实验3-1：基于UDP服务设计可靠传输协议并编程实现

2210977 陈恩宝 信息安全

receive.cpp

三次握手（Connect函数）--- 建立连接

```
int Connect(SOCKET& sockServ, SOCKADDR_IN& ClientAddr, int& ClientAddrLen)
{
    HEADER header;
    char* Buffer = new char[sizeof(header)];

    // 接收第一次握手信息
    while (1)
    {
        // 通过绑定的socket传递、接收数据
        if (recvfrom(sockServ, Buffer, sizeof(header), 0,
(sockaddr*)&ClientAddr, &ClientAddrLen) == -1)
        {
            return -1;
        }
        memcpy(&header, Buffer, sizeof(header));
        if (header.flag == SYN && checkSum((u_short*)&header, sizeof(header)) ==
0)
        {
            cout << " 接收到第一次握手数据 " << endl;
            break;
        }
    }
    // 发送第二次握手信息
    header.flag = ACK;
    header.sum = 0;
    header.sum = checkSum((u_short*)&header, sizeof(header));
    memcpy(Buffer, &header, sizeof(header));

    if (sendto(sockServ, Buffer, sizeof(header), 0, (sockaddr*)&ClientAddr,
ClientAddrLen) == -1)
    {
        return -1;
    }
    else
    {
        cout << " 成功发送第二次握手数据 " << endl;
    }
    clock_t start = clock(); // 记录第二次握手发送时间

    // 接收第三次握手
    while (recvfrom(sockServ, Buffer, sizeof(header), 0, (sockaddr*)&ClientAddr,
&ClientAddrLen) <= 0)
```

```
    {
        // 超时重传
        if (clock() - start > MAX_TIME)
        {
            cout << " 第二次握手超时 " << endl;
            header.flag = ACK;
            header.sum = 0;
            header.flag = checkSum((u_short*)&header, sizeof(header));
            memcpy(Buffer, &header, sizeof(header));
            if (sendto(sockServ, Buffer, sizeof(header), 0,
(sockaddr*)&ClientAddr, ClientAddrLen) == -1)
            {
                return -1;
            }
            cout << " 已经重传 " << endl;
        }
    }

    // 解析收到的第三次握手的数据包
    HEADER temp1;
    memcpy(&temp1, Buffer, sizeof(header));

    if (temp1.flag == ACK_SYN && checkSum((u_short*)&temp1, sizeof(temp1)) == 0)
    {
        cout << " 接收到第三次握手数据" << endl;
        cout << " 成功连接" << endl;
    }
    else
    {
        cout << " 错误数据，请重试" << endl;
    }
    return 1;
}
```

这段代码实现了 **三次握手** 协议的服务器端部分，它通过 UDP 来模拟连接的建立过程。通过这个过程，服务器与客户端可以确保双方在数据通信前已经准备好并能顺利交换数据。以下是对每一阶段的详细说明。

1. 三次握手协议概述

三次握手是为了在通信之前建立可靠的连接，确保双方都准备好发送和接收数据。具体的握手过程如下：

- **第一次握手**：客户端向服务器发送一个带有 `SYN` 标志的数据包，表示客户端希望建立连接。
- **第二次握手**：服务器收到 `SYN` 包后，回复一个带有 `SYN` 和 `ACK` 标志的数据包，表示同意建立连接。
- **第三次握手**：客户端收到 `SYN+ACK` 包后，再次发送一个带有 `ACK` 标志的数据包，表示连接建立。

2. 接收客户端的第一次握手信息（SYN）

```
while (1)
{
    // 通过绑定的socket传递、接收数据
    if (recvfrom(sockServ, Buffer, sizeof(header), 0, (sockaddr*)&ClientAddr,
&ClientAddrLen) == -1)
```

```
    {
        return -1;
    }
    memcpy(&header, Buffer, sizeof(header));
    if (header.flag == SYN && checkSum((u_short*)&header, sizeof(header)) == 0)
    {
        cout << " 接收到第一次握手数据 " << endl;
        break;
    }
}
```

