网络安全技术

实
验
报
告

学　　院 网络空间安全学院

年　　级 2021

学　　号 2112060

姓　　名 孙蕗

目录

一、实验目的

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。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。

③ 熟练掌握 Linux 环境下的套接字编程技术，掌握 Linux 环境下多线程编程的基本方法

④ 设计并实现 ping 程序，探测目标主机是否可达。

二、实验要求

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。

Scanner 程序流程如下：

a. 首先判断是否需要输出帮助信息，如果需要输出帮助信息，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。

```
[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
```

b. 用户输入被扫描主机的 IP 地址，扫描起始端口和终止端口。

c. 判断 IP 地址是否与端口号是否错误，若错误，则提示用户并退出，否则继续执行下面的步骤。

d. 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，继续执行下面的步骤。

e. 判断是否进行 TCP connect 扫描（控制台命令行输入./Scanner -c），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，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。否则，继续执行下面的步骤。

```
[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 !
...
```

f. 判断是否进行 TCP SYN 扫描（控制台命令行输入./Scanner -s），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，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。否则，继续执行下面的步骤。

```
[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 !
...
```

g. 判断是否进行 TCP FIN 扫描（控制台命令行输入./Scanner -f），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，。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。否则，继续执行下面的步骤。

```
[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 !
...
```

h. 判断是否进行 UDP 扫描（控制台命令行输入./Scanner -u），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，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。否则，继续执行下面的步骤。

```
[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 !
...
```

i. 等待所有扫描子线程返回后退出。

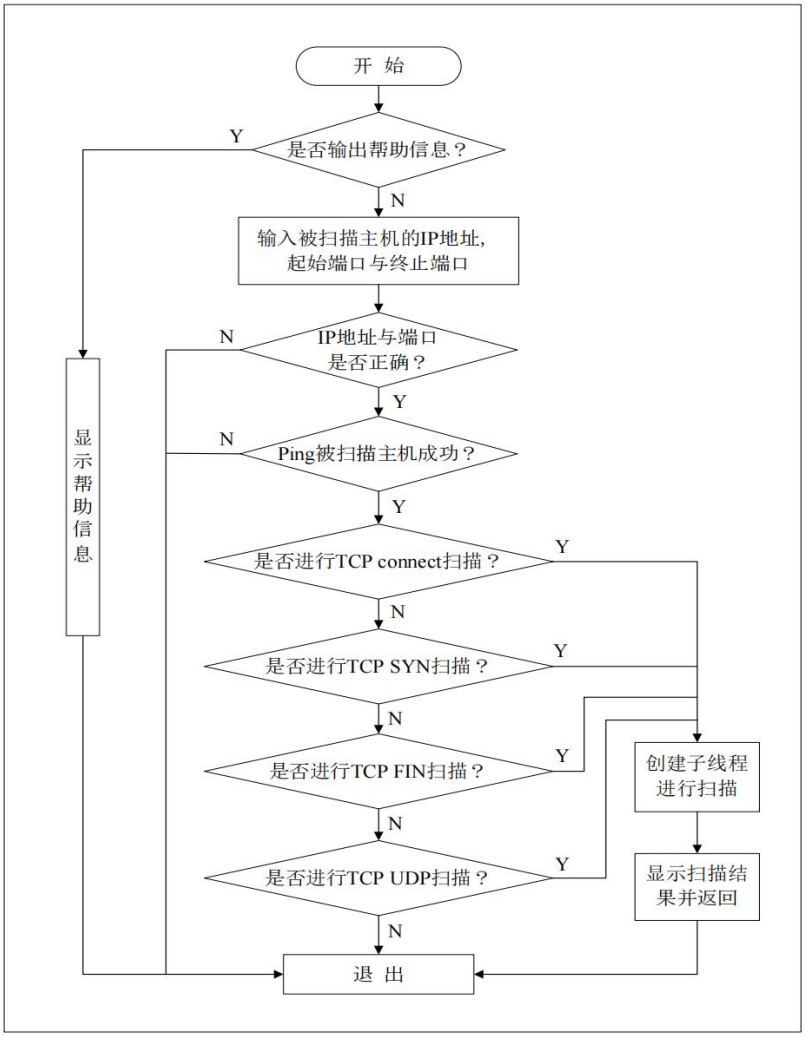
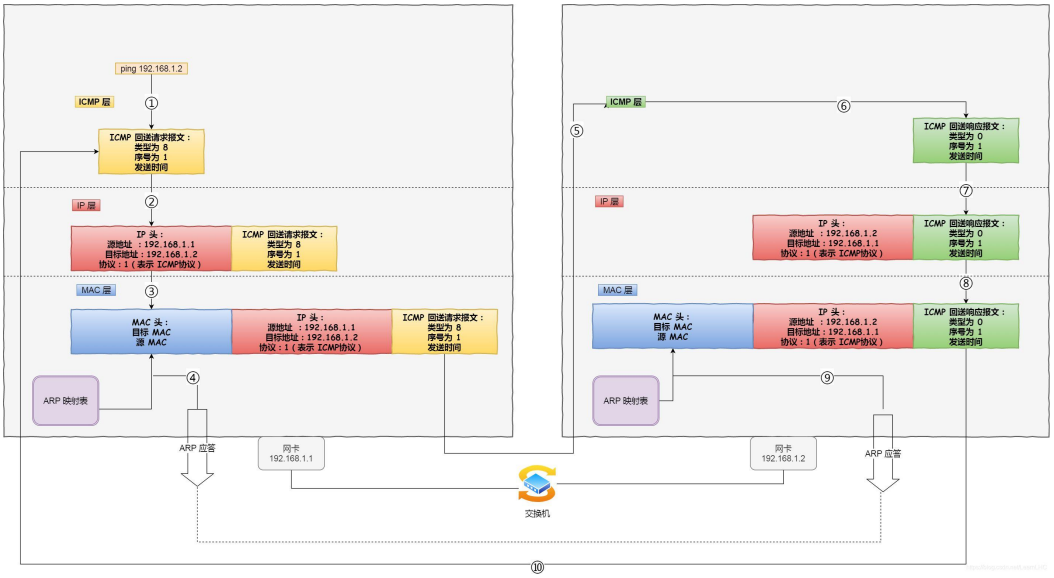


图 8-1 Scaner 程序流程图

三、实验内容

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。ping 命令使用网络层的 ICMP 协议。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。然后将数据包连同目的地址交给 IP 层，IP 层将本机地址作为源地址，将接收到的目的地址作为目的地址，协议字段设置为 1 表示 ICMP 协议，再加上其他信息构建一个 IP 数据包。通过发送 ARP 协议查询找到目的地址 IP 地址对应的 MAC 地址，由数据链路层构建一个数据帧，其中目的地址是 IP 层传过来的 MAC 地址，源地址则是本机的 MAC 地址。目标主机接收到数据帧后，先检查它的目的 MAC 地址是否是本机的 MAC 地址对比，如符合，则接收。接收后检查该数据帧，将 IP 数据包从帧中提取出来，交给本机的 IP 层。同样，IP 层检查后，将有用的信息提取后交给 ICMP 协议。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。



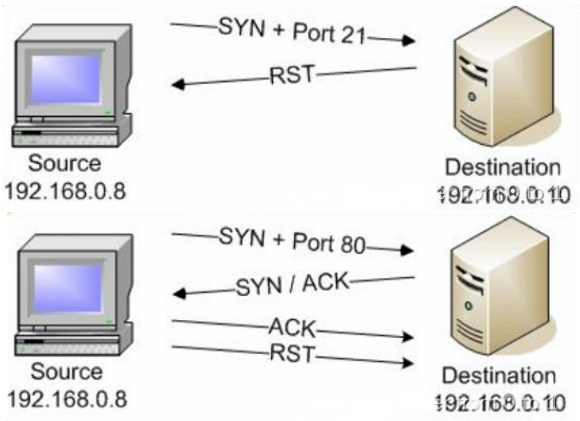
2. 与 TCP 相关的三种扫描

（1） Connect 扫描

扫描发起主机需要调用系统 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。

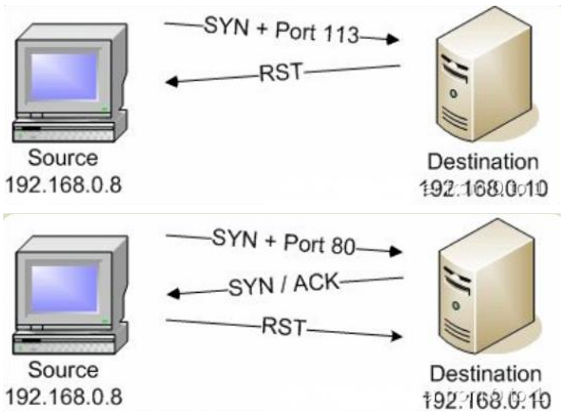
由于 TCP 协议是可靠协议，connect 系统调用不会在尝试发送第一个 SYN 包未得到响应的情况下就放弃，而是会经过多次尝试后才彻底放弃，因此需要较长的时间。此外，connect 失败会在系统中造成大量连接失败日志，容易被管理员发现。



