网 络 安 全 技 术

实 验 报 告

学　　院：网络空间安全学院
年　　级：2021 级
专　　业：信息安全、法学双学位
学　　号：2113203
姓　　名：付政烨
手 机 号：15719340667
完成日期：2024 年 6 月 3 日

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。
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

二、 实验内容

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。

三、 实验步骤

（一） 主要数据结构

代码定义了一些用于网络扫描和测试的结构体，这些结构体包含了网络通信中常见的参数和协议头部信息，详见下表:

结构体	功能	成员变量
TCPConThrParam	用于存储 TCP 连接线程的参数	HostIP, BeginPort, EndPort
TCPConHostThrParam	用于存储特定主机的 TCP 连接线程参数	HostIP, HostPort
UDPThrParam	用于存储 UDP 扫描线程的参数	HostIP, BeginPort, EndPort, LocalHostIP
UDPScanHostThrParam	用于存储特定主机的 UDP 扫描线程参数	HostIP, HostPort, LocalPort, LocalHostIP
TCPSYNThrParam	用于存储 TCP SYN 扫描线程的参数	HostIP, BeginPort, EndPort, LocalHostIP
TCPSYNHostThrParam	用于存储特定主机的 TCP SYN 扫描线程参数	HostIP, HostPort, LocalPort, LocalHostIP
TCPFINThrParam	用于存储 TCP FIN 扫描线程的参数	HostIP, BeginPort, EndPort, LocalHostIP
TCPFINHostThrParam	用于存储特定主机的 TCP FIN 扫描线程参数	HostIP, HostPort, LocalPort, LocalHostIP
ipicmphdr	用于存储 IP 和 ICMP 头部信息	iphdr ip, icmphdr icmp
pseudohdr	用于创建校验和	saddr, daddr, useless, protocol, length

图 3.1 数据结构介绍

（二） 辅助函数

这部分代码主要实现了网络工具中的几个关键功能，包括计算校验和、端口验证、获取本机 IP 地址以及通过 ICMP 协议实现 Ping 功能，每个功能分别对应一个函数。

1. 校验和计算

校验和的计算是网络通信中确保数据完整性的一种方法。在代码中，in_cksum 函数用于计算校验和，主要应用于 IP 头部和 ICMP 消息等协议的数据验证。

（1）函数定义

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

- ptr 是指向待计算校验和数据的指针。
- nbytes 是数据的总字节数。

（2）校验和计算流程

这段代码循环处理所有的 16 位字。每次循环，将当前指针 ptr 所指向的 16 位字加到 sum 中，并将 ptr 向前移动到下一个 16 位字，同时 nbytes 减少 2（因为处理了 2 个字节）。

累加 16 位字

```
while(nbytes > 1) {
    sum += *ptr++;
    nbytes -= 2;
}
```

如果数据的总字节数是奇数，那么最后将剩下一个字节未处理。此时，代码将一个额外的 16 位变量 oddbyte 初始化为 0，然后将这个单字节赋值到 oddbyte 的低字节部分，最后加到 sum 中。

处理剩余的单字节（如果有）

```
if (nbytes == 1) {
    oddbyte = 0;
    *((u_char *) &oddbyte) = *(u_char *)ptr;
    sum += oddbyte;
}
```

接下来的代码首先把 sum 的高 16 位和低 16 位相加。由于前面的累加可能导致溢出，这一步确保将所有的溢出位也计入最终的结果。然后再次折叠，确保结果适合 16 位。最后，将 sum 取反得到校验和，然后返回。这里取反是因为在使用这个校验和时，接收方会将所有的 16 位字（包括校验和）再次相加，正确的情况下结果应该为全 1（即 16 位全是 1，二进制表示为 65535 或十六进制的 0xFFFF）。

调整并得到最终的校验和）

```
sum = (sum >> 16) + (sum & 0xffff);
sum += (sum >> 16);
answer = ~sum;
return(answer);
```

2. 端口正确性验证

端口正确性验证是网络编程中常见的一个环节，用来确保指定的端口号是有效的并符合预期的范围。代码中的 IsPortOK 函数用于验证给定的端口号范围是否合理。

```
bool IsPortOK(unsigned bPort,unsigned ePort) {
    if(bPort>ePort)
        return false;
    else {
        if(bPort <1|| bPort>65535||ePort <1|| ePort>65535)
            return false;
        else
            return true;
    }
}
```

1. **检查起始端口是否大于结束端口**：如果起始端口号大于结束端口号，函数立即返回 false。在实际应用中，端口的范围应该是从一个较小的数到一个较大的数，这个检查帮助确保用户输入的端口范围是合逻辑的。

2. **检查端口号是否在有效范围内**：这部分代码验证端口号是否都在有效的范围内。有效的 TCP/UDP 端口号是从 1 到 65535。此处检查两个端口号，确保它们都不低于 1 且不高于 65535。
3. **返回真值**：如果端口号符合所有的条件，函数最终返回 true，表示端口号有效。

3. 获取本机 IP 地址

代码通过运行 UNIX/Linux 系统的 ifconfig 命令，通过一系列的文本处理命令来筛选并提取 IP 地址信息。首先，它运行 ifconfig 命令以获取所有网络接口的详细状态。接下来，使用 grep 命令过滤出包含“inet”字样的行，这些行包含了 IP 地址信息。为了排除本地回环地址（如 127.0.0.1），代码使用 grep -v 127 命令去除这些行。然后，通过 awk 和 cut 命令进一步处理，以从格式化的输出中提取出纯净的 IP 地址。最终，使用 fscanf 读取这个地址，并通过 inet_addr 转换为适合程序使用的格式。此方法依赖于系统的命令行工具和特定的命令格式，因此仅适用于 Unix/Linux 系统，且要求程序具有执行系统命令的权限。

```
unsigned int GetLocalHostIP(void) {
    FILE *fd;
    char buf[20] = {0x00};
    fd = popen("/sbin/ifconfig␣|␣grep␣inet␣|␣grep␣-v␣127␣|␣awk␣'{print␣$2}'␣|
        ␣cut␣-d␣\":\"␣-f␣2", "r");
    if(fd == NULL) {
        fprintf(stderr, "cannot␣get␣source␣ip␣->␣use␣the␣-f␣option\n");
        exit(-1);
    }
    fscanf(fd,"%20s",buf);
    return(inet_addr(buf));
}
```

