天津大学

《计算机网络实践》课程报告

TCP 在应用层的设计与实现

学　　号 3022244290

姓　　名 陈秋澄

学　　院 智能与计算学部

专　　业 计算机科学与技术

年　　级 2022 级

任课教师 赵增华

2024 年 8 月 31 日

一、报告摘要

简要介绍需要解决的具体问题、协议设计和实现，以及主要实验结果。（可放到最后写）

二、任务要求

本次实践需要在应用层使用 UDP 作为基本协议实体并在其基础上实现 TCP 的基本功能。

UDP 协议作为基础的网络层通信协议，其核心职责仅限于将报文传递给下层的协议实体，并不包含如 TCP 协议所提供的可靠性保障等特性。然而，正因如此，UDP 协议能够实现更高的平均传输速率，使其特别适用于那些对可靠性要求较低，但对实时性要求较高的应用场景，如流媒体传输服务。

TCP 的主要目标是在不同进程间提供稳定且安全的面向连接服务。为了在相对不可靠的通信环境中实现这一目标，系统需要具备以下功能：连接的建立与关闭、基础的数据传输机制、确保通信的可靠性以及实施流量控制。

2.1 连接管理

TCP 客户端首先在三次握手后与指定的服务器建立连接，随后通过此连接与服务器进行数据交换，最终通过四次挥手关闭该连接。此外，需要在有丢包和延迟的情况下，正确处理在建立连接时每一种数据包丢失的情况以及关闭连接时服务器端和客户端先后或同时断开时的情况；并且保证数据传输的稳定性。

连接管理部分，主要针对：连接建立——三次握手以及连接关闭——四次挥手两方面。

2.2 可靠数据传输

TCP 给发送的每一个数据包都分配一个唯一的序列号。接收端在接收到数据包后，会根据序列号对数据进行排序，并将有序的数据传送给应用层。序列号确保了数据的按序到达，并允许接收端丢弃重复的数据包。并且，接收端在成功接收到数据包后，会向发送端发送一个确认应答，告知其已接收到的最后一个数据包的序列号。如果发送端在预设的时间内未收到确认应答，则认为数据包可能已丢失，将重新发送该数据包。此外，TCP 为每一个已发送的数据包设置一个超时计时器。如果在超时计时器到期之前未收到相应的确认应答，则发送端将重发该数据包。在本模块中，我们需要完成以下任务：

1. 开发并集成序列号分配、数据校验、累积确认机制以及超时重传功能，确保数据传输的完整性和可靠性。
2. 部署定时器机制，依据 RFC793 Sec 3.7 的描述，以动态追踪并实时评估往返时延（RTT），优化数据传输效率。
3. 正确管理发送与接收缓冲区、实现窗口滑动机制，以控制数据流、避免

拥塞：在 TCP 协议的数据传输优化策略中，滑动窗口技术通过自适应地调整窗口尺寸，控制两台通信主机间的数据流。具体到数据传输协议的选择上，采用 GBN 协议来实现以上关键功能。

2.3 流量控制

流量控制的核心目的是对系统中的数据流、请求或信息流量进行管理和调节，以确保系统的稳定运行和资源的合理利用。本模块需保证：

1. 接收端具备评估当前空闲缓冲区容量的能力，据此设定“Advertised Window”值。
2. 发送端依据从接收端获取的“Advertised Window”信息，灵活调整其自身的数据发送范围（发送窗口）。
3. 针对接收端“Advertised Window”显示为 0 的特殊情形，发送端采取恰当的应对措施，执行有效的零窗口探测机制，以确保数据传输的顺畅与高效。

2.4 拥塞控制

拥塞控制是计算机网络中一个重要的概念，旨在防止过多的数据注入网络，通过调节发送方的发送速率来避免网络中的路由器或链路过载，确保网络性能的稳定性和高效性。

依照 TCP Reno 要求，本模块需要实现以下三种机制：慢启动、拥塞避免、快速重传，即在发生超时、冗余重传、收到 new ack 等几种情况下的拥塞状态转化。

三、协议设计

注意：协议设计的内容要涵盖“计算机网络课程实践说明书”中“二、TCP 功能需求”要求的所有内容。这是评分的依据。

3.1 总体设计

根据任务分析划分功能模块、理清模块组织结构和关系等。

3.1.1 模块设计

根据实践指导书的说明，可将 TJU_TCP 的模块架构划分成以下层次，如图 3-1 所示：

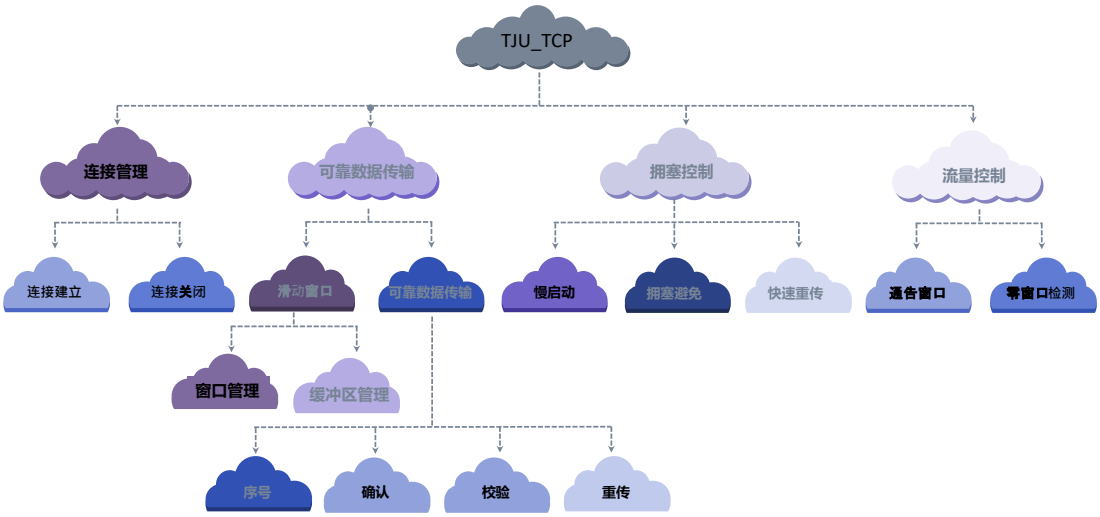

图 3-1 功能模块关系

3.2 连接管理的设计

说明三次握手建立连接、四次挥手关闭连接的原理、主要数据结构和协议规则。用 FSM 图表示主要工作流程。

3.2.1 原理

1. 连接建立相关原理

TCP 进行三次握手建立连接时，操作系统会为其维护两个队列，分别是半连接队列和全连接队列。三次握手过程原理如下：

首先 client 端发起第一次握手，发送请求连接的报文段报文完成发送后，进入 SYN_SENT 状态。相关参数为 SYN = 1，seq = n。

接着，server 端收到报文，将该连接加入到 SYN 半连接队列中，并在对该报文段校验后向 client 端发送响应报文，进入 SYN_RECV 状态。相关参数为 SYN = 1，ACK = 1，seq = m，ack = n+1。此时，第二次握手完成。