（2） SYN 扫描

TCP SYN 扫描会向目标主机待扫描端口发送 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。

由于 SYN 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）。

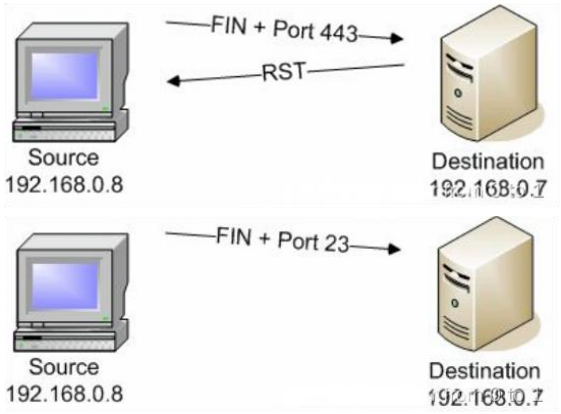


（3） FIN 扫描

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。

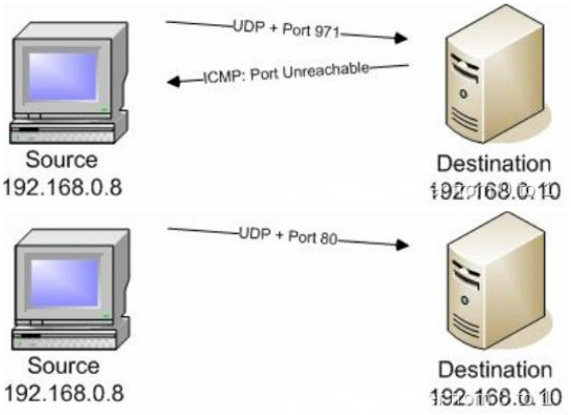
FIN 扫描具有良好的隐蔽性，不会留下日志。但是它的应用具有很大的局限性，由于不同系统实现网络协议栈的细节不同，FIN 扫描只能扫描 Linux/UNIX 系统。对于 Windows 系统而言，由于无论端口开放与否，都会返回 RST 数据包，因此对端口的状态无法进行判断。



3. UDP 扫描

UDP 扫描会向目标主机被扫描端口发送 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。

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。



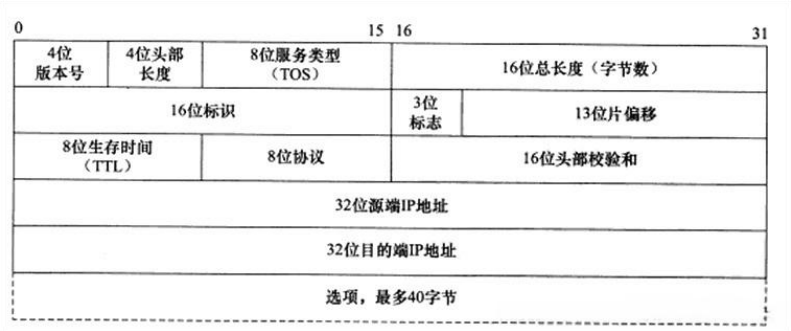
4. 常用协议的报头结构

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（IP 协议头），tcphdr（TCP 协议头），udphdr（UDP 协议头），icmphdr （ICMP 协议头）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。

（1） IP 协议头




```
//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;
};
```

（2） TCP 协议头

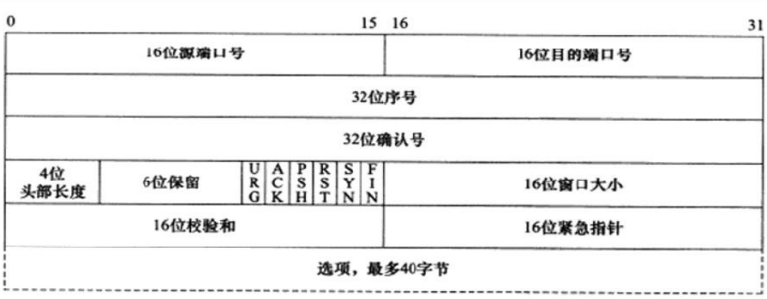


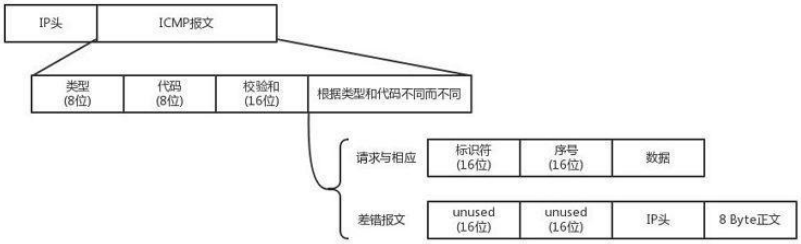
图 3-3 TCP 头部结构

```
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
};
```

（3） UDP 协议头

```
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
};
```

（4） ICMP 协议头




```
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;
};
```

四、 实验步骤

1. 端口扫描器 Scanner

端口扫描器程序 Scanner 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。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。

2. ping 程序

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 函数完整流程如下图所示。

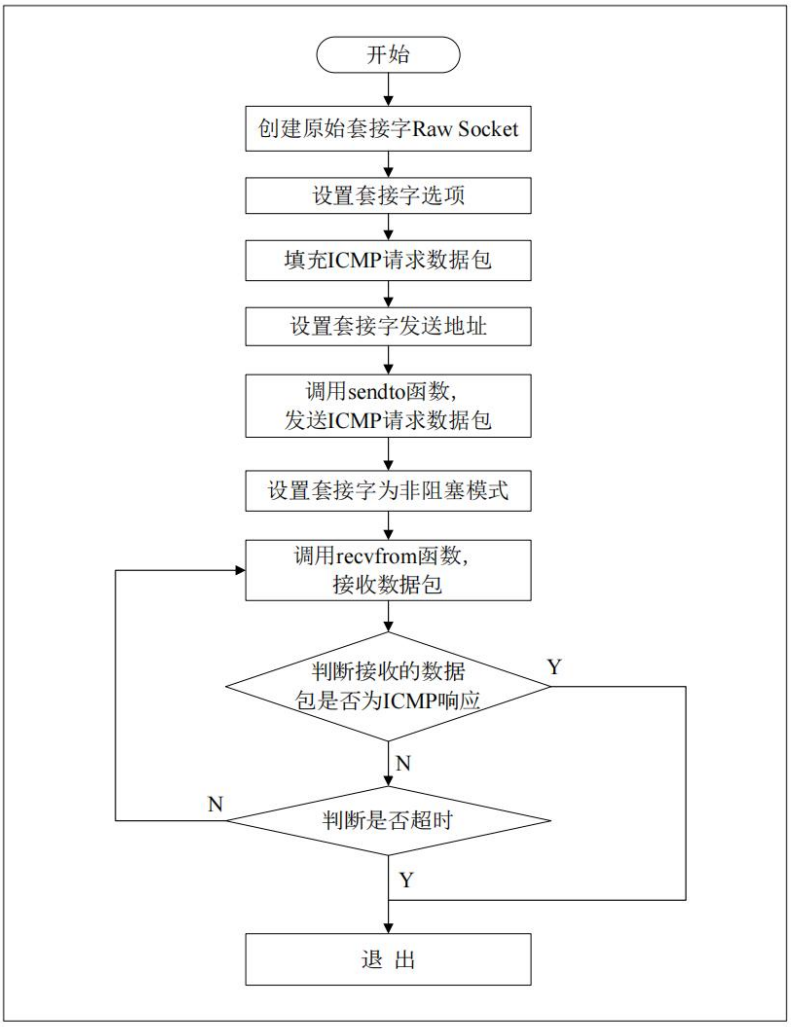


图 8-2　Ping 函数流程图

在 Scanner.h 文件中实现 Ping 函数，Ping 函数的流程可分为以下 8 个步骤。

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

```
//创建套接字
PingSock = socket(AF_INET,SOCK_RAW,IPPROTO_ICMP);
if (PingSock < 0)
{
    cout<<"Creat Ping socket error !"<<endl;
    printf("Errno: %d\n",errno);
    return false;
}
```