- **等待接收第一次握手（SYN）**：服务器通过 `recvfrom` 等待接收来自客户端的第一个数据包。这个数据包的 `flag` 字段应该为 `SYN`，表示客户端请求建立连接。
- **校验包**：在接收到数据后，服务器首先校验包的 `flag` 是否为 `SYN`，并通过 `checkSum` 函数验证数据包的完整性。
- **成功接收到第一次握手**：如果数据包有效且 `flag == SYN`，则输出接收到数据的消息，并跳出循环进入下一步。

3. 发送第二次握手（SYN + ACK）

```
header.flag = ACK;
header.sum = 0;
header.sum = checkSum((u_short*)&header, sizeof(header));
memcpy(Buffer, &header, sizeof(header));

if (sendto(sockServ, Buffer, sizeof(header), 0, (sockaddr*)&ClientAddr,
ClientAddrLen) == -1)
{
    return -1;
}
else
{
    cout << " 成功发送第二次握手数据 " << endl;
}
```

- **构造第二次握手数据包**：服务器接收到第一次握手（SYN）后，准备发送第二次握手响应包。这个包的 `flag` 字段设置为 `ACK`，表示服务器同意建立连接。
- **校验和计算**：在构造数据包后，重新计算包的校验和，确保数据完整。
- **发送数据包**：通过 `sendto` 函数将第二次握手数据包发送给客户端。
- **日志输出**：如果发送成功，打印成功发送第二次握手的数据包的日志。

4. 接收第三次握手信息（SYN + ACK）

```
clock_t start = clock(); // 记录第二次握手发送时间

while (recvfrom(sockServ, Buffer, sizeof(header), 0, (sockaddr*)&ClientAddr,
&ClientAddrLen) <= 0)
{
    // 超时重传
    if (clock() - start > MAX_TIME)
    {
        cout << " 第二次握手超时 " << endl;
        header.flag = ACK;
```

```
        header.sum = 0;
        header.flag = checkSum((u_short*)&header, sizeof(header));
        memcpy(Buffer, &header, sizeof(header));
        if (sendto(sockServ, Buffer, sizeof(header), 0, (sockaddr*)&ClientAddr,
ClientAddrLen) == -1)
        {
            return -1;
        }
        cout << " 已经重传 " << endl;
    }
}
```

- **等待第三次握手**：服务器需要等待客户端响应第三次握手数据包。第三次握手的数据包通常包含 `SYN` 和 `ACK` 标志。服务器通过 `recvfrom` 函数接收数据，并检查是否收到有效数据。
- **超时重传**：如果在规定时间内没有收到第三次握手包（`MAX_TIME`），服务器会重传第二次握手包。这是通过 `sendto` 重发第二次握手数据包实现的。
- **超时判断**：通过 `clock()` 函数记录发送时间，如果超过设定的 `MAX_TIME` 时间，则触发超时机制，重新发送第二次握手。

5. 解析收到的第三次握手包，确认连接建立

```
HEADER temp1;
memcpy(&temp1, Buffer, sizeof(header));

if (temp1.flag == ACK_SYN && checkSum((u_short*)&temp1, sizeof(temp1)) == 0)
{
    cout << " 接收到第三次握手数据" << endl;
    cout << " 成功连接" << endl;
}
else
{
    cout << " 错误数据，请重试" << endl;
}
```

- **解析第三次握手数据包**：接收到数据包后，服务器首先解析包内容，并验证 `flag` 是否为 `ACK_SYN`，表示客户端已经收到服务器的第二次握手包，并确认连接。
- **校验数据包**：再次验证数据包的校验和，确保包没有被损坏。
- **连接成功**：如果数据包有效并且 `flag` 为 `ACK_SYN`，服务器输出“成功连接”，表示三次握手成功，连接建立。