4. Ping 功能实现

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[1]。Ping 函数实现流程详见下图 3.2

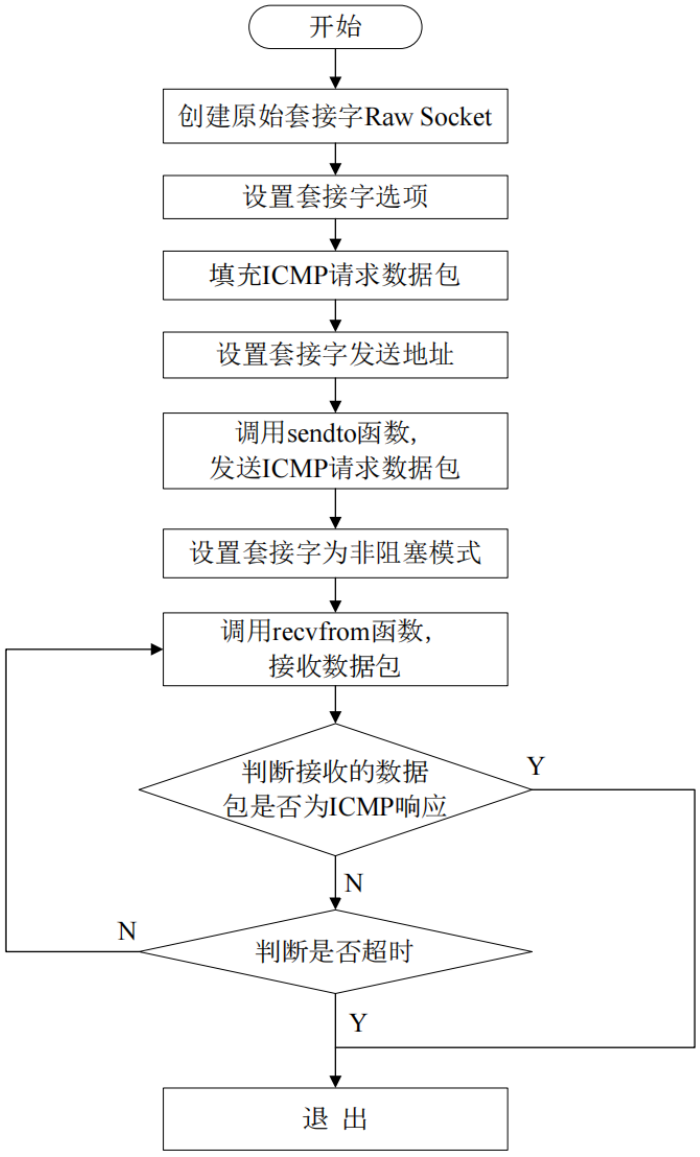

图 3.2 Ping 函数流程图

下面是详细介绍此功能实现的各个步骤和关键代码：

（1）设置套接字选项

创建原始套接字用于发送和接收 ICMP 数据包。原始套接字允许直接对较低层协议如 ICMP 进行访问，这是实现 Ping 功能的基础。

```
PingSock = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
on = 1;
ret = setsockopt(PingSock, 0, IP_HDRINCL, &on, sizeof(on));
```

设置套接字选项 IP_HDRINCL，告诉操作系统在发送数据包时包含 IP 头。因为使用原始套接字，用户需要自行构造 IP 头。

（2）构造 ICMP 请求数据包

- **IP 头部填充**：这部分代码初始化 IP 头部。其中，ihl 和 version 分别设置 IP 头长度和版本号。tot_len 是总长度，ttl 是生存时间，protocol 设置为 ICMP。

```
ip = (struct iphdr*)SendBuf;
ip->ihl = 5;
ip->version = 4;
ip->tos = 0;
...
```

- **ICMP 头部填充**：初始化 ICMP 头部。设置 ICMP 消息类型为 Echo Request（类型 8），并附加一个时间戳用于计算往返时间。

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

- **发送 ICMP 请求**：通过 sendto 函数发送构造好的 ICMP 数据包。如果发送失败，返回错误信息。

```
ret = sendto(PingSock, SendBuf, SendBufSize, 0, (struct sockaddr*)&PingHostAddr, sizeof
    (PingHostAddr));
```

- **接收 ICMP 响应**：代码尝试从套接字接收数据。由于套接字被设置为非阻塞模式，recvfrom 将立即返回，不会等待数据到达。

```
ret = recvfrom(PingSock, RecvBuf, 1024, 0, (struct sockaddr*)&FromAddr, (socklen_t*)&
    Addrlen);
```

- **处理 ICMP 响应**：这部分代码检查接收到的数据包是否来自指定的目标 IP 地址，并且 ICMP 消息类型为 Echo Reply。如果条件满足，则认为 Ping 操作成功。这个循环持续接收数据，直到收到期望的回复或超时。超时时间通常设置为几秒钟，确保不会无限等待响应。

```
if (SrcIP == HostIP && DstIP == LocalIP && Recvicmp->icmp_type ==
    ICMP_ECHOREPLY) {
    flags = true;
    break;
}
```

（三） TCP 连接扫描

TCP 连接扫用于检测指定主机上一系列端口的状态（开放或关闭）。它利用多线程技术来加速扫描过程，每个线程负责检测一个端口。整个扫描过程分为两个主要的线程函数：Thread_TCPconnectHost 和 Thread_TCPconnectScan。