b. 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。

```
//设置套接字选型
on = 1;
ret = setsockopt(PingSock, 0, IP_HDRINCL, &on, sizeof(on));
if (ret < 0)
{
    cout<<"Bind Ping socket option error !"<<endl;
    return false;
}
```

c. 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。

```
//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头部
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));
```

d. 设置套接字 PingSock 的目标主机地址。

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

e. 调用 sendto 函数，发送 ICMP 请求数据包。

```
//发送ICMP请求
ret = sendto(PingSock,SendBuf,SendBufSize,0,(struct sockaddr*)&PingHostAddr,sizeof(PingHostAddr));
if(ret < 0)
{
    cout<<"Send ping packet failed !"<<endl;
    return false;
}
```

f. 调用函数 fcntl 将套接字设置为非阻塞模式。

```
    //将套接字设置为非阻塞模式
    if(fcntl(PingSock, F_SETFL, O_NONBLOCK) == -1)
    {
        cout<<"Set socket in non-blocked model fail !"<<endl;
        return false;
    }
```

g. 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。

a）do while 循环等待接收 ICMP 响应。调用函数 recvfrom，循环等待接收 ICMP 响应数据包。

```
    //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);
```

b）对接收到的 ICMP 响应数据包判断源地址是否等于目标主机的 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。

```
    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ICMP响应数据包IP头的源地址
            DstIP = inet_ntoa(Recvip->ip_dst);        // 获得ICMP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地址
            // 本机IP地址，ICMP头type字段是否为ICMP_ECHOREPLY
            if (SrcIP == HostIP && DstIP == LocalIP && Recvicmp->icmp_type == ICMP_ECHOREPLY)
            {
                cout<<"Ping Host "<<HostIP<<" Successfully !"<<endl;
                flags =true;//PING成功，退出循环
                break;
            }
        }
```

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

```
        //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)
        {
            continue;
        }
        else
        {
            flags = false;
            break;
        }

    } while(true);

    return flags;
}
```

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。TCP connect 扫描通过两个线程函数 ThreadTCPconnectScan 和 ThreadTCPconnectHost 实现。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所示。

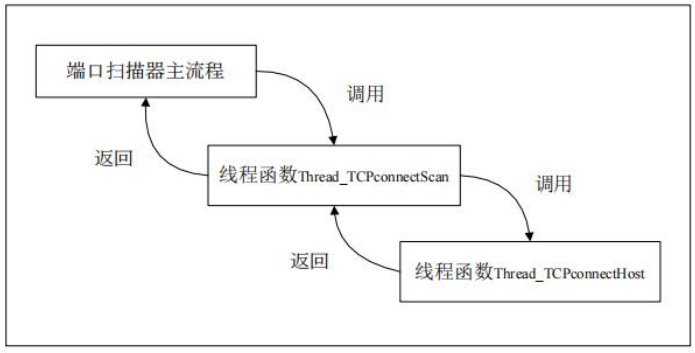


图 8-3 主流程，Thread_TCPconnectScan 和 Thread_TCPconnectHost 之间的关系

两个线程函数的代码如下。

（1） ThreadTCPconnectScan 函数

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。

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。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，修改完毕后再解锁以保证 TCPConThrdNum 的正确性。

```
int TCPConThrdNum;
//locker
pthread_mutex_t TCPConPrintlocker=PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t TCPConScanlocker=PTHREAD_MUTEX_INITIALIZER;
```

在函数中进行变量定义，获取目标主机 IP，起始端口，终止端口的等信息。将记录已创建的子线程数的变量 TCPConThrdNum 设为 0。

```
void* Thread_TCPconnectScan(void* param)
{
    //变量定义
    //获得扫描的目标主机IP，起始端口， 终止端口
    struct TCPConThrParam *p;
    string HostIP;
    unsigned BeginPort,EndPort,TempPort;

    pthread_t subThreadID;
    pthread_attr_t attr;
    int ret;
    p=(struct TCPConThrParam*)param;
    HostIP = p->HostIP;
    BeginPort = p->BeginPort;
    EndPort = p->EndPort;
    TCPConThrdNum = 0;
```

从起始端口到终止端口循环扫描目标主机的端口。设置子线程参数，将子线程设为分离状态。创建 connect 目标主机制定的端口子线程。在函数 Thread_TCPconnectHost 退出当前线程之前，将 TCPConThrdNum 加 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。

```
//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;
    //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);
    if (ret==-1)
    {
        cout<<"Can't create the TCP connect Host thread !"<<endl;
    }
    pthread_attr_destroy(&attr);

    pthread_mutex_lock(&TCPConScanlocker);
    TCPConThrdNum++;//线程数加 1
    pthread_mutex_unlock(&TCPConScanlocker);

    while (TCPConThrdNum>100)//如果子线程数大于 100，暂时休眠
    {
        sleep(3);
    }
}
```

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。

```
//等待所有子线程数为0
while (TCPConThrdNum != 0)
{
    sleep(1);
}
cout<<"TCP Connect Scan thread exit !"<<endl;
pthread_exit(NULL);
}
```

（2） ThreadTCPconnectHost 函数

ThreadTCPconnectHost 函数用来连接（connect）目标主机指定端口。函数 ThreadTCPconnectHost 的流程如下：

在函数中先进行一些变量的定义。

```
void* Thread_TCPconnectHost(void* param)
{
    //变量定义
    struct TCPConHostThrParam *p;
    string HostIP;
    unsigned HostPort;
    int ConSock;
    struct sockaddr_in HostAddr;
    int ret;
```

获取线程函数 Thread_TCPconnectScan 传来的目标主机 IP 地址和扫描端

口号，创建流套接字 ConSock,设置连接目标主机的套接字地址 HostAddr。

```
//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);
if(ConSock < 0)
{
    pthread_mutex_lock(&TCPConPrintlocker);
    cout<<"Create TCP connect Socket failed! "<<endl;
    pthread_mutex_unlock(&TCPConPrintlocker);
}
//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);
```

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。

```
//connect目标主机
ret = connect(ConSock,(struct sockaddr*)&HostAddr,sizeof(HostAddr));
if(ret==-1)//连接失败，端口关闭
{
    pthread_mutex_lock(&TCPConPrintlocker);
    cout<<"Host: "<<HostIP<<" Port: "<<HostPort<<" closed ! "<<endl;
    pthread_mutex_unlock(&TCPConPrintlocker);
}
else//连接成功，端口打开
{
    pthread_mutex_lock(&TCPConPrintlocker);
    cout<<"Host: "<<HostIP<<" Port: "<<HostPort<<" open ! "<<endl;
    pthread_mutex_unlock(&TCPConPrintlocker);
}
```

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

```
//退出线程
delete p;
close(ConSock);//关闭套接字
// 子线程数减1
pthread_mutex_lock(&TCPConScanlocker);
    TCPConThrdNum--;
pthread_mutex_unlock(&TCPConScanlocker);
```

4. TCP SYN 扫描

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。TCP SYN 扫描通过两个线程函数 ThreadTCP_SynScan 和 Thread_TCPSYNHost 实现。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 函数则用于完成对目标主机指定端口的 TCP