最后，第三次握手。client 端收到报文段，分别设置 ACK = 1，seq = n+1，ack = m+1，并发送报文，此后进入 ESTABLISHED 状态；server 端收到 client 端第三次握手响应后，将该连接从半连接队列中取出，建立 1 个新连接，加入到 ACCEPT 全连接队列中，同样将状态转为 ESTABLISHED。

此后当进程调用 accept()函数时，再将该连接取出来。

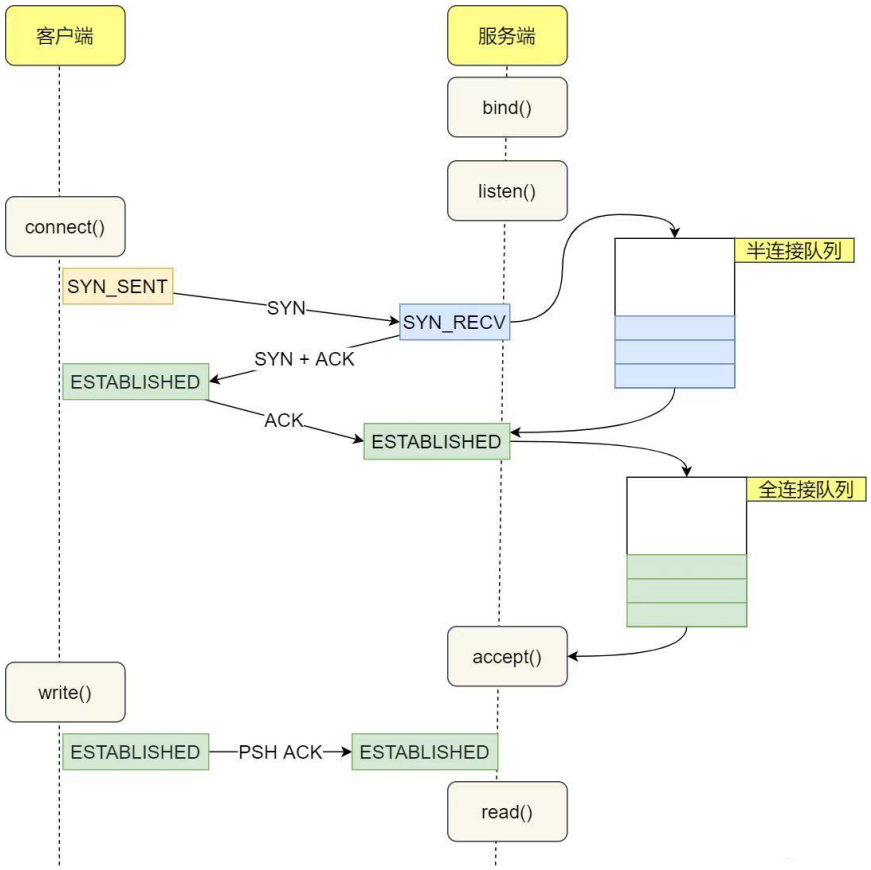

图 3-2 三次握手流程

2. 连接关闭相关原理

(1)双方先后断开连接

首先，client 端发起第一次挥手，状态由 ESTABLISHED 转为 FIN_WAIT_1，向 server 端发送的报文相关参数为连接终止位 FIN = 1，seq = u。当 client 端发送 FIN 报文时，表示其已没有数据要发送了。当然，client 端此时还是可以接收 server 的数据的。

Server 端收到 client 端的报文，发起第二次挥手并将状态从 ESTABLISHED 转为 CLOSE_WAIT，并等待 server 端自身的 socket 关闭等操作。报文相关参数为 ACK =1, seq = v，ack = u+1。

Client 端收到 server 端发起的第二次挥手，将状态由 FIN_WAIT_1 转为 FIN_Wait_2,等待 server 端关闭。

Server 端发起第三次挥手，状态由 CLOSE_WAIT 转为 LAST_ACK。报文相关参数为 FIN = 1，ACK = 1，seq = w，ack = u+1。

Client 收到 server 端发起的第三次挥手，并发起第四次挥手，状态也随之从 FIN_WAIT_2 转为 TIME_WAIT。报文相关参数为 ACK =1, seq = u+1，ack = w+1。

最后，server 收到 client 端发起的第四次挥手，进入 CLOSED 状态。与此同时，client 端等待 2MSL 后，也同样进入 CLOSED 状态。