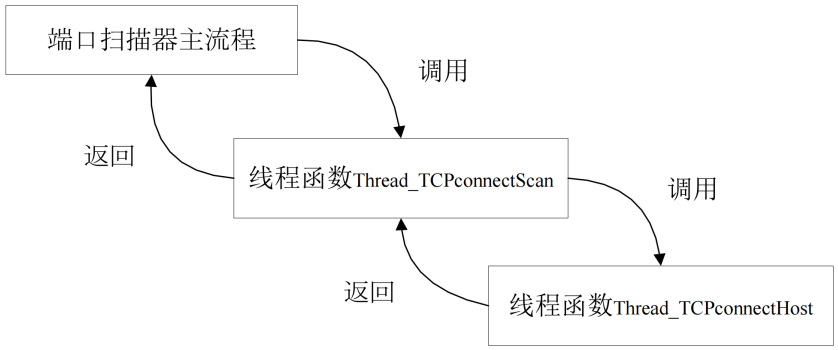

图 3.3 Thread_TCPconnectScan 和 Thread_TCPconnectHost 之间的关系

（1）Thread_TCPconnectHost 函数

此函数是每个端口扫描线程的入口点。它接收一个结构体参数，该结构体包含目标主机的 IP 地址和端口号。

1. **创建套接字**：使用 socket() 函数创建一个 TCP 流套接字。
2. **设置连接地址**：配置目标服务器的地址和端口。
3. **连接到主机**：尝试连接到指定的 IP 和端口。如果连接失败（即端口关闭），会输出相应的消息；如果成功，则表明端口开放。
4. **资源清理**：关闭套接字并删除用于线程参数的内存。

```
ConSock = socket(AF_INET, SOCK_STREAM, 0);
 ...
HostAddr.sin_family = AF_INET;
HostAddr.sin_addr.s_addr = inet_addr(&HostIP[0]);
HostAddr.sin_port = htons(HostPort);
 ...
ret = connect(ConSock, (struct sockaddr*)&HostAddr, sizeof(HostAddr));
```

（2）Thread_TCPconnectScan 函数

Thread_TCPconnectScan 函数是主扫描线程，负责生成子线程来并发扫描多个端口。

1. **循环遍历端口**：从起始端口到终止端口，为每个端口生成一个 Thread_TCPconnectHost 线程。
2. **线程创建和管理**：使用 pthread_create 创建线程，设置线程为分离状态，确保资源自动回收。
3. **线程数量控制**：使用互斥锁和条件变量控制同时活跃的线程数，以避免资源耗尽。
4. **等待所有线程完成**：在所有端口扫描完毕后，主线程等待所有子线程结束。

```
for (TempPort = BeginPort; TempPort <= EndPort; TempPort++) {
    ..
     pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_DETACHED);
```

```
        pthread_create (&subThreadID, &attr，Thread_TCPconnectHost, pConHostParam);
        ..
        while (TCPConThrdNum > 100) {
            sleep (3) ;
        }
}
while (TCPConThrdNum != 0) {
        sleep (1) ;
}
```

（四） TCP SYN 扫描

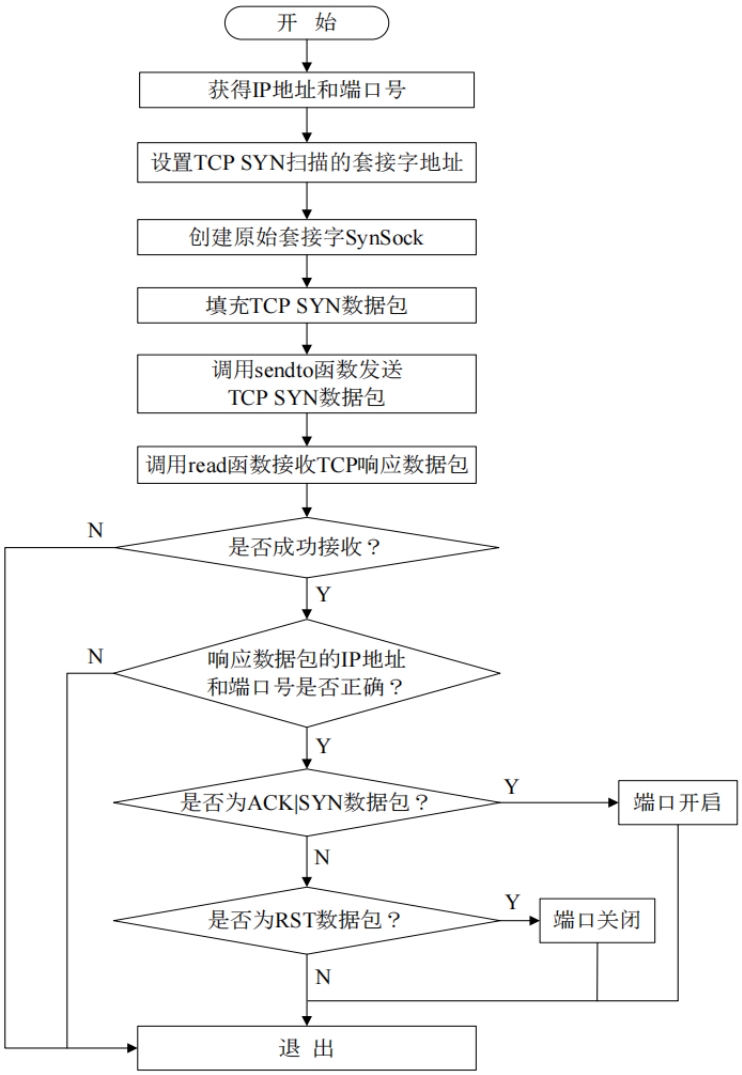

图 3.4 函数 Thread_TCPSYNHost 流程图

TCP SYN 扫描用于检测指定主机上一系列端口的状态（开放、关闭或过滤）。该方法使用原始套接字发送 SYN 包，接收 SYN-ACK 或 RST 响应来判断端口状态。整个过程通过多线程技术并发进行，以加速扫描。

（1）Thread_TCPSYNHost 函数

Thread_TCPSYNHost 函数是每个端口扫描任务的主要执行函数，它负责创建和发送一个 TCP SYN 数据包，并接收响应以判断端口的状态。

1. **参数获取和初始化**：首先，函数从传入的参数中解析出目标主机的 IP 地址、端口号，以及本地主机的 IP 地址和端口号。配置目标服务器的 IP 和端口，这些信息将用于 TCP 数据包的发送。

```
p = (struct TCPSYNHostThrParam*)param;
HostIP = p->HostIP;
HostPort = p->HostPort;
LocalPort = p->LocalPort;
LocalHostIP = p->LocalHostIP;
memset(&SYNScanHostAddr, 0, sizeof(SYNScanHostAddr));
SYNScanHostAddr.sin_family = AF_INET;
SYNScanHostAddr.sin_addr.s_addr = inet_addr(&HostIP[0]);
SYNScanHostAddr.sin_port = htons(HostPort);
```

