 2009-04-27 22:11 CST
Interesting ports on 192.168.1.1:
Not shown: 999 closed ports
PORT      STATE SERVICE
23/tcp open    telnet
MAC Address: 00:E0:FC:04:01:AA (Huawei Technologies CO.)

Interesting ports on 192.168.1.23:
Not shown: 996 filtered ports
PORT         STATE    SERVICE
139/tcp    open      netbios-ssn
445/tcp    open      microsoft-ds
912/tcp    open      unknown
2869/tcp closed unknown
MAC Address: 00:21:9B:13:4C:0F (Dell)

Interesting ports on 192.168.1.122:
Not shown: 999 filtered ports
PORT       STATE SERVICE
912/tcp open    unknown
MAC Address: 00:24:8C:0C:9F:DD (Unknown)

Interesting ports on 192.168.1.136:
Not shown: 981 closed ports
PORT         STATE SERVICE
25/tcp      open    smtp
53/tcp      open    domain
80/tcp      open    http
88/tcp      open    kerberos-sec
135/tcp    open    msrpc
139/tcp    open    netbios-ssn
389/tcp    open    ldap
445/tcp    open    microsoft-ds
464/tcp    open    kpasswd5
593/tcp    open    http-rpc-epmap
636/tcp    open    ldapssl
1025/tcp open    NFS-or-IIS
1027/tcp open    IIS
1037/tcp open    unknown
1038/tcp open    unknown
1041/tcp open    unknown
```

```
1050/tcp open    java-or-OTGfileshare
3268/tcp open    globalcatLDAP
3269/tcp open    globalcatLDAPssl
MAC Address: 00:0C:29:77:52:83 (VMware)

Interesting ports on 192.168.1.158:
Not shown: 996 filtered ports
PORT        STATE    SERVICE
139/tcp    open      netbios-ssn
445/tcp    open      microsoft-ds
2869/tcp closed unknown
3389/tcp open      ms-term-serv
MAC Address: 00:16:76:A9:55:3A (Intel)

Interesting ports on 192.168.1.222:
Not shown: 997 filtered ports
PORT        STATE    SERVICE
139/tcp    open      netbios-ssn
445/tcp    open      microsoft-ds
2869/tcp closed unknown
MAC Address: 00:24:8C:0D:58:0B (Unknown)

Interesting ports on 192.168.1.244:
Not shown: 999 closed ports
PORT      STATE SERVICE
22/tcp open    ssh

Nmap done: 255 IP addresses (7 hosts up) scanned in 36.59 seconds
```

CIDR 标志位虽然非常简洁，但有时候却显得不够灵活。例如，扫描 192.168.0.0/16，但略过任何以.0 或者.255 结束的 IP 地址（通常为广播地址）。 Nmap 通过设定每 8 位 IP 地址的范围支持这种扫描。用户可以用“-”分开的数字或范围列表为 IP 地址的每 8 位组指定范围。例如，192.168.0-255.1-254 将略过在该范围内以.0 和.255 结束的地址。范围设定不限于 IP 地址的最后 8 位。例如，0-255.0-255.13.37 将在整个互联网范围内扫描所有以 13.37 结束的地址。这种大范围的扫描对互联网调查研究也许有用。

此外，目标说明不仅局限于命令行指定的方式，还可以通过选项-iL 从列表中输入。如果用户希望对互联网中的主机进行探测，那么可以通过-iR 选项随机地选择目标主机。