6. 结束三次握手

如果数据包无误且符合要求，三次握手完成，双方可以开始数据通信。

在整个过程中，服务器通过验证包的 `flag` 字段和校验和来确保数据的可靠性，并使用超时重传机制来确保数据包能够成功到达对方。

计算16位校验和（checkSum函数）--- 差错检验

```
u_short checkSum(u_short* mes, int size)
{
    int count = (size + 1) / 2;
    // buffer相当于一个元素为u_short类型的数组，每个元素16位，相当于求校验和过程中的一个元素
```

```
    u_short* buf = (u_short*)malloc(size + 1);
    memset(buf, 0, size + 1);
    memcpy(buf, mes, size); // 将message读入buf
    u_long sum = 0;
    while (count--)
    {
        sum += *buf++;
        // 如果有进位则将进位加到最低位
        if (sum & 0xffff0000)
        {
            sum &= 0xffff;
            sum++;
        }
    }
    // 取反
    return ~(sum & 0xffff);
}
```

```
 int count = (size + 1) / 2;
```

将消息的字节大小 `size` 加 1 后，除以 2 来计算需要迭代的次数。这里加 是为了处理奇数个字节的情况，确保所有字节都被正确处理。

```
    u_short* buf = (u_short*)malloc(size + 1);
```

分配一个足够大的内存空间（`size + 1` 字节）。因为我们可能需要处理一个额外的字节（如果消息大小是奇数），为了确保没有越界，申请了 `size + 1` 字节的空间。

```
    memset(buf, 0, size + 1);
```

将分配的内存块 `buf` 初始化为 0，确保没有残留的数据干扰。

```
    memcpy(buf, mes, size); // 将message读入buf
```

将输入消息 `mes` 的内容拷贝到缓冲区 `buf` 中。这确保了接下来的处理不会修改原始数据。

```
    u_long sum = 0;
```

声明一个 `u_long` 类型的变量 `sum` 来存储累加的和。 `u_long` 是 32 位的，保证足够容纳累加后的值。

```
    while (count--)
```

`count--` 表示每次循环时会减少 `count` 的值，直到为 0 为止。每次循环将处理两个字节（一个 `u_short`）。

```
    {
        sum += *buf++;
```

将当前 `buf` 指向的 `u_short` 值加到 `sum` 上，然后将 `buf` 指针向后移动，指向下一个 `u_short` 数据。

```
if (sum & 0xffff0000)
```

检查是否有溢出的 16 位（即 sum的高 16 位是否有值）。如果有溢出：

- `sum &= 0xffff`：保留 `sum` 的低 16 位（丢弃高 16 位）。
- `sum++`：加 1，将溢出的部分加回 `sum` 中，保证校验和的正确性。

```
    return ~(sum & 0xffff);
```

`sum` 中存储的是计算得到的校验和。`sum & 0xffff` 保证只取 `sum` 的低 16 位。`~` 运算符对结果取反，得到最终的校验和并返回。

四次挥手（disconnect函数）--- 断开连接

四次挥手概述

四次挥手是为了确保连接的双方都能正确地关闭连接。其过程包括四个阶段：

1. **第一次挥手**：客户端向服务器发送一个带有 `FIN` 标志的数据包，表示客户端希望关闭连接。
2. **第二次挥手**：服务器收到 `FIN` 包后，回复一个带有 `ACK` 标志的数据包，表示服务器同意关闭连接。
3. **第三次挥手**：服务器主动关闭连接时，向客户端发送一个带有 `FIN` 标志的数据包，表示服务器请求关闭连接。
4. **第四次挥手**：客户端收到服务器的 `FIN` 包后，回复一个带有 `ACK` 标志的数据包，表示客户端同意关闭连接，连接结束。