2. **创建套接字**：使用 socket 函数创建一个原始套接字。原始套接字允许用户应用程序直接访问底层协议，如 TCP，用于手动构造包。

```
SynSock = socket(PF_INET, SOCK_RAW, IPPROTO_TCP);
if (SynSock < 0) {
  pthread_mutex_lock(&TCPSynPrintlocker);
  pthread_mutex_unlock(&TCPSynPrintlocker);
}
```

3. **构造 TCP SYN 数据包**：为了正确计算 TCP 头部的校验和，需要构造一个伪头部，并填充 TCP 头部各个字段。

```
struct pseudohdr *ptcph = (struct pseudohdr*)sendbuf;
ptcph->saddr = LocalHostIP;
```

```
...
ptcph->length = htons(sizeof(struct tcphdr));
struct tcphdr *tcph = (struct tcphdr*)(sendbuf + sizeof(struct pseudohdr));
tcph->th_sport = htons(LocalPort);
tcph->th_dport = htons(HostPort);
...
tcph->th_sum = in_cksum((unsigned short*)ptcph, 20 + 12); // 计算校验和
```

4. **发送 SYN 数据包**：使用 sendto 函数发送构造好的 SYN 包；**接收响应数据包**：使用 read 函数读取网络上的响应数据包。

```
len = sendto(SynSock, tcph, 20, 0, (struct sockaddr *)&SYNScanHostAddr, sizeof(
    SYNScanHostAddr));
if (len < 0) {
  pthread_mutex_lock(&TCPSynPrintlocker);
  pthread_mutex_unlock(&TCPSynPrintlocker);
}
...
len = read(SynSock, recvbuf, 8192);
if (len <= 0) {
  pthread_mutex_lock(&TCPSynPrintlocker);
  pthread_mutex_unlock(&TCPSynPrintlocker);
}
```

5. **分析响应并判断端口状态**：根据接收到的数据包中的 TCP 标志位来判断端口状态。如果收到 SYN-ACK，则端口开放；如果收到 RST，则端口关闭。

```
struct ip *iph = (struct ip *)recvbuf;
struct tcphdr *tcph = (struct tcphdr *)&recvbuf[iph->ip_hl * 4];
if (tcph->th_flags & TH_ACK) {
  pthread_mutex_lock(&TCPSynPrintlocker);
  pthread_mutex_unlock(&TCPSynPrintlocker);
}
if (tcph->th_flags & TH_RST) {
  pthread_mutex_lock(&TCPSynPrintlocker);
```

```
    pthread_mutex_unlock(&TCPSynPrintlocker);
}
```

6. **关闭套接字和清理内存**：完成操作后关闭套接字并删除分配的内存，减少资源泄露。**线程计数器管理**：使用互斥锁更新全局线程计数器，确保线程同步。

```
close(SynSock);
delete p;
...
pthread_mutex_lock(&TCPSynScanlocker);
TCPSynThrdNum--;
pthread_mutex_unlock(&TCPSynScanlocker);
```

（2）Thread_TCPSynScan 函数

1. **初始化参数**：首先，从传入的参数中解析出目标主机的 IP 地址、扫描的起始端口和终止端口，以及本地主机的 IP 地址。这些信息存储在一个结构体 TCPSYNThrParam 中，该结构体通过函数的参数 param 传入。

```
p = (struct TCPSYNThrParam*)param;
HostIP = p->HostIP;
BeginPort = p->BeginPort;
EndPort = p->EndPort;
LocalHostIP = p->LocalHostIP;
```

2. **端口遍历和线程创建**：函数接着进入一个循环，从起始端口遍历到终止端口。对于每个端口，函数会创建一个子线程的参数结构体 (TCPSYNHost-ThrParam)，并填充它包括目标主机 IP、当前扫描的端口、本机用于发送的端口（计算得出），以及本机的 IP 地址。然后，设置线程属性为分离状态，这意味着线程结束时会自动释放所有资源。这样做可以避免内存泄漏，使得主线程不需要显式地回收每个子线程的资源。接着，子线程执行 Thread_TCPSYNHost 函数，这个函数负责实际发送 SYN 包并接收响应。

```
struct TCPSYNHostThrParam *pTCPSYNHostParam = new TCPSYNHostThrParam;
```

```
pTCPSYNHostParam->HostIP = HostIP;
 ...
pTCPSYNHostParam->LocalHostIP = LocalHostIP;
 pthread_attr_init (&attr);
 pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_DETACHED);
pthread_create (&subThreadID, &attr, Thread_TCPSYNHost, pTCPSYNHostParam);
 pthread_attr_destroy (&attr);
```

3. **线程数量控制**：为避免同时运行的线程数过多，消耗过多系统资源，函数内部设置了线程数量的上限（例如，100 个线程）。

```
pthread_mutex_lock(&TCPSynScanlocker);
TCPSynThrdNum++;
pthread_mutex_unlock(&TCPSynScanlocker);
 ...
while (TCPSynThrdNum > 100) {
    sleep (3);
}
```

4. **等待所有子线程完成**：最后，主扫描线程将在一个循环中等待，直到所有子线程都已经结束。这通过检查一个共享的线程数量计数器 TCPSynThrdNum 实现，当计数器归零时，循环结束。所有子线程完成后，主扫描线程输出结束信息并退出。

```
while (TCPSynThrdNum != 0) {
    sleep (1);
}
 ...
pthread_exit (NULL);
```