SYN 扫描。

（1） Thread_TCPSYNHost 函数

Thread_TCPSYNHost 函数的流程如下图所示

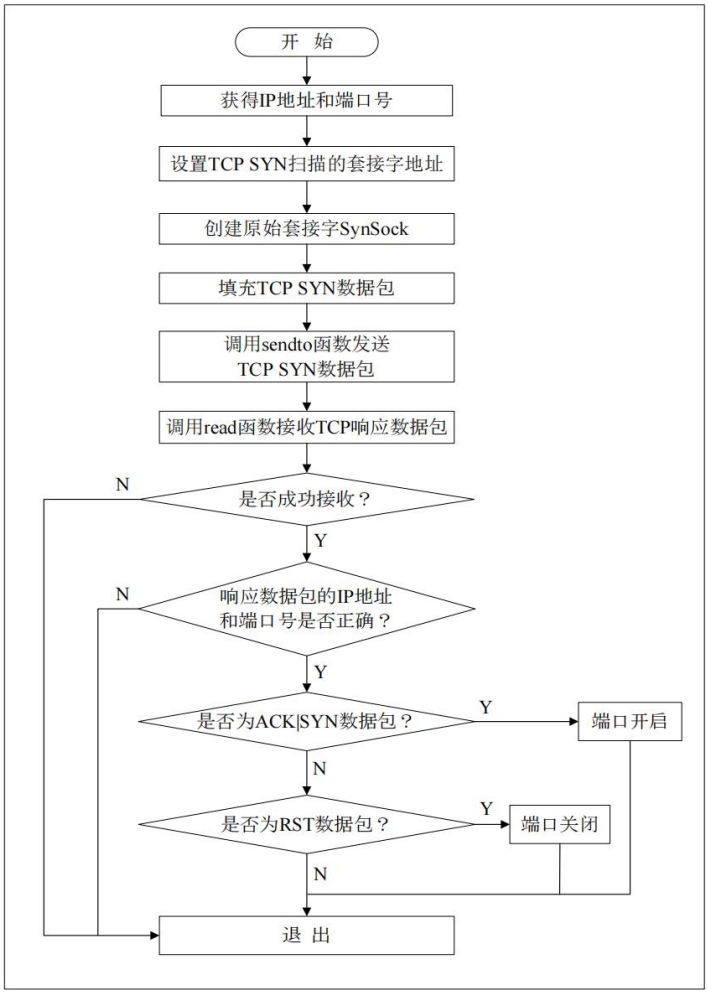


图 8-4 函数 Thread_TCPSYNHost 流程图

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。

a. 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。

```
void* Thread_TCPSYNHost(void* param)
{
    struct TCPSYNHostThrParam *p;
    string HostIP;
    unsigned HostPort,LocalPort,LocalHostIP;
    int SynSock;
    int len;
    char sendbuf[8192];
    char recvbuf[8192];
    struct sockaddr_in SYNScanHostAddr;
    //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;
    HostIP = p->HostIP;
    HostPort = p->HostPort;
    LocalPort = p->LocalPort;
    LocalHostIP = p->LocalHostIP;
```

b. 设置 TCP SYN 扫描的套接字地址。

```
    //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);
```

c. 创建原始套接字 SynSock。

```
    //创建套接字
    SynSock = socket(PF_INET, SOCK_RAW, IPPROTO_TCP);
    // cout << "Create raw socket successfully." << endl;
    if (SynSock < 0)
    {
        pthread_mutex_lock(&TCPSynPrintlocker);
        cout << "Can't create raw socket !" << endl;
        pthread_mutex_unlock(&TCPSynPrintlocker);
    }
```

d. 填充 TCP SYN 数据包。TCP 头 flags 字段的 SYN 位设置为 1。

```
    //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);
```

在填充 SYN 数据包的时候调用了函数 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。

```
//校验和
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);
}
```

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

```
//发送TCP SYN数据包
len=sendto(SynSock, tcph, 20, 0, (struct sockaddr *)&SYNScanHostAddr, sizeof(SYNScanHostAddr));
if( len < 0)
{
    pthread_mutex_lock(&TCPSynPrintlocker);
    cout<<"Send TCP SYN Packet error !"<<endl;
    pthread_mutex_unlock(&TCPSynPrintlocker);
}
```

f. 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。

```
//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
{
    pthread_mutex_lock(&TCPSynPrintlocker);
    cout<<"Read TCP SYN Packet error !"<<endl;
    pthread_mutex_unlock(&TCPSynPrintlocker);
}
```

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。

```
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
    struct ip *iph=(struct ip *)recvbuf;
    int i=iph->ip_hl*4;
    struct tcphdr *tcph=(struct tcphdr *)&recvbuf[i];

    string SrcIP = inet_ntoa(iph->ip_src);        // TCP响应包中的源地址
    string DstIP = inet_ntoa(iph->ip_dst);        //TCP响应包中的目的地址
    struct in_addr in_LocalhostIP;
    in_LocalhostIP.s_addr = LocalHostIP;
    string LocalIP = inet_ntoa(in_LocalhostIP);  // 本机地址

    unsigned SrcPort = ntohs(tcph->th_sport);     // TCP响应包中的源端口号
    unsigned DstPort = ntohs(tcph->th_dport);     // TCP响应包中的目的端口号
    if(HostIP == SrcIP && LocalIP == DstIP && SrcPort == HostPort && DstPort == LocalPort)
    {
        if (tcph->th_flags == TH_SYN || tcph->th_flags == TH_ACK) //判断是否为SYN|ACK数据包
        {
            // 端口开启
            pthread_mutex_lock(&TCPSynPrintlocker);
            cout<<"Host: "<<SrcIP<<" Port: "<<ntohs(tcph->th_sport)<<" closed !"<<endl;
            pthread_mutex_unlock(&TCPSynPrintlocker);
        }
        if (tcph->th_flags == TH_RST) // 判断是否为RST数据包
        {
            // 端口关闭
            pthread_mutex_lock(&TCPSynPrintlocker);
            cout<<"Host: "<<SrcIP<<" Port: "<<ntohs(tcph->th_sport)<<" open !"<<endl;
            pthread_mutex_unlock(&TCPSynPrintlocker);
        }
    }

}
```

g. 关闭套接字 ConSock，子线程数 TCPSynThrdNum 减 1，退出线程

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

pthread_mutex_lock(&TCPSynScanlocker);
TCPSynThrdNum--;
pthread_mutex_unlock(&TCPSynScanlocker);
```

（2） Thread_TCPSynScan 函数

Thread_TCPSynScan 是扫描的主线程函数，该函数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。

使用变量 TCPSynThrdNum 来记录已经创建的子线程数。在函数 Thread_TCPSynScan 中，每创建一个连接指定端口的子线程，就将 TCPSynThrdNum 加 1。为了保证多个不同线程对子线程数 TCPSynThrdNum 的互斥访问，在修改 TCPSynThrdNum 之前需要加互斥锁 TCPSynScanlocker，修改完毕后再解锁以保证 TCPSynThrdNum 的正确性。

```
int TCPSynThrdNum;

//locker
pthread_mutex_t TCPSynPrintlocker=PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t TCPSynScanlocker=PTHREAD_MUTEX_INITIALIZER;
extern unsigned short in_cksum(unsigned short *ptr, int nbytes);
```

在函数中进行变量定义，获取目标主机 IP，起始端口，终止端口的等信息。将记录已创建的子线程数的变量 TCPSynThrdNum 设为 0。

```
void* Thread_TCPSynScan(void* param)
{
    // 变量定义
    struct TCPSYNThrParam *p;
    string HostIP;
    unsigned BeginPort,EndPort,TempPort,LocalPort,LocalHostIP;
    pthread_t listenThreadID,subThreadID;
    pthread_attr_t attr,lattr;
    int ret;
    //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;
    HostIP = p->HostIP;
    BeginPort = p->BeginPort;
    EndPort = p->EndPort;
    LocalHostIP = p->LocalHostIP;
    //循环遍历扫描端口
    TCPSynThrdNum = 0;
    LocalPort = 1024;
```

从起始端口到终止端口循环扫描目标主机的端口。设置子线程参数，将子线程设为分离状态。创建 SYN 目标主机制定的端口子线程。在函数 Thread_TCPSYNHost 退出当前线程之前，将 TCPSynThrdNum 加 1。若子线程数大于 100，线程 Thread_TCPSynScan 就会暂时休眠，等待子线程数降低后再继续工作。