1. 接收客户端的第一次挥手（FIN）

```
while (1)
{
    int length = recvfrom(sockServ, Buffer, sizeof(header) + MAXSIZE, 0,
(sockaddr*)&ClientAddr, &ClientAddrLen);
    memcpy(&header, Buffer, sizeof(header));
    if (header.flag == FIN && checkSum((u_short*)&header, sizeof(header)) == 0)
    {
        cout << " 接收到第一次挥手数据 " << endl;
        break;
    }
    else
    {
        return -1;
    }
}
```

- **等待接收客户端的 `FIN` 包**：服务器通过 `recvfrom` 接收来自客户端的数据包。客户端通过 `FIN` 包请求断开连接。
- **校验包的合法性**：接收到数据包后，服务器首先检查包的 `flag` 字段是否为 `FIN`，并通过 `checkSum` 校验包的完整性。
- **成功接收到第一次挥手数据**：如果数据包有效且 `flag == FIN`，则表示客户端请求断开连接，服务器打印日志并跳出循环，准备发送第二次挥手数据。

2. 发送第二次挥手（ACK）

```
header.flag = ACK;
header.sum = 0;
header.sum = checkSum((u_short*)&header, sizeof(header));
memcpy(Buffer, &header, sizeof(header));
if (sendto(sockServ, Buffer, sizeof(header), 0, (sockaddr*)&ClientAddr,
ClientAddrLen) == -1)
{
    cout << " 发送第二次挥手失败" << endl;
    return -1;
}
else
{
    cout << " 成功发送第二次挥手数据" << endl;
}
```

- **构造第二次挥手数据包**：服务器收到客户端的 `FIN` 包后，构造一个带有 `ACK` 标志的数据包，表示服务器同意关闭连接。
- **校验和计算**：重新计算数据包的校验和，并将其存储在包中。
- **发送数据包**：通过 `sendto` 函数发送带有 `ACK` 标志的包给客户端。
- **日志输出**：如果数据包发送成功，打印"成功发送第二次挥手数据"的日志。

3. 发送第三次挥手（FIN_ACK）

```
header.flag = FIN_ACK;
header.sum = 0;
header.sum = checkSum((u_short*)&header, sizeof(header)); // 计算校验和
if (sendto(sockServ, (char*)&header, sizeof(header), 0, (sockaddr*)&ClientAddr,
ClientAddrLen) == -1)
{
    return -1;
}
else
{
    cout << " 成功发送第三次挥手数据" << endl;
}
```

- **构造第三次挥手数据包**：服务器向客户端发送一个带有 `FIN_ACK` 标志的数据包，表示服务器请求断开连接。`FIN_ACK` 标志是 `FIN` 和 `ACK` 的组合。
- **计算校验和**：重新计算包的校验和，确保数据的完整性。
- **发送数据包**：通过 `sendto` 函数发送第三次挥手数据包。
- **日志输出**：如果数据包发送成功，打印"成功发送第三次挥手数据"的日志。

4. 接收客户端的第四次挥手（ACK）

```
clock_t start = clock();
while (recvfrom(sockServ, Buffer, sizeof(header), 0, (sockaddr*)&ClientAddr,
&ClientAddrLen) <= 0)
{
    if (clock() - start > MAX_TIME) // 超时，重新传输第三次挥手
    {
        cout << " 第三次挥手超时" << endl;
        header.flag = FIN;
        header.sum = 0;                                                    // 校验和置0
        header.sum = checkSum((u_short*)&header, sizeof(header)); // 计算校验和
```

```
        memcpy(Buffer, &header, sizeof(header));                    // 将首部放入缓
冲区
        sendto(sockServ, Buffer, sizeof(header), 0, (sockaddr*)&ClientAddr,
ClientAddrLen);
        start = clock();
        cout << " 已重传第三次挥手数据" << endl;
    }
}
```