（五） 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_TCPFINHost 函数

此函数为 TCP FIN 扫描的核心，处理单个端口的扫描过程。

1. **初始化和套接字创建**：首先从传入的结构体参数中获取目标主机的 IP、端口以及本机的端口和 IP 地址，然后创建两个原始套接字，一个用于发送 FIN 包，另一个用于接收响应。
2. **构造并发送 FIN 包**：构建一个 TCP 包，其中 TCP 头部的 th_flags 字段设置为 TH_FIN，表示这是一个结束连接的请求。之后，通过 sendto 函数发送这个包。
3. **接收和分析响应**：使用非阻塞模式的套接字接收响应。如果在指定的时间内（例如 5 秒）收到来自目标主机的 RST 响应，认为端口是关闭的。如果在这段时间内没有收到任何响应，则认为端口可能开放。
4. **超时处理**：通过计时来判断是否超过了等待响应的时间限制。
5. **资源清理和线程同步**：完成扫描后，关闭套接字，释放分配的资源，并通过互斥锁更新全局线程计数器。

```
1 FinSock = socket(PF_INET, SOCK_RAW, IPPROTO_TCP);
2 FinRevSock = socket(PF_INET, SOCK_RAW, IPPROTO_TCP);
3  ...
4 tcph->th_flags = TH_FIN;
5 sendto(FinSock, tcph, 20, 0, (struct sockaddr *)&FINScanHostAddr, sizeof(
      FINScanHostAddr));
6 len = recvfrom(FinRevSock, recvbuf, sizeof(recvbuf), 0, (struct sockaddr*)&FromAddr, (
      socklen_t*)&FromAddrLen);
7 gettimeofday(&TpEnd,NULL);
8 TimeUse = (1000000 * (TpEnd.tv_sec - TpStart.tv_sec) + (TpEnd.tv_usec - TpStart.tv_usec
      )) / 1000000.0;
9  ...
```

（2）Thread_TCPFinScan 函数

该函数作为主扫描线程，负责管理和创建对指定端口范围的扫描子线程。

1. **端口遍历和子线程创建**：从开始端口到结束端口遍历，为每个端口初始化扫描参数并创建一个 Thread_TCPFINHost 子线程。
2. **线程属性设置**：将子线程设置为分离状态，这样子线程结束后会自动释放所有资源。
3. **线程数量控制**：确保同时活动的线程数量不超过 100，以避免过度消耗系统资源。
4. **等待所有子线程完成**：在所有子线程结束前，主线程持续等待。
5. **线程结束**：所有子线程结束后，输出扫描完成的消息并退出。

```
for (TempPort=BeginPort;TempPort<=EndPort;TempPort++) {
        struct TCPFINHostThrParam
        ...
        pthread_attr_init (&attr);
        pthread_attr_setdetachstate (&attr,PTHREAD_CREATE_DETACHED);
        ret=pthread_create(&subThreadID,&attr,Thread_TCPFINHost,pTCPFINHostParam);
        pthread_attr_destroy (&attr);
        pthread_mutex_lock(&TCPFinScanlocker);
        ...
        pthread_mutex_unlock(&TCPFinScanlocker);
        while (TCPFinThrdNum>100)
            sleep(3);
    }
    while (TCPFinThrdNum != 0)
            sleep(1);
        ...
    pthread_exit (NULL);
```

（六） UDP 扫描

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。

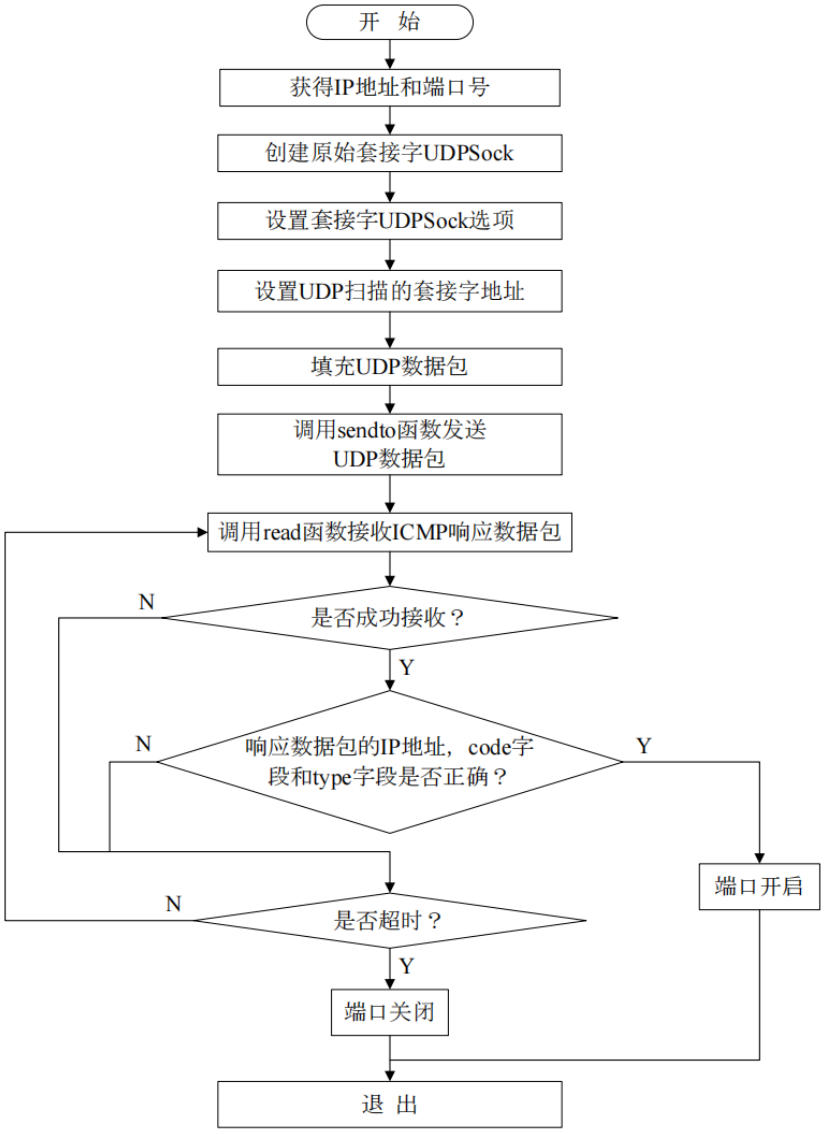

图 3.5 函数 UDPScanHost 流程图

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

（1）UDPScanHost 函数

此函数负责发送 UDP 数据包到指定的目标主机和端口，并接收可能的 ICMP 响应。以下是关键步骤：

1. **套接字创建与设置**：首先创建一个原始套接字，用于发送 UDP 数据包并接收 ICMP 响应。使用 SOCK_RAW 和 IPPROTO_ICMP 参数创建套接字，以允许处理 ICMP 数据包。

```
UDPSock = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
```