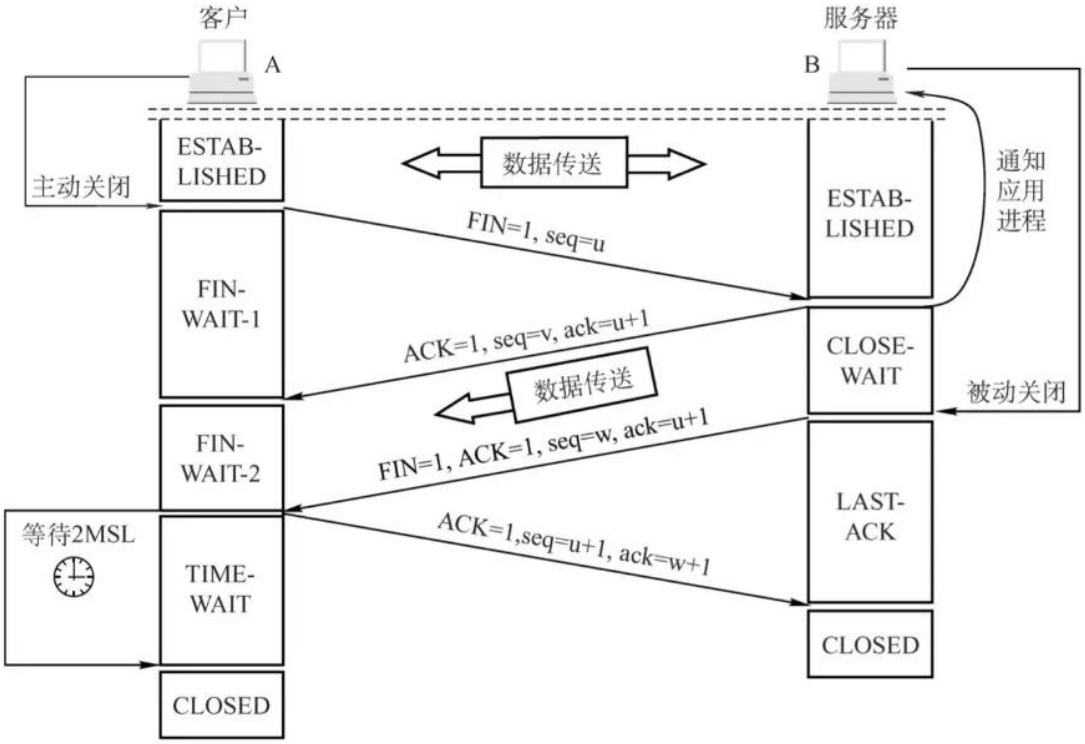

图 3-3 四次挥手流程

（2）双方同时断开连接

客户端和服务端状态对称，因为同时断开连接即同时发出 FIN 包，所以二者的状态转换和上文（1）中的客户端的状态转换类似，只不过把 FIN_WAIT_2 叫做 CLOSING。此外，双方要在没有收到自己发出的 FIN 包对应的 ACK 包的情况下，对对方的 FIN 包做出应答。

3.2.2 主要数据结构

依据上文提到的原理，TCP 报文首部结构如下表所示：

表 1 TCP 报文首部结构

数据结构	含义或作用
source_port	源端口
destination_port	目的端口
seq_num	Sequence 序号
ack_num	Acknowledgement 序号
hlen	包头长度
plen	整个数据包长度
flags	标志位，如 SYN、FIN、ACK
advertised_window	接收方发送给发送方用于流量控制的建议窗口大小
ext	额外的数据，无实际意义

此外，这里厘清一些概念：

1. 序号字段 seq 占 4 字节，TCP 报文首部的序号字段值指本报文所发送的数据的第一个字节的序号。
2. 确认号字段 ack 同样占 4 字节，指期待收到对方下一个报文段的第一个数据字节的序号。
3. 确认 ACK：当且仅当 ACK＝1，确认号字段 ack 才有效，且建立连接后所有报文段的 ACK 均为 1。
4. 同步 SYN：用于在连接建立时同步序号，SYN＝1 指该报文为连接请求或连接接受报文。
5. 终止 FIN：当 FIN＝1，该报文段的发送方的数据已经全部发送完成，并请求释放传输连接。

依据上文，TCP 进行三次握手建立连接时，操作系统会为其维护两个队列，分别是半连接队列和全连接队列。因此，增加相关数据结构创建队列。

表 2 socket 相关数据结构

数据结构	含义或作用
tju_tcp_t* syn_queue[MAX_SOCK]	半连接队列
uint16_t syn_num	半连接队列元素个数
pthread_mutex_t syn_lock	半连接队列的锁
tju_tcp_t* accept_queue[MAX_SOCK]	全连接队列
uint16_t accept_num	全连接队列元素个数
pthread_mutex_t accept_lock	全连接队列的锁
tju_tcp_t* get_from_syn()	从半连接队列中取出 socket
tju_tcp_t* get_from_accept()	从全连接队列中取出 socket

下面，对 socket 地址数据结构进行说明：

表 3 socket 地址数据结构

数据结构	含义或作用
typedef struct {	结构体
uint32_t ip;	IP 地址
uint16_t port;	端口号
}tju_sock_addr;	

3.2.3 协议规则

1. 连接建立的协议规则

（1）基本流程

处于 CLOSED 状态的 Server 端调用 tju_sock 接口获取处于 LISTEN 状态的 socket。

Client 端从 tju_conn 接口向 Server 端发送报文 a，以请求连接（必要时进行超时重传，在 3.3 节可靠数据传输章节与 RDT 一并阐述），并将状态转为 SYN_SENT。

Server 端收到报文 a，发送报文 b，将状态转为 SYN_RCVD。

Client 端收到报文 b，将状态转为 ESTABLISHED，向上层接口返回一个建立连接的 socket。同时向 server 端发送报文 c。

Server 端 1 收到报文 c，建立新的 socket 并将其放到初始的 socket 全连接队列。此后当进程调用 accept()函数时，再将该连接取出来。

FSM 图如下图所示：

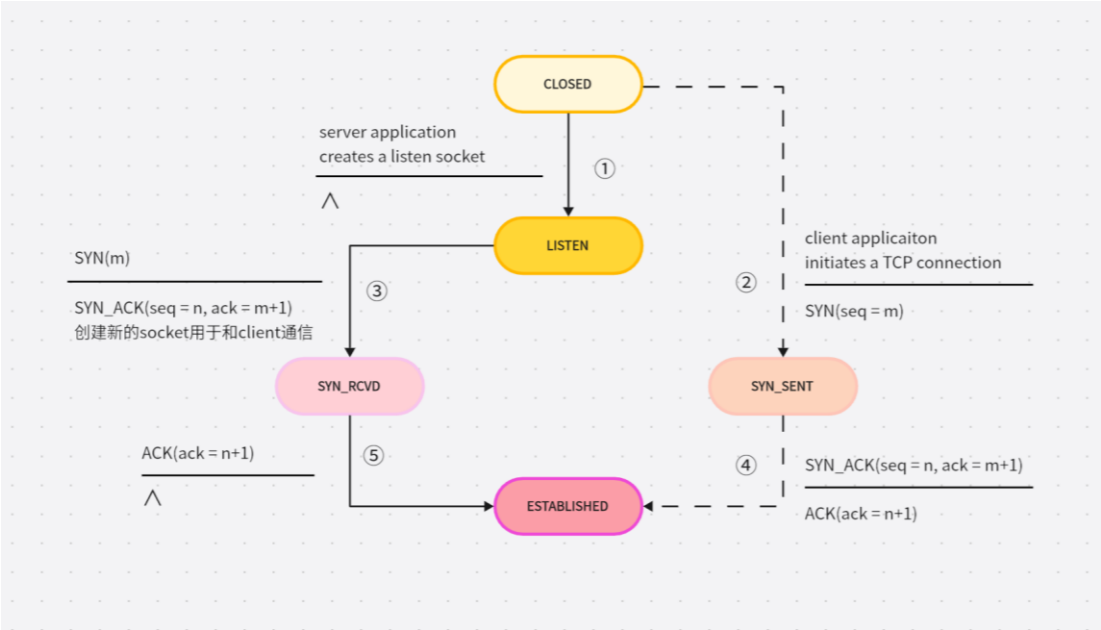

图 3-4 连接建立的状态机

（2）异常情况处理

下面，介绍针对不同丢包情况设计的不同的应对思路。

a) SYN 包丢失

当遇到 SYN 包丢失的问题时，客户端会启动一个专门的定时器线程来监控。若此过程中发生超时，客户端将周期性地重新发送 SYN 包，并且每次重传后，其超时重传时间（RTO）会加倍。这一过程将持续进行，直至客户端接收到来自服务器的第二个包的确认信号，随后关闭该定时器线程。