- **等待第四次挥手数据包（ACK）**：服务器等待接收客户端的第四次挥手包，即客户端响应服务器的 `FIN_ACK` 包。这表明客户端确认断开连接。
- **超时重传**：如果在设定的超时时间内（`MAX_TIME`）未收到第四次挥手包，则认为发生了超时，服务器会重新发送第三次挥手数据包。超时重传是为了确保客户端最终能够收到第三次挥手包并响应。
- **校验和重新计算**：如果需要重传，重新计算校验和并发送数据包。
- **日志输出**：如果发生超时，打印重传日志，并重发第三次挥手数据包。

5. 四次挥手完成，连接断开

```
cout << " 四次挥手结束，连接断开！" << endl;
```

- **连接关闭**：如果成功接收到客户端的第四次挥手包，或者在重传过程中最终收到第四次挥手包，表示连接关闭。服务器打印“四次挥手结束，连接断开”的日志，标志着连接的完全断开。

超时重传

涉及超时重传的部分主要出现在 **三次握手** 和 **四次挥手** 过程中的超时检测与重传机制。超时重传是为了确保在网络条件不稳定或丢包的情况下，数据包能够成功地到达对方，并且协议能够顺利完成。

超时重传的主要部分

1. **三次握手过程中的重传**：
    - 在三次握手的第二步，服务器需要发送 `ACK` 包来确认接收到客户端的 `SYN` 包。接着，服务器等待接收客户端发来的 `ACK_SYN` 包，这个过程涉及超时重传。
2. **四次挥手过程中的重传**：
    - 在四次挥手的第三步，服务器需要发送 `FIN_ACK` 包来告诉客户端关闭连接。接着，服务器等待接收客户端的 `ACK` 包来确认关闭连接。这个过程同样涉及超时重传。

1. 三次握手过程中的重传

```
clock_t start = clock(); // 记录第二次握手发送时间

// 接收第三次握手
while (recvfrom(sockServ, Buffer, sizeof(header), 0, (sockaddr*)&ClientAddr,
&ClientAddrLen) <= 0)
{
    // 超时重传
    if (clock() - start > MAX_TIME)
    {
        cout << " 第二次握手超时 " << endl;
        header.flag = ACK;
        header.sum = 0;
        header.flag = checkSum((u_short*)&header, sizeof(header));
        memcpy(Buffer, &header, sizeof(header));
```

```
        if (sendto(sockServ, Buffer, sizeof(header), 0, (sockaddr*)&ClientAddr,
ClientAddrLen) == -1)
        {
            return -1;
        }
        cout << " 已经重传 " << endl;
    }
}
```

解释：

- 在建立连接时，客户端会发送带有 `SYN` 标志的数据包，服务器收到后回复 `ACK` 包。接着，客户端发送 `ACK_SYN` 包来完成三次握手。
- 服务器使用 `clock()` 记录当前时间，在等待客户端的 `ACK_SYN` 包时，如果超过了设定的最大超时时间（`MAX_TIME`），服务器会重新发送第二次握手的 `ACK` 包。
- `MAX_TIME` 设置为 0.5 秒，即如果超过 0.5 秒未收到第三次握手的响应，服务器会进行重传。
- 重传机制通过 `sendto` 函数实现，将之前构造的 `ACK` 包重新发送给客户端。
- 重传成功后，打印“已经重传”。

关键点：

- **超时检测**：`clock() - start > MAX_TIME` 判断是否超时。
- **重传机制**：在超时的情况下，重新发送第二次握手的 `ACK` 包。

---

2. 四次挥手过程中的重传

```
clock_t start = clock();
while (recvfrom(sockServ, Buffer, sizeof(header), 0, (sockaddr*)&ClientAddr,
&ClientAddrLen) <= 0)
{
    if (clock() - start > MAX_TIME) // 超时，重新传输第三次挥手
    {
        cout << " 第三次挥手超时" << endl;
        header.flag = FIN;
        header.sum = 0;                                          // 校验和置0
        header.sum = checkSum((u_short*)&header, sizeof(header)); // 计算校验和
        memcpy(Buffer, &header, sizeof(header));                 // 将首部放入缓
冲区
        sendto(sockServ, Buffer, sizeof(header), 0, (sockaddr*)&ClientAddr,
ClientAddrLen);
        start = clock();
        cout << " 已重传第三次挥手数据" << endl;
    }
}
```