2. **数据包构造**：构造 UDP 数据包，其中包括填充 IP 头和 UDP 头。设置了源端口、目的端口、以及数据包长度。UDP 伪头部被用于计算校验和。

```
udp->source = htons(LocalPort);
udp->dest = htons(HostPort);
udp->len = htons(sizeof(struct udphdr));
udp->check = in_cksum((u_short *)pseudo, sizeof(struct udphdr) + sizeof(struct
    pseudohdr));
```

3. **发送数据包**：通过前面设置的套接字和目标地址发送 UDP 数据包。

```
sendto(UDPSock, packet, ip->tot_len, 0, (struct sockaddr *)&UDPScanHostAddr, sizeof(
    UDPScanHostAddr));
```

4. **发送数据包**：通过前面设置的套接字和目标地址发送 UDP 数据包。

```
n = read(UDPSock, (struct ipicmphdr *)&hdr, sizeof(hdr));
if ((hdr.ip.saddr == inet_addr(&HostIP[0])) && (hdr.icmp.code == 3) && (hdr.icmp.type ==
    3)) {
    cout << "Host: " << HostIP << " Port: " << HostPort << " closed !" << endl;
}
```

5. **资源清理**：关闭套接字并释放动态分配的内存。

（2）Thread_UDPScan 函数

Thread_UDPScan 主要用于组织和控制 UDP 扫描过程。它从传入的参数结构体中提取目标主机的 IP 地址、起始端口和结束端口，然后逐个端口调用 UDP-ScanHost 函数来执行实际的扫描。

1. **主要参数**：函数首先从传入的结构体 UDPThrParam 中提取目标主机的 IP 地址、扫描的起始端口号和结束端口号，以及本机的 IP 地址和端口号。这些信息用于构建每个端口扫描的参数。

```
p = (struct UDPThrParam*)param;
HostIP = p->HostIP;
BeginPort = p->BeginPort;
EndPort = p->EndPort;
LocalHostIP = p->LocalHostIP;
```

2. **端口遍历**：在提取了所有必要的参数之后，函数进入一个循环，从 BeginPort 开始到 EndPort 结束。对于循环中的每个端口，它都会执行以下操作：（1）设置子扫描任务的参数：为每个端口创建一个新的 UDPScanHostThrParam 结构体实例，填充相应的参数，包括目标主机的 IP 地址、当前扫描的端口、本机用于发送的端口（这里设置为端口号加上 1024，作为源端口）和本机 IP 地址。（2）执行端口扫描：调用 UDPScanHost 函数，将刚才创建的参数传递给它，以执行对特定端口的扫描。

```
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);
}
```

3. **扫描完成**：在所有端口都被扫描之后，函数输出一个结束消息，并通过

pthread_exit(NULL); 正式结束线程。由于这个函数是顺序执行的，每个端口的扫描都必须等待前一个完成后才能开始。这意味着扫描的总时间将是单个端口扫描时间的累加。在实际应用中，如果端口范围较大，这可能导致整个扫描过程相当缓慢。此外，该方法未采用多线程或异步 I/O，可能无法充分利用现代多核处理器的性能。

```
1 pthread_exit (NULL);
```

（七） 主程序

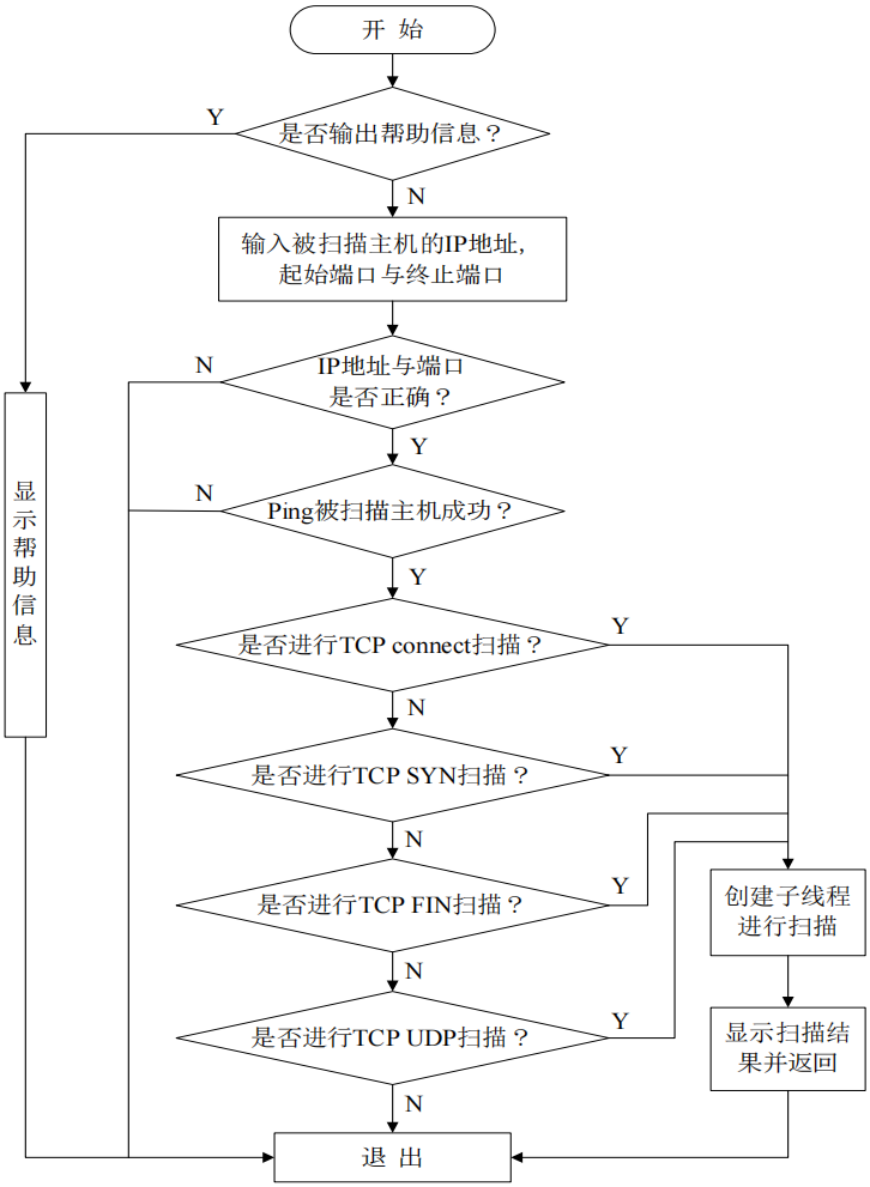

图 3.6 主程序流程图

这部分代码是一个网络扫描器的主控制程序，它根据命令行参数选择并执行不同类型的网络扫描。它支持 TCP 连接扫描、TCP SYN 扫描、TCP FIN 扫描以及 UDP 扫描。该程序结合了命令行界面、网络编程和多线程技术，适用于探测指定主机上不同端口的状态。