b) SYN | ACK 包丢失

若 SYN | ACK 包在传输过程中丢失，客户端将无法顺利过渡到下一个状态，因此其定时器线程将保持开启状态。在此情境下，客户端的定时器线程将负责重新发送 SYN 包，以触发服务器再次发送 SYN | ACK 包。同时，服务器端也会启动其定时器线程，以应对可能的重传需求。

c) ACK 包丢失

当 ACK 包未能成功送达服务器时，服务器将停留在当前状态，此时服务器的定时器线程将保持开启状态以监测超时。一旦超时发生，服务器的定时器线程将负责重新发送 SYN | ACK 包，以促使客户端再次发送 ACK 包。这一过程将重复进行，直到服务器成功接收到 ACK 包并更新其状态，随后关闭定时器线程。

2. 连接关闭的协议规则

（1）双方先后断开连接

该情况对应下图右侧流程。

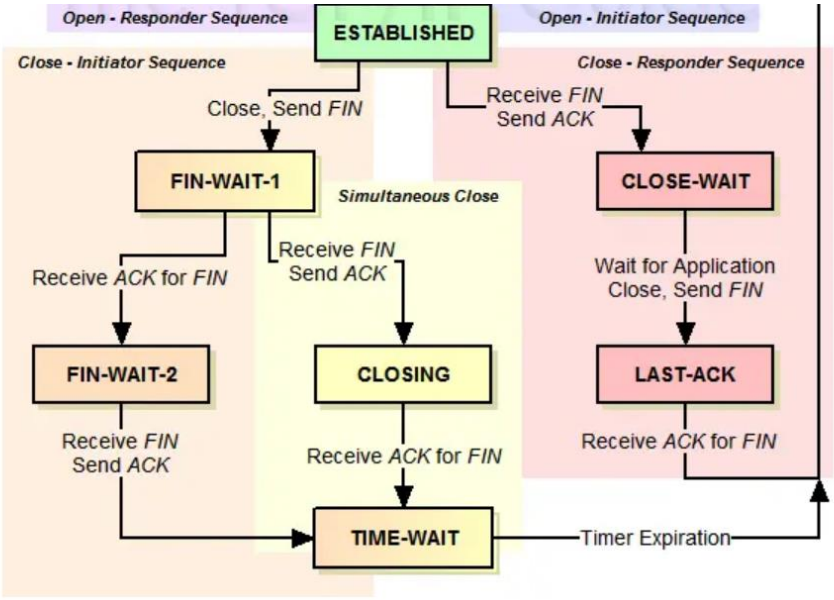

图 3-5 连接关闭总流程

详细阐述如下：

a) 首轮信号（第一次挥手）：
客户端启动断开连接的流程，发送一个包含 FIN 标志（FIN=1）的报文段（序列号 seq=u）。该操作使客户端状态转变为 FIN_WAIT_1，同时停止数据发送，主动终止 TCP 连接。在四次挥手流程中，ACK 报文负责确认接收，而 SYN 报文用于同步连接状态。

b) 服务端的响应确认（第二次挥手）：
服务端在接收到 FIN 报文后，会回复一个 ACK 报文，其序列号被设置为客户端序列号的下一个值（ack=u+1），以此确认已接收到客户端的断开请求。此时，服务端进入 CLOSE_WAIT 状态，TCP 连接进入半关闭状态，即仅允许从服务端向客户端的数据传输。客户端在收到此确认后，其状态转变为 FIN_WAIT_2，准备接收来自服务端的断开信号。

c) 服务端的断开请求（第三次挥手）：
若服务端也决定关闭连接，它将模仿客户端的首次操作，但会同时设置 FIN 和 ACK 标志（FIN=1, ACK=1），并指定一个序列号（seq=w）。此时，服务端状态转变为 LAST_ACK（最终确认），等待客户端的最终确认。

d) 客户端的最终确认与等待（第四次挥手）：
客户端在接收到来自服务端的 FIN|ACK 报文后，会发送一个 ACK 报文作为回应，其序列号设置为服务端发送的 FIN 报文序列号的下一个值（ack=w+1）。此时，客户端进入 TIME_WAIT 状态，通常等待 2MSL（即两倍的最大报文生存时间），以确保服务端收到其 ACK 报文后已安全关闭连接。一旦计时结束，客户端也会进入 CLOSED 状态。这一过程中，TCP 连接在客户端保持活动状态，直至计时结束。

（2）双方同时断开连接

a) 双方同时发起连接终止信号：
客户端与服务器几乎同时决定结束它们之间的通信会话，于是几乎同步发出 FIN 报文，各自随即进入 FIN_WAIT_1 的等待状态，预示连接的断开。

b) 相互确认对方的终止请求：

鉴于双方几乎同时发出 FIN 报文，这些报文在网络上交错传递。一旦接收到对方的 FIN 报文，双方都会迅速响应，通过发送 ACK 确认报文来告知对方其关闭请求已被成功接收；促使双方的状态从 FIN_WAIT_1 转变为一个更为接近终止的状态，可视为 CLOSING 状态。

c) 进入等待确保阶段：

完成 ACK 报文的交换后，为确保所有传输的数据包都能得到妥善处理，包括确认接收到的 FIN 报文，双方都将进入 TIME_WAIT 状态。这一阶段通常持续 2MSL 的长度，让网络中可能存在的残留数据包有足够的时间得以处理，从而防止因 ACK 报文丢失而引发的连接状态不一致问题。

d) 彻底断开连接：

经过 TIME_WAIT 状态设定的时间（即 2MSL）后，双方确信所有必要的通信都已妥善处理，便会同时进入 CLOSE 状态，标志着它们之间的连接已经完全且安全地终止。