解释：

- 在四次挥手过程中，服务器会发送 `FIN_ACK` 包，告知客户端关闭连接。然后，服务器等待客户端的 `ACK` 包来确认连接关闭。
- 如果在设定的最大超时时间 `MAX_TIME` 内（0.5秒）没有收到客户端的确认包，服务器会认为连接关闭请求没有得到确认，因此进行重传。
- 在重传时，服务器会重新发送第三次挥手的 `FIN_ACK` 包，同时重新计算校验和。
- 重传后，服务器会继续等待客户端的确认包（第四次挥手）。

关键点：

- **超时检测**：同样使用 `clock() - start > MAX_TIME` 判断是否超时。
- **重传机制**：在超时的情况下，重新发送第三次挥手的 `FIN_ACK` 包。

重传流程的整体分析

- **超时检测**：重传机制的核心是时间的检测，通过记录 `clock()` 时间来计算是否超时。如果超时，执行重传。
- **重传条件**：`clock() - start > MAX_TIME` 用来检查是否超过设定的最大等待时间。如果超时，则触发重传。
- **重传动作**：当超时发生时，通过 `sendto()` 函数重新发送上一次发送的 `ACK` 或 `FIN_ACK` 包。
- **重传次数**：在此代码中，重传只会在超时后执行一次，即在超时后立即重传。如果仍然没有收到回应，服务器会继续等待直到下一次超时或连接失败。

send.cpp

三次握手（Connect函数）--- 建立连接，超时重传

该函数实现了客户端通过 **UDP** 协议与服务器进行 **三次握手** 建立连接的过程，具体功能是让客户端与服务器之间建立一个可靠的通信连接，模拟三次握手过程。下面是对代码的详细介绍，逐步分析每一部分的操作。

代码解析

```
int Connect(SOCKET& socketClient, SOCKADDR_IN& servAddr, int& servAddrlen)
{
    HEADER header;
    char* Buffer = new char[sizeof(header)];
```

首先，定义了一个 `HEADER` 结构体和一个缓冲区 `Buffer`，缓冲区大小为 `header` 结构体的大小。`HEADER` 结构体用于存储每次握手的控制信息，包括校验和、数据长度、标志位和序列号等。

第一次握手

```
// 第一次握手
    header.flag = SYN;
    header.sum = 0; // 校验和置0
    // 计算校验和
    header.sum = checkSum((u_short*)&header, sizeof(header));
    // 将数据头放入buffer
    memcpy(Buffer, &header, sizeof(header));
    if (sendto(socketClient, Buffer, sizeof(header), 0, (sockaddr*)&servAddr,
servAddrlen) == -1)
    {
        return -1;
    }
    else
    {
        cout << "[\033[1;31mSend\033[0m] 成功发送第一次握手数据" << endl;
    }
```

1. **设置标志位**：在第一次握手时，客户端发送一个带有 `SYN` 标志位的数据包来向服务器发起连接请求。
    - `header.flag = SYN;`：设置 `flag` 为 `SYN`（即，`SYN` 位为 1，表示这是连接请求包）。
2. **计算校验和**：校验和在每次发送数据时都需要计算，目的是确保数据在传输过程中没有错误。
    - `header.sum = checkSum((u_short*)&header, sizeof(header));`：计算校验和并赋值给 `header.sum`。
3. **发送数据**：客户端使用 `sendto` 函数将 `header` 数据包发送到服务器。
    - `sendto(socketClient, Buffer, sizeof(header), 0, (sockaddr*)&servAddr, servAddrlen)` 发送包，`Buffer` 中存储了第一次握手的数据包。
4. **日志输出**：如果发送成功，输出日志提示发送了第一次握手数据。