1. 程序功能与参数解析

- **命令行参数检查**：首先检查命令行参数的数量。如果不是预期的数量，程序会提示参数错误并退出。

```
if (argc != 2) {
    cout<<"Parameter␣error␣!"<<endl;
    return -1;
}
```

- **帮助信息**：如果传入-h 参数，程序会显示帮助信息，列出所有支持的扫描命令和对应的标志。

```
if (!strcmp(pHelp, argv[1])) {
    cout<<"Scanner:␣usage:␣[-h]␣␣␣--help␣information"<<endl;
    ...
    cout<<"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣[-u]␣␣␣--UDP␣scan"<<endl;
    return 1;
}
```

2. IP 地址和端口范围输入

- **输入 IP 地址**：请求用户输入目标主机的 IP 地址，检查 IP 地址的格式是否正确。

```
cout<<"Please␣input␣IP␣address␣of␣a␣Host:";
cin>>HostIP;
if(inet_addr(&(HostIP[0])) == INADDR_NONE) {
    cout<<"IP␣address␣wrong!"<<endl;
    return -1;
}
```

- **输入端口范围**：请求用户输入端口的开始和结束范围，并检查端口范围的有效性。

```
cout<<"Please input the range of port..."<<endl;
cout<<"Begin Port:";
cin>>BeginPort;
cout<<"End Port:";
cin>>EndPort;
if (IsPortOK(BeginPort, EndPort)) {
    cout<<"Scan Host "<<HostIP<<" port "<<BeginPort<<"~"<<EndPort<<" ..."<<
        endl;
}
else {
    cout<<"The range of port is wrong !"<<endl;
    return -1;
}
```

3. 扫描执行

- **获取本地 IP 地址**：函数 GetLocalHostIP() 用于获取执行扫描的本地机器的 IP 地址。

```
LocalHostIP = GetLocalHostIP();
```

- **Ping 主机**：使用 Ping 函数测试是否可以达到目标主机，若不能，则终止扫描。

```
if (Ping(HostIP, LocalHostIP) == false ) {
    cout<<"Ping Host "<<HostIP<<" failed, stop scan it !"<<endl;
    return -1;
}
```

4. 扫描类型选择与线程管理

对于每种扫描类型（TCP connect, TCP SYN, TCP FIN, UDP），执行步骤如下：

- **创建扫描线程**：为每种扫描类型创建一个新线程，传递包含 IP 地址和端口信息的参数结构体。

```
ret = pthread_create(&ThreadID, NULL, Thread_TCPconnectScan, &TCPConParam);
```

- **线程等待**：使用 pthread_join 等待扫描线程完成，确保在进入下一步之前所有扫描活动已经结束。

```
ret = pthread_join(ThreadID, NULL);
if (ret != 0) {
    cout<<"call␣pthread_join␣function␣failed␣!"<<endl;
    return -1;
}
else {
    cout<<"TCP␣Connect␣Scan␣finished␣!"<<endl;
    return 0;
}
```

（八） 编写 makefile 文件

在处理包含多个.cpp 文件的程序时，逐行输入编译和链接命令不仅繁琐，而且容易出错。为了解决这一问题，可以在这些代码文件所在的目录中创建一个 makefile 文件。这样，每次编译时，只需在 Shell 命令行中输入 make 命令即可。以下是该程序的 makefile 文件内容。由于程序中使用了多线程编程技术，因此在最终链接生成可执行文件时需添加 -pthread 参数才能通过。此外，makefile 文件中还包含 make clean 命令，用于清理编译过程中生成的中间文件，方便下次编译。只需在 Shell 命令行中输入 make clean，即可删除在编译链接过程中生成的文件。

```
# 变量定义
SRC := $(wildcard ./src/*.cpp)          # 源文件搜索，自动寻找 src 目录下的所有 .cpp
```

```
    文件
OBJ := $( patsubst ./ src/%.cpp, ./ obj/%.o, $(SRC)) # 生成目标文件路径，将 src 替换为 obj 目录
INCLUDE := ./include/                    # 包含头文件的目录
CC := g++                                # 编译器
CXXFLAGS := −w −I$(INCLUDE)              # 编译选项，−w 禁止编译警告，−I 指定包含目录
LIBS := −lpthread                        # 链接的库，这里链接了 pthread
TARGET := Scanner                        # 目标文件名

# 虚拟目标
.PHONY: all clean run

# 主要目标
all : $(TARGET)

# 将源代码编译成目标文件
$(OBJ): ./ obj/%.o : ./ src/%.cpp
    $(CC) −c $< −o $@ $(CXXFLAGS)        # $< 表示依赖列表中的第一个依赖文件，$@ 表示目标文件

# 链接目标文件生成最终可执行文件
$(TARGET): $(OBJ)
    $(CC) $^ $(LIBS) −o $@               # $^ 表示所有的依赖文件，$@ 表示目标文件

# 清理生成的文件
clean:
    rm −rf $(TARGET) $(OBJ)              # 删除目标文件和所有目标对象文件

# 运行程序
run:
    ./ $(TARGET)                         # 执行生成的目标文件
```

Makefile 脚本定义了源文件的搜索路径、编译器选项、头文件目录和链接的库。通过具体的规则，自动将 src 目录下的.cpp 文件编译为对象文件，并链接成最终的可执行文件 Scanner。此外，还提供了清理和运行的目标，使得开发者可以方便地构建和执行程序。