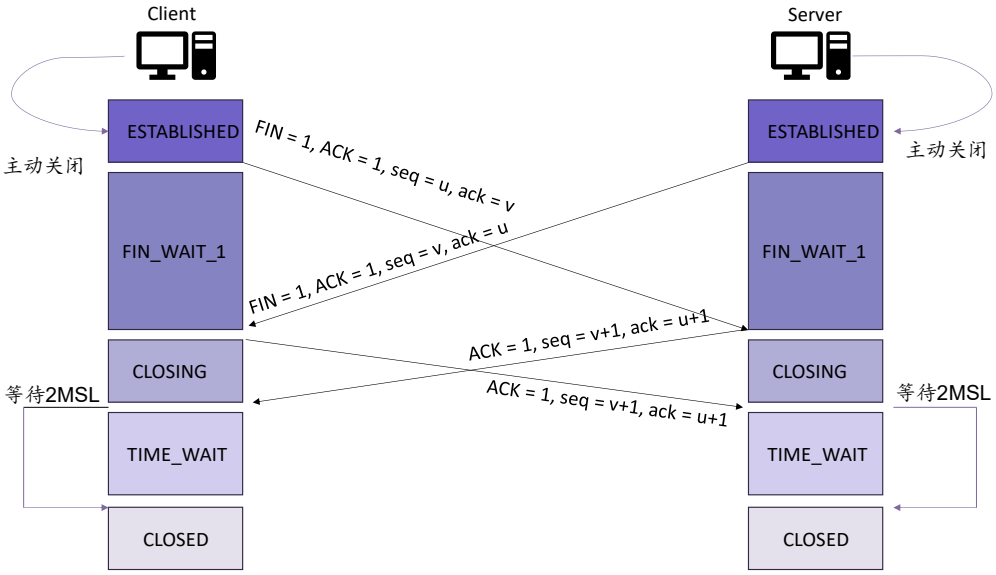

图 3-6 双方同时关闭连接

3.3 可靠数据传输的设计

说明 TCP 可靠数据传输的主要数据结构和协议规则。用 FSM 图表示主要工作流程。

说明发送端和接收端缓冲区的管理，滑动窗口的设计方法。

3.3.1 主要数据结构

3.4 流量控制的设计

说明流量控制的原理和设计方法。

3.5 拥塞控制的设计

说明拥塞控制的原理、主要数据结构和协议规则。用 FSM 图表示主要工作流程。

四、协议实现

详细描述功能实现的细节。主要功能模块使用流程图或者伪代码来辅助说明。禁止贴源码。

注意：协议实现的内容要和“协议设计”部分相对应。每项功能设计都要有相应的实现。

4.1 连接管理的实现

4.1.1 连接建立的实现

1. 实现 connect 和 accept 接口

```
tju_accept(listen_sock)
```

```
if 全连接队列中无 sock，则阻塞；
else
    队列变更(-1);
    for(int i=0;i<MAX_SOCK;i++)
        if(acceptqueue[i]!=NULL)
            // 将接受的连接放入已建立的连接表中
            established_socks[i] = acceptqueue[i];
            acceptqueue[i]=NULL;
            // 设置新连接为当前连接
            new_conn = established_socks[i];
            break;
```

```
tju_connect(sock, target_addr)
```

```
给 sock 绑定本地和目标 ip 址及端口
将建立了连接的 socket 放入内核 已建立连接哈希表中
// 发送 SYN（第一次挥手）
Create packet;
sendToLayer3(syn, LEN);
sock->state = SYN_SENT;
阻塞等待直到 socket 状态变为 ESTABLISHED
```

2. 状态转换的实现

连接建立针对三种状态：

（1） socket 的状态为 LISTEN，由 server 端处理：

```
    sock->state == LISTEN：
1       if 收到不是 SYN 或者目的地不是 server：忽视它；
2       else 分配 new_sock 建立连接
3       更改状态为 SYN_RECV；
4       把 new_conn 放到半连接队列；
5       将建立了连接的 socket 放入内核全连接哈希表中；
6       server 端发送 syn+ack 包，进行第二次挥手。
7       此外，若数据包标志位为 SYN_FLAG_MASK 且当前状态 SYN_RECV：
8           进行序号 6 的操作
```

（2） socket 的状态为 SYN_RECV，由 client 端处理：

```
sock->state == SYN_RECV，server 等待 client 的 ACK：
    将半连接删除，放入全连接队列
    状态变为 ESTABLISHED
    等待 tju_accept 调用
```

（3） socket 的状态为 SYN_SENT，由 client 端处理：

```
sock->state == SYN_SENT，client 等待第二次握手的 SYN|ACK ：
    if(dst!=sock->established_local_addr.port||flags!=ACK_FLAG_MASK+SYN_
FLAG_MASK)
        不是期待的 SYN|ACK，忽略该数据包，由发送线程进行重发
    将半连接删除，全连接放入
    状态变为 ESTABLISHED
    返回 ACK 报文答复，client 确认 TCP 建立成功
```

4.1.2 连接关闭的实现

从代码实现的角度考虑，我们不需要在意当前情况是两方同时发起关闭还是双方先后关闭。可以分析状态转换图、根据 socket 状态以及接收到的报文进行状态的转换以及处理：

1. tju_close 函数的实现

```
tju_close (sock)
    ①Client 端调用 close（）
        当发送缓冲区内还有数据没发完，则阻塞；
        // 发送 FIN
        sendToLayer3(msg ,DEFAULT_HEADER_LEN);
```

```
        sock->state = FIN_WAIT_1;
    ②server 端调用 close（）
        sleep(1);
        这种情况为双方同时关闭
    若此时状态不是 CLOSED，则阻塞
    释放 sock
```

2. 状态转换的实现

（1）socket 的状态为 ESTABLISHED

```
1   If：get_flags(pkt)==ACK_FLAG_MASK
2       if 收到不是 SYN 或者目的地不是 server：忽视它；
3       序号小于期望的序号，丢弃这个包，发送一个 ack 告知发送端期望序号
4       序号等于期望的序号，接收这个包，放入缓冲区
5           if 接收缓冲区已满，丢弃分组
            if 失序报文缓冲区为空，直接放入顺序缓存
            else 将失序缓冲区与该包序号连续的报文取出发送到顺序缓冲区
            接下来恢复无序缓冲区，使得其依然从游标 0 开始存储报文
6       序号大于期望的序号,放入失序缓冲区
7   If：get_flags(pkt) == FIN_FLAG_MASK
8       开始进行第二次挥手
9       发送包
10      变更状态，socket->CLOSE_WAIT
11      server 端在一段时间后发送 FIN
12      开始进行第三次挥手
13      发送包
14      变更状态，socket->LAST_ACK
```