```
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);
    if (ret==-1)
    {
        cout<<"Can't create the TCP SYN Scan Host thread !"<<endl;
    }

    pthread_attr_destroy(&attr);
    pthread_mutex_lock(&TCPSynScanlocker);
    // 子线程数+1
    TCPSynThrdNum++;
    pthread_mutex_unlock(&TCPSynScanlocker);
    // 子线程数大于100休眠
    while (TCPSynThrdNum>100)
    {
        sleep(3);
    }
}
```

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。

```
//等待所有子线程返回
while (TCPSynThrdNum != 0)
{
    sleep(1);
}
cout<<"TCP SYN scan thread exit !"<<endl;
// 返回主流程
pthread_exit(NULL);
```

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。

（1） Thread_TCPFinScan 函数

线程函数 Thread_TCPFinScan 与 Thread_TCPSynScan 函数基本相同，Thread_TCPFinScan 负责遍历端口号，创建 TCP FIN 扫描子线程。

使用变量 TCPFinThrdNum 来记录已经创建的子线程数。在函数 Thread_TCPFINScan 中，每创建一个连接指定端口的子线程，就将 TCPFinThrdNum 加 1。为了保证多个不同线程对子线程数 TCPSynThrdNum 的互斥访问，在修改 TCPFinThrdNum 之前需要加互斥锁 TCPFinScanlocker，修改完毕后再解锁以保证 TCPFinThrdNum 的正确性。

```
int TCPFinThrdNum;

//locker
pthread_mutex_t TCPFinPrintlocker=PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t TCPFinScanlocker=PTHREAD_MUTEX_INITIALIZER;
extern unsigned short in_cksum(unsigned short *ptr, int nbytes);
```

在函数中进行变量定义，获取目标主机 IP，起始端口，终止端口的等信息。将记录已创建的子线程数的变量 TCPSynThrdNum 设为 0。

```
// 变量定义
struct TCPFINThrParam *p;
string HostIP;
unsigned BeginPort,EndPort,TempPort,LocalPort,LocalHostIP;
pthread_t listenThreadID,subThreadID;
pthread_attr_t attr,lattr;
int ret;
//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 TCPFINThrParam*)param;
HostIP = p->HostIP;
BeginPort = p->BeginPort;
EndPort = p->EndPort;
LocalHostIP = p->LocalHostIP;
//循环遍历扫描端口
TCPFinThrdNum = 0;
LocalPort = 1024;
```

从起始端口到终止端口循环扫描目标主机的端口。设置子线程参数，将子线程设为分离状态。创建 FIN 目标主机制定的端口子线程。在函数 Thread_TCPFINHost 退出当前线程之前，将 TCPFinThrdNum 加 1。若子线程数大于 100，线程 Thread_TCPFinScan 就会暂时休眠，等待子线程数降低后再继续工作。

```
for (TempPort=BeginPort;TempPort<=EndPort;TempPort++)
{
    //设置子线程参数
    struct TCPFINHostThrParam *pTCPFINHostParam = new TCPFINHostThrParam;
    pTCPFINHostParam->HostIP = HostIP;
    pTCPFINHostParam->HostPort = TempPort;
    pTCPFINHostParam->LocalPort = TempPort + LocalPort;
    pTCPFINHostParam->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_TCPFINHost,pTCPFINHostParam);
    if (ret==-1)
        cout<<"Can't create the TCP FIN Scan Host thread !"<<endl;
    pthread_attr_destroy(&attr);
    pthread_mutex_lock(&TCPFinScanlocker);
    // 子线程数加1
    TCPFinThrdNum++;
    pthread_mutex_unlock(&TCPFinScanlocker);
    // 子线程数大于100休眠
    while (TCPFinThrdNum>100)
        sleep(3);
}
```

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。

```
// 等待所有子线程返回
while (TCPFinThrdNum != 0)
    sleep(1);
cout<<"TCP FIN scan thread exit !"<<endl;
// 返回主流程
pthread_exit(NULL);
```

（2） Thread_TCPFINHost 函数

线程函数 Thread_TCPFINHost 的流程与线程函数 Thread_TCPSYNHost 基本相同。Thread_TCPFINHost 流程如下

a. 获得线程函数 Thread_TCPFinScan 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。

```
void* Thread_TCPFINHost(void* param)
{
    // 变量声明
    struct TCPFINHostThrParam* p;
    string HostIP, SrcIP, DstIP, LocalIP;
    unsigned HostPort, LocalPort, SrcPort, DstPort, LocalHostIP;
    struct sockaddr_in FINScanHostAddr, FromAddr, FinRevAddr;
    struct in_addr in_LocalhostIP;
    int FinSock, FinRevSock;
    int len, FromAddrLen;
    char sendbuf[8192];
    char recvbuf[8192];
    struct timeval TpStart, TpEnd;
    float TimeUse;
    //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 TCPFINHostThrParam*)param;
    HostIP = p->HostIP;
    HostPort = p->HostPort;
    LocalPort = p->LocalPort;
    LocalHostIP = p->LocalHostIP;
```

b. 设置 TCP FIN 扫描的套接字地址。

```
    //设置TCP SYN扫描的套接字地址
    memset(&FINScanHostAddr, 0, sizeof(FINScanHostAddr));
    FINScanHostAddr.sin_family = AF_INET;
    FINScanHostAddr.sin_addr.s_addr = inet_addr(&HostIP[0]);
    FINScanHostAddr.sin_port = htons(HostPort);
```

c. 创建原始套接字 FinSock、FinRevSock。

```
    //创建套接字
    FinSock = socket(PF_INET, SOCK_RAW, IPPROTO_TCP);
    if (FinSock < 0)
    {
        pthread_mutex_lock(&TCPFinPrintlocker);
        cout << "Can't creat raw socket !" << endl;
        pthread_mutex_unlock(&TCPFinPrintlocker);
    }
    FinRevSock = socket(PF_INET, SOCK_RAW, IPPROTO_TCP);
    if (FinRevSock < 0)
    {
        pthread_mutex_lock(&TCPFinPrintlocker);
        cout << "Can't creat raw socket !" << endl;
        pthread_mutex_unlock(&TCPFinPrintlocker);
    }
```

d. 填充 TCP FIN 数据包。TCP 头 flags 字段的 FIN 位设置为 1。

```
//填充TCP FIN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_FIN; //TCP头flags字段的FIN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);
```

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

```
//发送TCP FIN数据包
len = sendto(FinSock, tcph, 20, 0, (struct sockaddr*)&FINScanHostAddr, sizeof(FINScanHostAddr));
if (len < 0)
{
    pthread_mutex_lock(&TCPFinPrintlocker);
    cout << "Send TCP FIN Packet error !" << endl;
    pthread_mutex_unlock(&TCPFinPrintlocker);
}
```

f. 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。

```
// 将套接字设置为非阻塞模式
if (fcntl(FinRevSock, F_SETFL, O_NONBLOCK) == -1)
{
    pthread_mutex_lock(&TCPFinPrintlocker);
    cout << "Set socket in non-blocked model fail !" << endl;
    pthread_mutex_unlock(&TCPFinPrintlocker);
}
```

g. do while 循环接收 TCP 响应数据包。调用 recvfrom 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。

```
//接收 TCP 响应数据包循环
FromAddrLen = sizeof(struct sockaddr_in);
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);
```

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。