四、 实验结果

```
fu@fu-virtual-machine:~/桌面/fzy/Lab4_PortScanner$ sudo ./Scanner -c
Please input IP address of a Host:127.0.0.1
Please input the range of port...
Begin Port:1
End Port:255
Scan Host 127.0.0.1 port 1~255 ...
Ping Host 127.0.0.1 Successfully !
Begin TCP connect scan...
Host: 127.0.0.1 Port: 1 closed !
Host: 127.0.0.1 Port: 2 closed !
Host: 127.0.0.1 Port: 9 closed !
Host: 127.0.0.1 Port: 8 closed !
Host: 127.0.0.1 Port: 7 closed !
Host: 127.0.0.1 Port: 6 closed !
Host: 127.0.0.1 Port: 10 closed !
Host: 127.0.0.1 Port: 4 closed !
Host: 127.0.0.1 Port: 3 closed !
Host: 127.0.0.1 Port: 5 closed !
Host: 127.0.0.1 Port: 11 closed !
Host: 127.0.0.1 Port: 12 closed !
Host: 127.0.0.1 Port: 13 closed !
Host: 127.0.0.1 Port: 14 closed !
Host: 127.0.0.1 Port: 15 closed !
```

图 4.1 TCP connect scan

```
fu@fu-virtual-machine:~/桌面/fzy/Lab4_PortScanner$ sudo ./Scanner -u
Please input IP address of a Host:127.0.0.1
Please input the range of port...
Begin Port:1
End Port:255
Scan Host 127.0.0.1 port 1~255 ...
Ping Host 127.0.0.1 Successfully !
Begin UDP scan...
Host: 127.0.0.1 Port: 1 open !
Host: 127.0.0.1 Port: 2 open !
Host: 127.0.0.1 Port: 3 open !
Host: 127.0.0.1 Port: 4 open !
Host: 127.0.0.1 Port: 5 open !
Host: 127.0.0.1 Port: 6 open !
Host: 127.0.0.1 Port: 7 open !
Host: 127.0.0.1 Port: 8 open !
Host: 127.0.0.1 Port: 9 open !
Host: 127.0.0.1 Port: 10 open !
Host: 127.0.0.1 Port: 11 open !
Host: 127.0.0.1 Port: 12 open !
Host: 127.0.0.1 Port: 13 open !
Host: 127.0.0.1 Port: 14 open !
Host: 127.0.0.1 Port: 15 open !
```

图 4.2 UDP scan

```
fu@fu-virtual-machine:~/桌面/fzy/Lab4_PortScanner$ sudo ./Scanner -s
Please input IP address of a Host:127.0.0.1
Please input the range of port...
Begin Port:1
End Port:255
Scan Host 127.0.0.1 port 1~255 ...
Ping Host 127.0.0.1 Successfully !
Begin TCP SYN scan...
Host: 127.0.0.1 Port: 1 closed !
Host: 127.0.0.1 Port: 5 closed !
Host: 127.0.0.1 Port: 4 closed !
Host: 127.0.0.1 Port: 3 closed !
Host: 127.0.0.1 Port: 2 closed !
Host: 127.0.0.1 Port: 6 closed !
Host: 127.0.0.1 Port: 7 closed !
Host: 127.0.0.1 Port: 12 closed !
Host: 127.0.0.1 Port: 11 closed !
Host: 127.0.0.1 Port: 10 closed !
Host: 127.0.0.1 Port: 9 closed !
Host: 127.0.0.1 Port: 13 closed !
Host: 127.0.0.1 Port: 8 closed !
Host: 127.0.0.1 Port: 15 closed !
```

图 4.3 TCP SYN scan

```
fu@fu-virtual-machine:~/桌面/fzy/Lab4_PortScanner$ sudo ./Scanner -s
Please input IP address of a Host:127.0.0.1
Please input the range of port...
Begin Port:1
End Port:255
Scan Host 127.0.0.1 port 1~255 ...
Ping Host 127.0.0.1 Successfully !
Begin TCP FIN scan...
Host: 127.0.0.1 Port: 1 closed !
Host: 127.0.0.1 Port: 5 closed !
Host: 127.0.0.1 Port: 4 closed !
Host: 127.0.0.1 Port: 3 closed !
Host: 127.0.0.1 Port: 2 closed !
Host: 127.0.0.1 Port: 6 closed !
Host: 127.0.0.1 Port: 7 closed !
Host: 127.0.0.1 Port: 12 closed !
Host: 127.0.0.1 Port: 11 closed !
Host: 127.0.0.1 Port: 10 closed !
Host: 127.0.0.1 Port: 9 closed !
Host: 127.0.0.1 Port: 13 closed !
Host: 127.0.0.1 Port: 8 closed !
Host: 127.0.0.1 Port: 15 closed !
```

图 4.4 TCP FIN scan

```
fu@fu-virtual-machine:~/桌面/fzy/Lab4_PortScanner$ sudo ./Scanner -h
[sudo] fu 的密码:
Scanner: usage: [-h]  --help information
                [-c]  --TCP connect scan
                [-s]  --TCP syn scan
                [-f]  --TCP fin scan
                [-u]  --UDP scan
```

图 4.5 help info

五、 实验结论

通过这次实验，我对网络协议栈的理解更加深刻，特别是对 TCP/IP 协议的各层作用和功能有了更加具体的认识。实验中，我深入了解了 TCP 和 UDP 协议

的不同扫描技术，如 TCP 的三种扫描方法（Connect 扫描、SYN 扫描、FIN 扫描）以及 UDP 扫描的特殊性。

编写端口扫描程序的过程中，我面临了多个挑战，包括对套接字编程的掌握、多线程的使用以及原始套接字的权限问题。通过不断的试验和错误，我逐步掌握了 Linux 环境下的套接字编程技术，特别是在网络编程中错误处理和资源管理的重要性。例如，必须确保每个套接字在使用完毕后正确关闭，以避免资源泄漏。此外，我也体会到了多线程编程在实际应用中的强大功能，它能显著提高程序的执行效率。在实现 TCP 连接扫描和 SYN 扫描的过程中，合理地使用线程池管理线程，使得端口扫描任务可以并行处理，大大缩短了总体的扫描时间。

通过这次实验，我不仅提升了编程技能和问题解决能力，还增强了对网络安全领域的兴趣。这次实验让我认识到理论知识与实际操作之间的差距，并激发了我进一步深入研究网络安全技术的热情。

参考文献

[1] 吴功宜. 计算机网络高级教程[J]. 计算机教育, 2008(1): 1