（2）socket 的状态为 FIN_WAIT_1

```
1   If：flag==ACK_FLAG_MASK
2       变更状态，sock->state = FIN_WAIT_2；
3   If: flags == ACK_FLAG_MASK +FIN_FLAG_MASK
4       发送 ack 数据包
5       变更状态，sock->state = CLOSING;
```

（3）socket 的状态为 FIN_WAIT_2

```
1   If:dst!=sock->established_local_addr.port||flags!=
    ACK_FLAG_MASK+FIN_FLAG_MASK
2       该数据包不是第四次挥手期待的 FIN_ACK，忽略
3   If: flags == ACK_FLAG_MASK+FIN_FLAG_MASK
4       sendToLayer3(ack_pkt ,DEFAULT_HEADER_LEN);
5       sock->state = TIME_WAIT;
6       一段时间后，sock->state = CLOSED;
```

（4）socket 的状态为 CLOSING

```
If：sock->state == CLOSING && get_flags(pkt) == ACK_FLAG_MASK
    sock->state = TIME_WAIT;
    一段时间后，sock->state = CLOSED;
```

（5）socket 的状态为 LAST_ACK

```
If：sock->state == LAST_ACK && flags != ACK_FLAG_MASK
    sock->state = CLOSED;
```

下面，再用状态转换图表明 server 端和 client 端的状态变化：

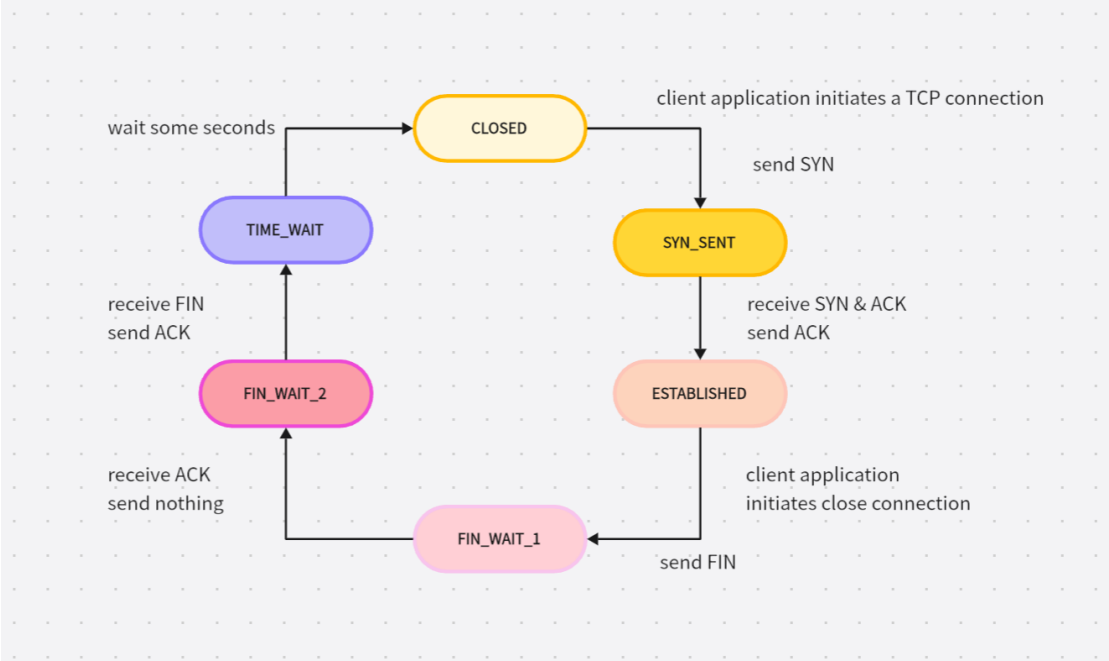

图 4-1 Client 端状态转换

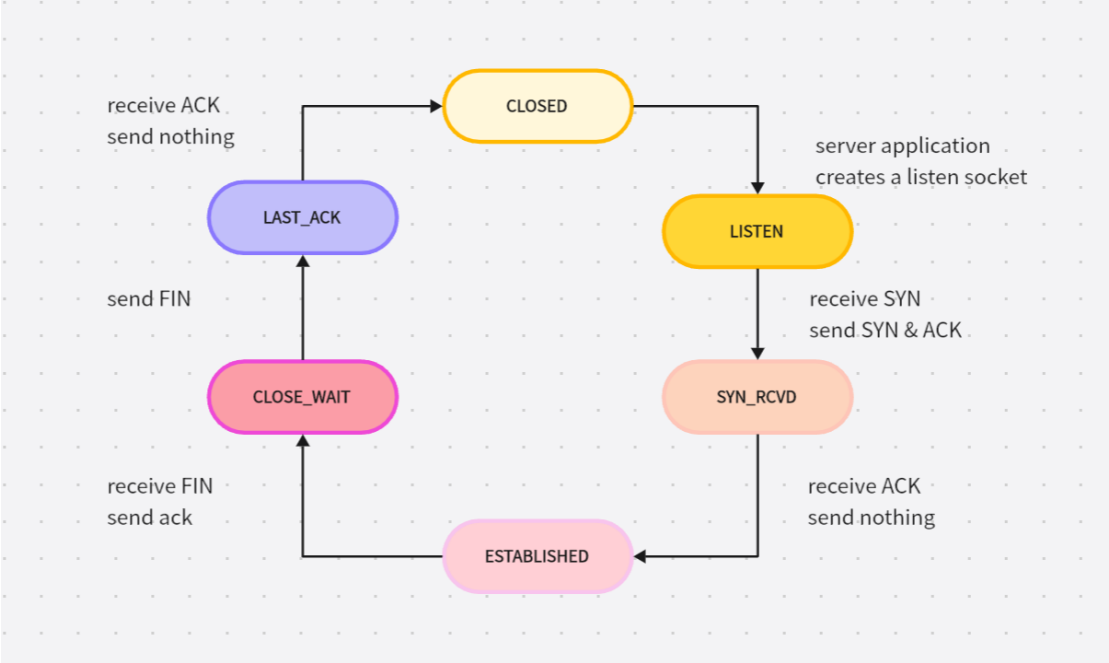

图 4-2 server 端状态转换

4.2 可靠传输的实现

4.3 流量控制的实现

4.4 拥塞控制的实现

五、实验结果及分析

测试所实现协议的功能和性能，并对性能结果进行分析。需要针对考察点逐一展开。

5.1 连接管理的功能测试与结果分析

1. 本地测试与平台测评

本地测试 establish 和 close 均为 100 分：

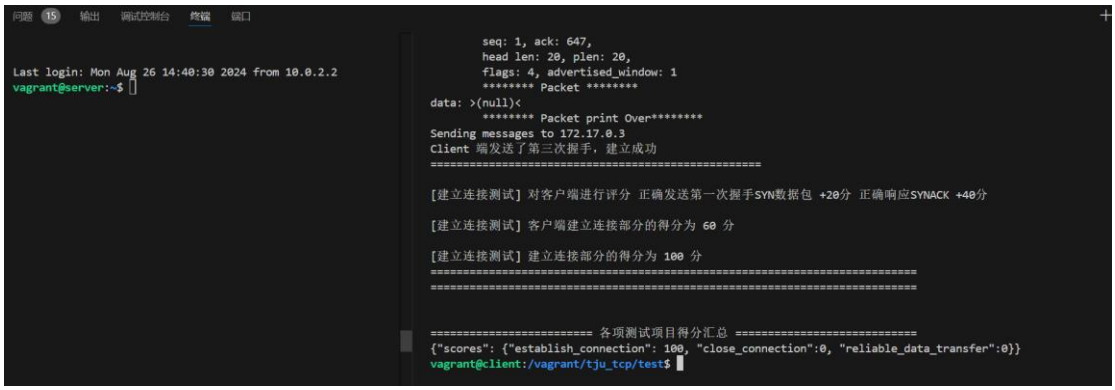

图 5-1 连接建立本地结果

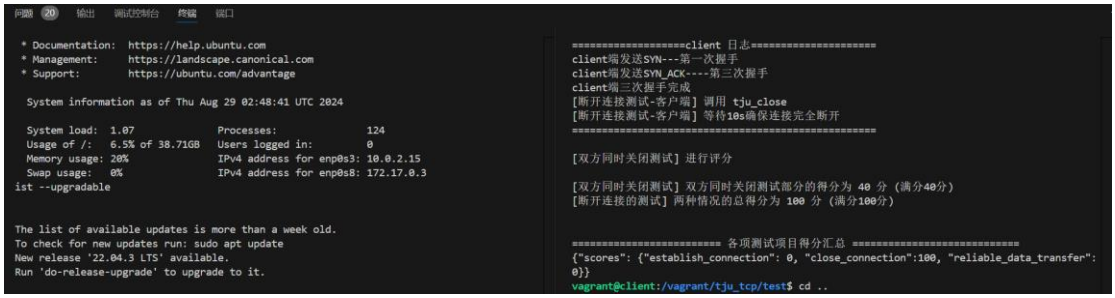

图 5-2 连接关闭本地结果

平台测试：

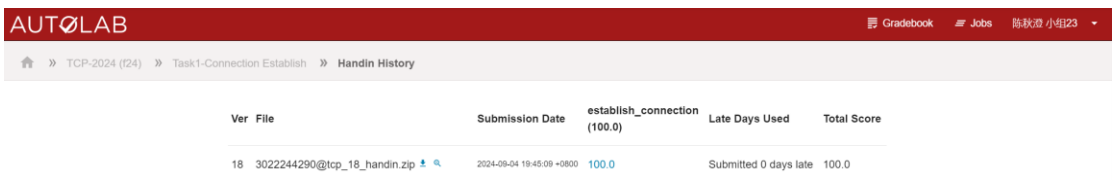

图 5-3 连接建立平台结果