```
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);
        //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 (SrcIP == HostIP)// 响应数据包的源地址等于目标主机地址
        {
            struct ip* iph = (struct ip*)recvbuf;
            int i = iph->ip_hl * 4;
            struct tcphdr* tcph = (struct tcphdr*)&recvbuf[i];
            SrcIP = inet_ntoa(iph->ip_src);        // TCP响应数据包的源IP地址
            DstIP = inet_ntoa(iph->ip_dst);        // TCP响应数据包的目的IP地址
            in_LocalhostIP.s_addr = LocalHostIP;
            LocalIP = inet_ntoa(in_LocalhostIP);  // 本机IP地址
            unsigned SrcPort = ntohs(tcph->th_sport);     // TCP响应包的源端口号
            unsigned DstPort = ntohs(tcph->th_dport);     // TCP响应包的目的端口号
            if (HostIP == SrcIP && LocalIP == DstIP && SrcPort == HostPort && DstPort == LocalPort)
            {
                if (tcph->th_flags == 0x14) //判断是否为 RST 数据包
                {
                    pthread_mutex_lock(&TCPFinPrintlocker);
                    cout << "Host: " << SrcIP << " Port: " << ntohs(tcph->th_sport) << " closed !" << endl;
                    pthread_mutex_unlock(&TCPFinPrintlocker);
                }
            }
            break;
        }
    }
```

如果等待时间超过 3 秒，那么就认为被扫描端口是开启的。

```
    // 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
    {
        pthread_mutex_lock(&TCPFinPrintlocker);
        cout << "Host: " << HostIP << " Port: " << HostPort << " open !" << endl;
        pthread_mutex_unlock(&TCPFinPrintlocker);
        break;
    }
} while (true);
```

h. 关闭套接字 ConSock，子线程数 TCPSynThrdNum 减 1，退出线程

```
    //退出子线程
    delete p;
    close(FinSock);
    close(FinRevSock);
    pthread_mutex_lock(&TCPFinScanlocker);
    // 子线程数减1
    TCPFinThrdNum--;
    pthread_mutex_unlock(&TCPFinScanlocker);
}
```

6. UDP 扫描

UDP 扫描是通过线程函数 Thread_UDPScan 和普通函数 UDPScanHost 实现的。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。

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。

（1） Thread_UDPScan 函数

Thread_UDPScan 函数在从启始端口（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)
{
    struct UDPThrParam* p;
    string HostIP;
    unsigned BeginPort, EndPort, TempPort, LocalPort, LocalHostIP;
    pthread_t subThreadID;
    pthread_attr_t attr;
    int ret;

    p = (struct UDPThrParam*)param;
    HostIP = p->HostIP;
    BeginPort = p->BeginPort;
    EndPort = p->EndPort;
    LocalHostIP = p->LocalHostIP;
    //遍历端口，逐次扫描
    LocalPort = 1024;

    for (TempPort = BeginPort; TempPort <= EndPort; TempPort++)
    {
        UDPScanHostThrParam* pUDPScanHostParam = new UDPScanHostThrParam;
        pUDPScanHostParam->HostIP = HostIP;
        pUDPScanHostParam->HostPort = TempPort;
        pUDPScanHostParam->LocalPort = TempPort + LocalPort;
        pUDPScanHostParam->LocalHostIP = LocalHostIP;
        UDPScanHost(pUDPScanHostParam);
    }
    cout << "UDP Scan thread exit !" << endl;
    pthread_exit(NULL);
}
```

（2） UDPScanHost 函数

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。

UDPScanHost 函数的流程如下图所示。

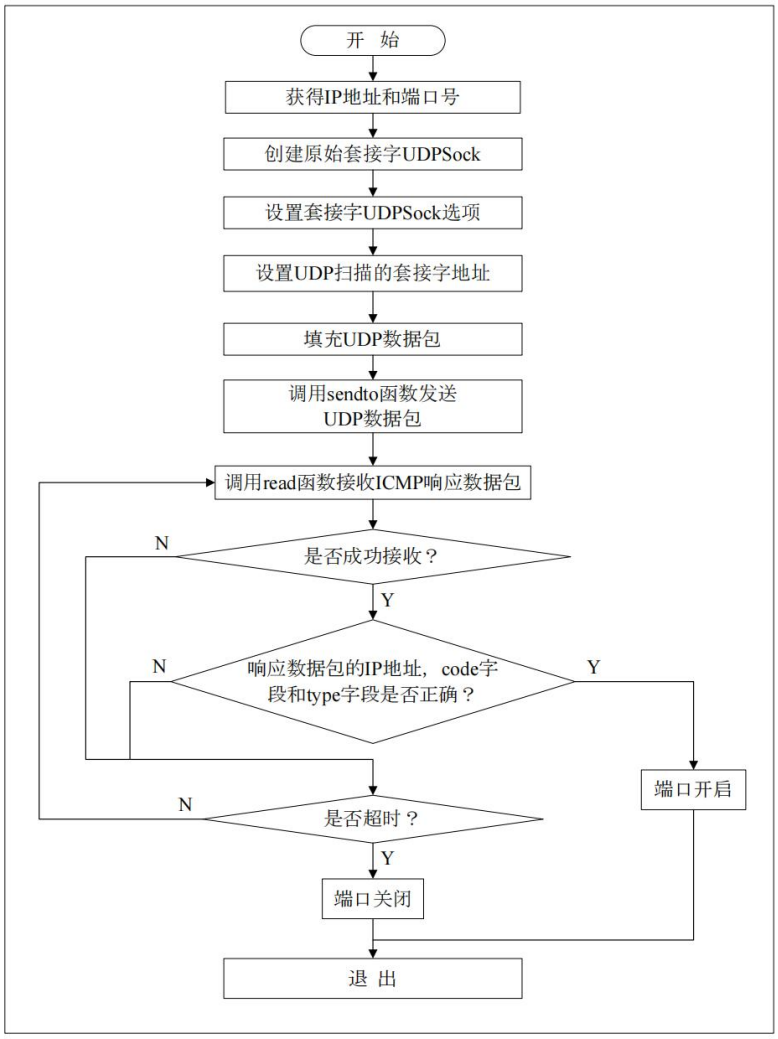


图 8-5 函数 UDPScanHost 流程图

UDPScanHost 函数的流程如下：

a. 获取目标主机 IP 地址和扫描端口号，以及本机 IP 地址和端口号

```
void UDPScanHost(struct UDPScanHostThrParam* p)
{
    // 变量定义
    string HostIP;
    unsigned HostPort, LocalPort, LocalHostIP;

    int UDPSock;
    struct sockaddr_in UDPScanHostAddr, FromHostAddr;
    int n, FromLen;
    int on, ret;

    struct icmp* icmp;
    struct ipicmphdr hdr;

    struct iphdr* ip;
    struct udphdr* udp;
    struct pseudohdr* pseudo;
    char packet[sizeof(struct iphdr) + sizeof(struct udphdr)];

    char SendBuf[2];
    char RecvBuf[100];

    int HeadLen;
    struct timeval TimeOut;
    struct timeval TpStart, TpEnd;
    float TimeUse;

    // 获得目标主机IP地址
    HostIP = p->HostIP;
    HostPort = p->HostPort;
    LocalPort = p->LocalPort;
    LocalHostIP = p->LocalHostIP;
```

b. 创建原始套接字 UDPSock

```
    // 创建套接字UDPSock
    UDPSock = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
    if (UDPSock < 0)
    {
        pthread_mutex_lock(&UDPPrintlocker);
        cout << "Can't creat raw icmp socket !" << endl;
        pthread_mutex_unlock(&UDPPrintlocker);
    }
```

c. 设置套接字 UDPSock 的选项

```
    // 设置套接字UDPSock选项
    on = 1;
    ret = setsockopt(UDPSock, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on));
    if (ret < 0)
    {
        pthread_mutex_lock(&UDPPrintlocker);
        cout << "Can't set raw socket !" << endl;
        pthread_mutex_unlock(&UDPPrintlocker);
    }
```

d. 设置 UDP 扫描的套接字地址

```
// 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);
```

e. 填充 UDP 数据包。UDP 头的校验和字段调用函数 in_cksum 进行计算

```
// 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->saddr = inet_addr("192.168.175.132");
ip->daddr = inet_addr(&HostIP[0]);
```

f. 调用 sendto 函数向目标主机的指定端口发送 UDP 数据包

```
// 发送UDP数据包
n = sendto(UDPSock, packet, ip->tot_len, 0, (struct sockaddr*)&UDPScanHostAddr, sizeof(UDPScanHostAddr));
if (n < 0)
{
    pthread_mutex_lock(&UDPPrintlocker);
    cout << "Send message to Host Failed !" << endl;
    pthread_mutex_unlock(&UDPPrintlocker);
}
```

g. 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。

```
// 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*/
            pthread_mutex_lock(&UDPPrintlocker);
            cout << "Host: " << HostIP << " Port: " << HostPort << " closed !" << endl;
            pthread_mutex_unlock(&UDPPrintlocker);
            break;
        }
    }
```