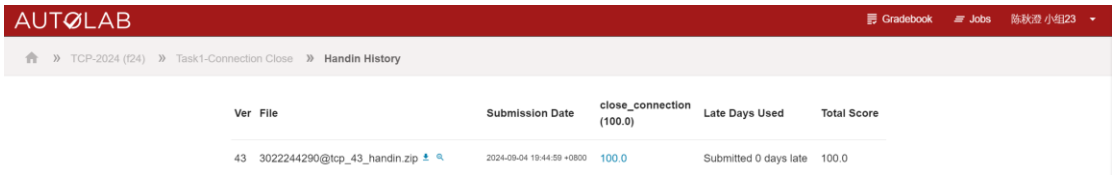

图 5-4 连接关闭平台结果

2. 实验结果分析

通过测评结果，已完成 TCP 连接建立与关闭的模块，并实现日志模块，将状态实时打印，成功实现第一周的连接管理任务。

5.2 可靠传输的功能测试与结果分析

5.3 流量控制的功能测试与结果分析

5.4 拥塞控制的功能测试与结果分析

5.5 TCP 协议性能测试与结果分析

六、总结

总结在实践过程中遇到的各类问题、困难以及解决过程中的收获，对实践内容等方面的体会与建议。

6.1 问题与解决

6.1.1 第一周

1. 软件安装

起初，本人的 win11 系统无法安装这种情况给定 6.1.26 版本的 virtualbox，双击可执行文件后弹出如下指示：

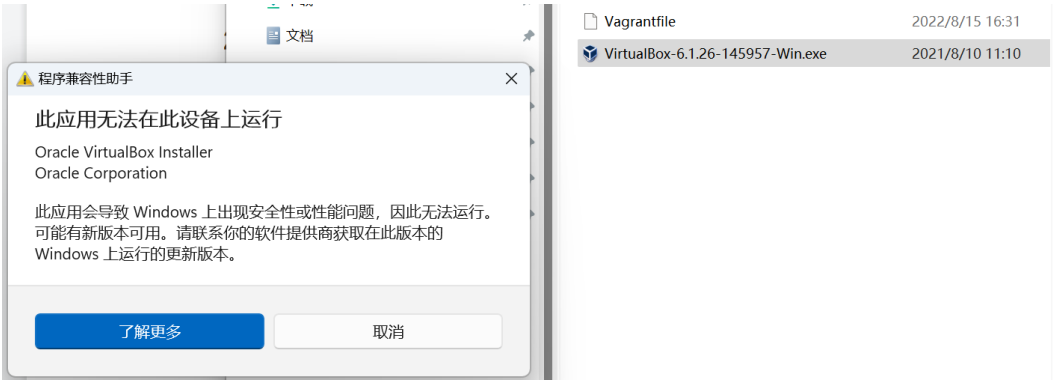

图 6-1 第一周问题 1

经查阅资料，在 Windows 设置中关闭“内存完整性”，重启设备即可继续安装。

2. 环境配置

在环境配置过程中，在添加 box 安装 vagrant 插件后试图运行 vagrant up 以开启虚拟机，但出现如下报错：

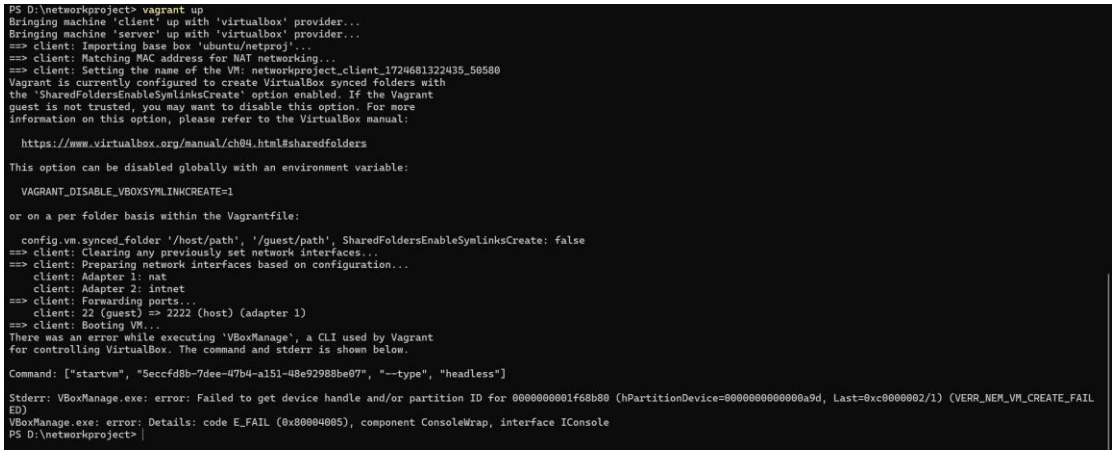

图 6-2 第一周问题 2

上网查找后发现没有相近问题的解决措施；幸运的是，经过尝试，用管理员身份开启终端，运行以下指令以关闭上学期计算机网络课程实践用到的 Hyper-v，重启电脑后，该问题成功解决。

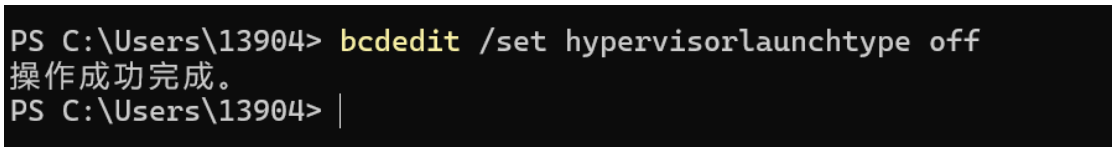

图 6-3 第一周问题 2 的解决方法

3. 环境错误

误修改 Vagrantfile 文件中 client 端和 server 端的 IP 地址后会报错如下：

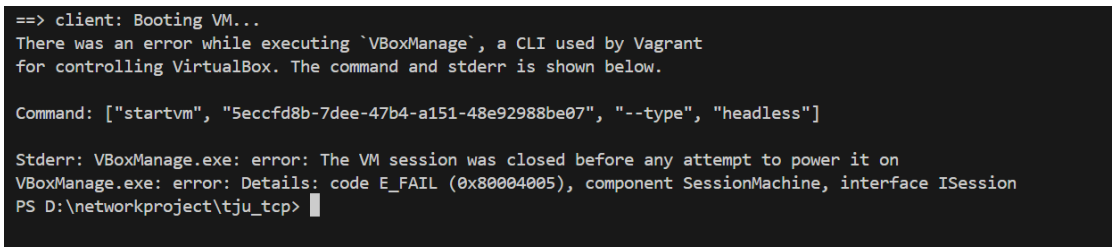

图 6-4 第一周问题 3 的解决方法

但修改回原来的地址后，重新尝试运行 vagrant up 仍报错。

解决：打开任务管理器，关闭 VirtualBox 相关进程，重新运行指令，成功修复。

同时，明白了本地自动测试包中设置的虚拟机 ip 为：客户端 172.17.0.2，服务端 172.17.0.3。所以在使用本地自动测试包前，需在项目目录下的 Vagrantfile 文件以及 tju_tcp/src 目录下的 kernel.c 文件中对 client 端和 server 端的 ip 地址进行修改。而基础框架其他文件中例如 client.c 和 server.c 中也有相关 ip 设置，不涉及本地自动测试。

4. 其他问题

```
src/tju_tcp.c: In function ‘tju_handle_packet’:
src/tju_tcp.c:287:9: error: a label can only be part of a statement and a declaration is
 not a statement
  287 |         char* packet_FIN_ACK4=create_packet_buf(get_dst(pkt),get_src(pkt),get_se
q(pkt),get_seq(pkt)+1,\
      |         ^~~~
make: *** [Makefile:19: build/tju_tcp.o] Error 1
Traceback (most recent call last):
  File "test.py", line 420, in <module>
  File "test.py", line 399, in main
  File "test.py", line 74, in compile_source_files
  File "<decorator-gen-3>", line 2, in run
  File "fabric/connection.py", line 30, in opens
  File "fabric/connection.py", line 723, in run
  File "invoke/context.py", line 102, in _run
  File "invoke/runners.py", line 380, in run
  File "invoke/runners.py", line 442, in _run_body
  File "invoke/runners.py", line 509, in _finish
invoke.exceptions.UnexpectedExit: Encountered a bad command exit code!

Command: 'cd /vagrant/tju_tcp && make'

Exit code: 2

Stdout: already printed

Stderr: already printed
```

图 6-5 第一周问题 4 的解决方法

原因是我在 case 之后进行变量的声明而没有将 case 下的语句以一个大括号包括。

分析：由于 switch 的几个 case 语句在同一个作用域（因为 case 语句只是标签，它们共属于一个 swtich 语句块），所以如果在某个 case 下面声明变量的话，对象的作用域是在俩个花括号之间，也就是整个 switch 语句，其他的 case 语句也能看到，这样就可能出现错误。

解决：我们可以在 case 后面的语句加上大括号，明确声明的变量的作用域是仅在本 case 之中。

5. 第一周遇到的问题主要是代码架构的分析与理解。这一周花了许多的时间梳理代码结构，理解任务背后的原理，但不清楚如何用代码实现。最后查阅了大量资料，复习 C 语言指针使用、与结构体等知识，学习了包括 getoftime、fprintf 等多个函数，一点点调试，尝试完成作业。

6.2 收获、体会及建议