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。

```
    //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*/
        pthread_mutex_lock(&UDPPrintlocker);
        cout << "Host: " << HostIP << " Port: " << HostPort << " open !" << endl;
        pthread_mutex_unlock(&UDPPrintlocker);
        break;
    }
} while (true);
```

h. 关闭套接字 UDPSock，返回

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

7. 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。

打开(O) Makefile /mnt/hgfs/ubuntu-share/Network Security/Lab4_PortScanner 保存(S)

```
SRC=$(wildcard ./src/*.cpp)
OBJ=$(patsubst ./src/%.cpp, ./obj/%.o, $(SRC))
INCLUDE=./include/
CC=g++
TARGET=Scanner
CXXFLAGS=-w -lpthread -I
ALL:$(TARGET)

$(OBJ):./obj/%.o:./src/%.cpp
	$(CC) -c $< -o $@ $(CXXFLAGS) $(INCLUDE)

$(TARGET):$(OBJ)
	$(CC) $(OBJ) -lpthread -o $(TARGET)

clean:
	rm -rf $(TARGET) $(OBJ)

run:
	./$(TARGET)
.PHONY:clean ALL run
```

Makefile 制表符宽度：8 第 8 行，第 1 列 插入

五、 实验结果

1. 生成目标文件

执行命令如下

```
judy@judy-virtual-machine:/mnt/hgfs/ubuntu-share/Network Security/Lab4_PortScanner$ make
g++ -c src/Scanner.cpp -o obj/Scanner.o -w -lpthread -I ./include/
g++ -c src/TCPConnectScan.cpp -o obj/TCPConnectScan.o -w -lpthread -I ./include/
g++ -c src/TCPFINScan.cpp -o obj/TCPFINScan.o -w -lpthread -I ./include/
g++ -c src/TCPSYNScan.cpp -o obj/TCPSYNScan.o -w -lpthread -I ./include/
g++ -c src/UDPScan.cpp -o obj/UDPScan.o -w -lpthread -I ./include/
```

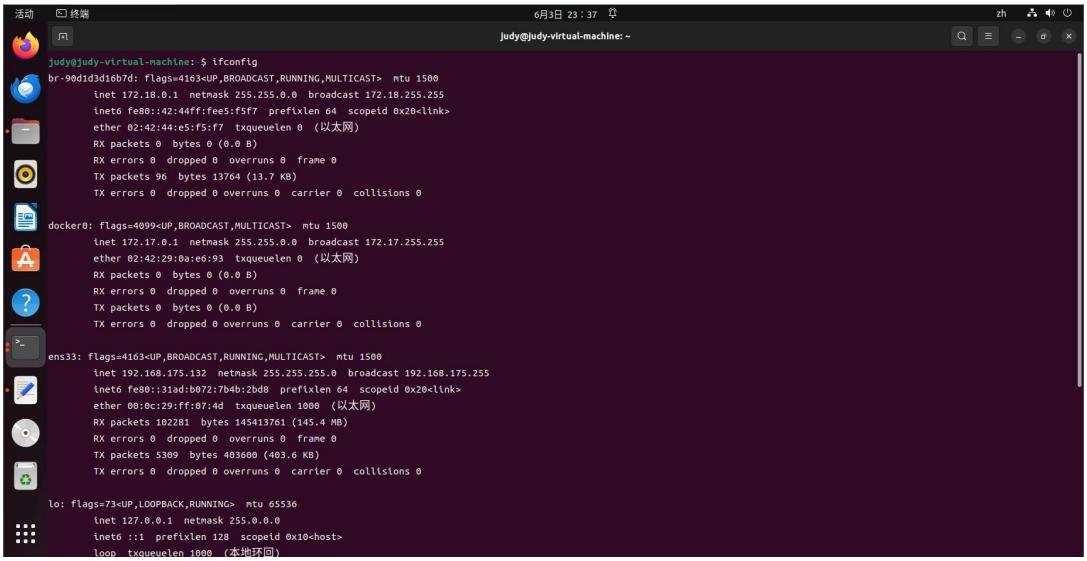
2. 输出帮助信息

`./Scanner -h`输出帮助信息，结果如下

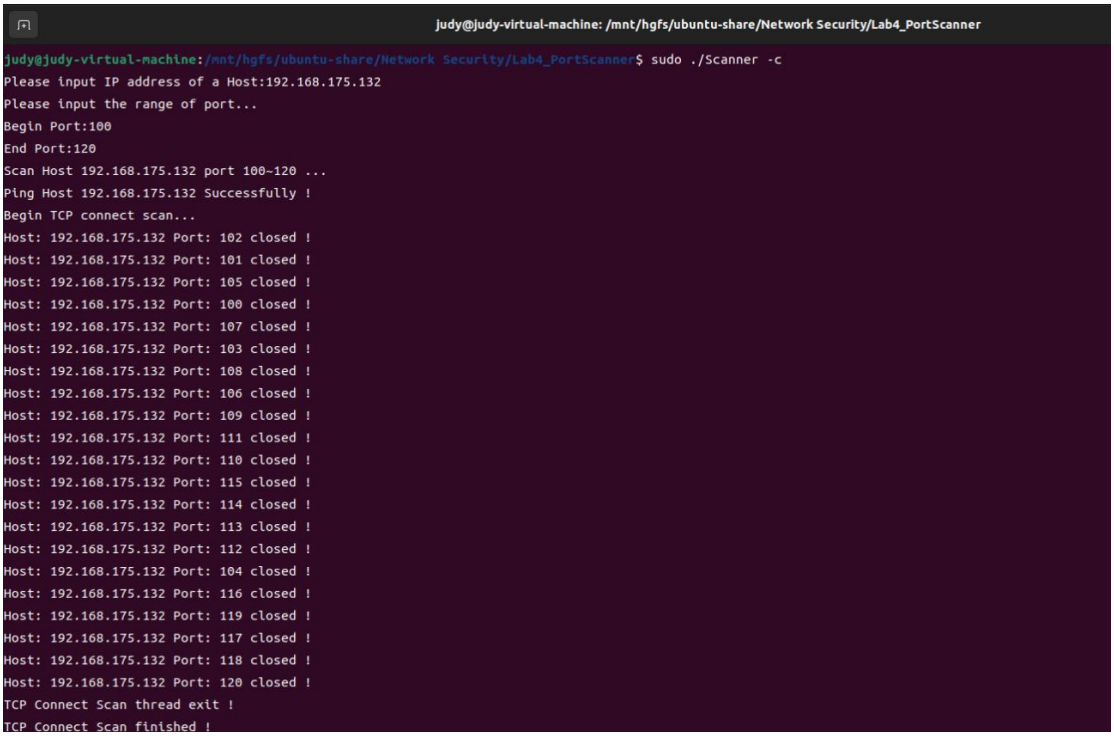
```
judy@judy-virtual-machine:/mnt/hgfs/ubuntu-share/Network Security/Lab4_PortScanner$ ./Scanner -h
Scanner: usage: [-h]  --help information
                [-c]  --TCP connect scan
                [-s]  --TCP syn scan
                [-f]  --TCP fin scan
                [-u]  --UDP scan
```

3. TCP connect 扫描

终端输入 ifconfig 查看本机 IP 地址，可知 ens33 接口的 IP 地址是 192.168.175.132。



使用 sudo ./Scanner -c 命令进行 TCP connect 扫描，输入主机的 IP 地址，以及 TCP connect 扫描的起始和终止端口号。结果如下



4. TCP SYN 扫描

使用 sudo ./Scanner -s 命令进行 TCP SYN 扫描，输入主机的 IP 地址，以及 TCP SYN 扫描的起始和终止端口号。结果如下

```
judy@judy-virtual-machine:/mnt/hgfs/ubuntu-share/Network Security/Lab4_PortScanner$ sudo ./Scanner -s
Please input IP address of a Host:192.168.175.132
Please input the range of port...
Begin Port:56
End Port:66
Scan Host 192.168.175.132 port 56~66 ...
Ping Host 192.168.175.132 Successfully !
Begin TCP SYN scan...
Host: 192.168.175.132 Port: 56 closed !
Host: 192.168.175.132 Port: 57 closed !
Host: 192.168.175.132 Port: 59 closed !
Host: 192.168.175.132 Port: 60 closed !
Host: 192.168.175.132 Port: 61 closed !
Host: 192.168.175.132 Port: 63 closed !
Host: 192.168.175.132 Port: 65 closed !
Host: 192.168.175.132 Port: 66 closed !
TCP SYN scan thread exit !
TCP SYN Scan finished !
```

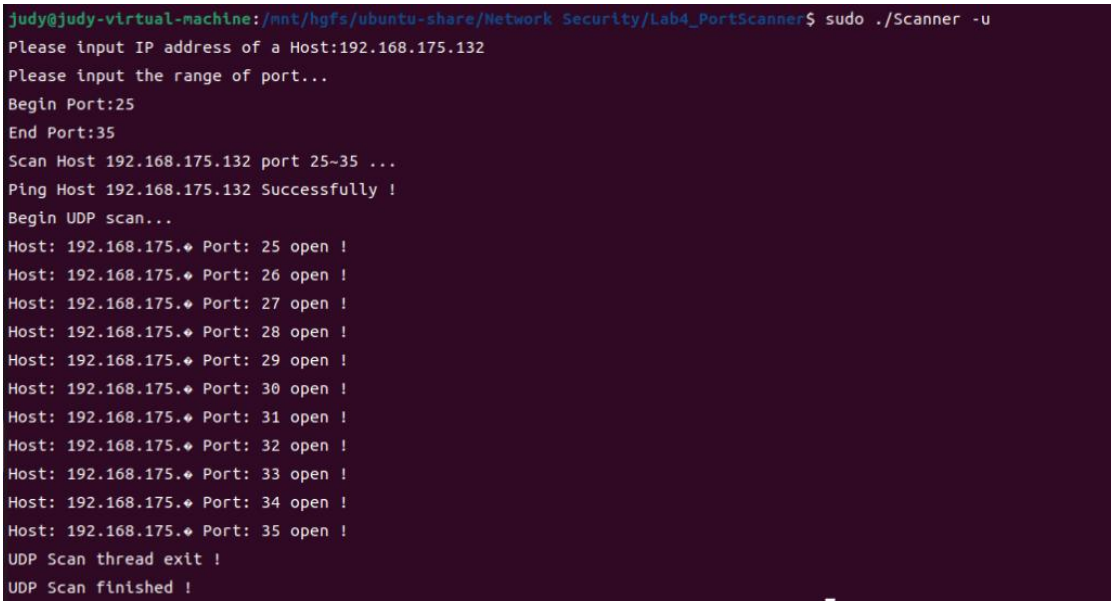
5. TCP FIN 扫描

使用 sudo ./Scanner -f 命令进行 TCP FIN 扫描，输入主机的 IP 地址，以及 TCP FIN 扫描的起始和终止端口号。结果如下

```
judy@judy-virtual-machine:/mnt/hgfs/ubuntu-share/Network Security/Lab4_PortScanner$ sudo ./Scanner -f
Please input IP address of a Host:192.168.175.132
Please input the range of port...
Begin Port:89
End Port:99
Scan Host 192.168.175.132 port 89~99 ...
Ping Host 192.168.175.132 Successfully !
Begin TCP FIN scan...
Host: 192.168.175.132 Port: 89 closed !
Host: 192.168.175.132 Port: 90 closed !
Host: 192.168.175.132 Port: 91 closed !
Host: 192.168.175.132 Port: 92 closed !
Host: 192.168.175.132 Port: 94 closed !
Host: 192.168.175.132 Port: 95 closed !
Host: 192.168.175.132 Port: 96 closed !
Host: 192.168.175.132 Port: 97 closed !
Host: 192.168.175.132 Port: 98 closed !
Host: 192.168.175.132 Port: 99 closed !
TCP FIN scan thread exit !
TCP FIN Scan finished !
```

6. UDP 扫描

使用 sudo ./Scanner -u 命令进行 UDP 扫描，输入主机的 IP 地址，以及 UDP 扫描的起始和终止端口号。结果如下



六、 实验遇到的问题及其解决方法

1. 在实现 Ping 程序、TCP SYN 扫描和 TCP FIN 扫描时，创建原始套接字(SOCK_RAW)可能会失败，导致无法发送或接收数据包。确保运行程序的用户具有管理员权限，因为创建原始套接字需要高权限。另外，可以在 Linux 系统上使用 sudo 命令运行程序。

2. 在 TCP connect 扫描和 TCP SYN 扫描中，多个线程同时操作共享变量（如线程计数器），可能导致竞态条件和数据不一致问题。使用互斥锁（mutex）保护共享变量的访问。在每次修改共享变量前，加锁以保证线程安全，修改完毕后解锁。

3. Ping 程序发送 ICMP 请求后无法接收到响应，可能是由于防火墙或路由器设置阻止 ICMP 响应。检查并配置本机和目标主机的防火墙设置，确保 ICMP 数据包不被阻止。可以暂时关闭防火墙进行测试。

通过此次实验，我不仅掌握了多种端口扫描技术，还积累了解决实际编程问题的经验。这些问题的解决过程，使我更加深入理解了网络编程和系统设计中的细节和挑战。

七、 实验结论

通过此次实验，我深入了解并实际实现了多种端口扫描技术，包括 Ping 扫描、TCP connect 扫描、TCP SYN 扫描、TCP FIN 扫描以及 UDP 扫描。整个实验过程中，既有理论知识的学习，也有实际代码的编写和调试，使我对网络扫描技

术有了更加全面和深入的理解。

首先，通过 Ping 程序的实现理解了 ICMP 协议的工作原理。Ping 扫描通过发送 ICMP 请求包并解析响应包，可以快速有效地判断目标主机的可达性。Ping 扫描是端口扫描器中一个重要的前置步骤，能在进行详细端口扫描前筛选出不可达的主机，节省扫描时间通过创建和发送 ICMP 数据包，以及接收和解析响应包，我练习了如何利用 Ping 程序来检测目标主机的可达性。。

在 TCP connect 扫描部分，通过创建子线程并发扫描多个端口，大大提高了扫描效率。TCP connect 扫描利用流式套接字的 connect 函数，能够直接判断目标端口的开放状态。然而，如何保证多线程环境下的线程安全和资源管理，也让我在编程过程中花费了不少时间和精力。

TCP SYN 扫描和 TCP FIN 扫描的实现，进一步加深了对套接字的理解。TCP SYN 扫描通过原始套接字手动构造 SYN 包，并解析返回的 TCP 响应包，能隐蔽地探测端口状态。该方法仅发送 SYN 包，不建立完整的 TCP 连接，因此比 TCP connect 扫描更加隐蔽。TCP FIN 扫描与 TCP SYN 扫描类似，利用原始套接字构造 FIN 包并解析响应包。不同的是，FIN 扫描适用于探测开放的端口在不合法请求时的反应。这两种扫描方式都需要手动构造 TCP 数据包，并通过分析返回的 TCP 响应包来判断端口状态。特别是 TCP SYN 扫描，由于其隐蔽性，在实际应用中具有重要意义。

UDP 扫描部分的实现虽然没有采用多线程，但由于 UDP 协议的无连接性和 ICMP 响应包的不可达性，使得这部分实验同样具有挑战性。通过逐个端口发送 UDP 数据包，并分析 ICMP 响应包，成功实现了对目标主机 UDP 端口的扫描。

本次实验不仅让我掌握了多种端口扫描技术及其实现方法，还让我深刻体会到在实际编程过程中，需要综合考虑效率、安全性和资源管理等多方面的因素。这些经验和教训对我未来的网络编程和系统设计将有很大的帮助。