设置为非阻塞模式

```
clock_t start = clock(); // 记录发送第一次握手时间

    // 为了函数能够继续运行，设置socket为非阻塞状态
    u_long mode = 1;
    ioctlsocket(socketClient, FIONBIO, &mode);
```

1. **开始计时**：记录发送第一次握手数据包的时间，用于超时检测。
2. **非阻塞模式**：调用 `ioctlsocket` 将套接字设置为 **非阻塞模式**，确保 `recvfrom` 函数不会因为等待接收数据而阻塞。非阻塞模式下，`recvfrom` 如果没有数据可读，会立即返回。

第二次握手（接收并验证服务器的 ACK+超时重传）

```
 // 第二次握手
    while (recvfrom(socketClient, Buffer, sizeof(header), 0,
(sockaddr*)&servAddr, &servAddrlen) <= 0)
    {
        // 超时需要重传
        if (clock() - start > MAX_TIME) // 超时，重新传输第一次握手
        {
            cout << "[\033[1;33mInfo\033[0m] 第一次握手超时" << endl;
            header.flag = SYN;
            header.sum = 0;                                                  // 校验和
置0
            header.sum = checkSum((u_short*)&header, sizeof(header)); // 计算校验
和
            memcpy(Buffer, &header, sizeof(header));                    // 将数据
头放入Buffer
            sendto(socketClient, Buffer, sizeof(header), 0,
(sockaddr*)&servAddr, servAddrlen);
            start = clock();
            cout << "[\033[1;33mInfo\033[0m] 已经重传" << endl;
        }
    }
```

1. **等待接收第二次握手数据**：`recvfrom` 等待接收来自服务器的响应。在此步骤，客户端会接收到带有 `ACK` 标志的数据包，表示服务器确认了客户端的连接请求。
2. **超时重传机制**：

- 如果在设定的超时时间 `MAX_TIME` 内（0.5秒）没有收到响应，客户端会重新发送第一次握手数据。
- `clock() - start > MAX_TIME`：检查是否超过了最大等待时间。
- 如果超时，重传第一次握手数据包，并重新开始计时。此时输出“第一次握手超时”以及“已经重传”。

第二次握手（接收到服务器的 ACK）

```
// 第二次握手，收到来自接收端的ACK
    // 进行校验和检验
    memcpy(&header, Buffer, sizeof(header));
    if (header.flag == ACK && checkSum((u_short*)&header, sizeof(header)) == 0)
    {
        cout << "[\033[1;32mReceive\033[0m] 接收到第二次握手数据" << endl;
    }
    else
    {
        cout << "[\033[1;33mInfo\033[0m] 错误数据，请重试" << endl;
        return -1;
    }
```

1. **校验和检验**：收到的数据包进行校验和验证，以确保数据包没有在传输过程中损坏。如果数据包的校验和正确且 `flag` 为 `ACK`，则表示服务器成功接收到客户端的连接请求，并准备好进行数据传输。
2. **日志输出**：如果接收到正确的 `ACK` 包，打印日志“接收到第二次握手数据”。如果校验失败或收到不正确的数据包，输出“错误数据，请重试”，并返回 `-1` 结束连接过程。

第三次握手（发送 ACK_SYN 给服务器）

```
cpp复制代码    // 进行第三次握手
    header.flag = ACK_SYN;
    header.sum = 0;
    header.sum = checkSum((u_short*)&header, sizeof(header)); // 计算校验和
    if (sendto(socketClient, (char*)&header, sizeof(header), 0,
(sockaddr*)&servAddr, servAddrlen) == -1)
    {
        return -1;
    }
    else
    {
        cout << "[\033[1;31mSend\033[0m] 成功发送第三次握手数据" << endl;
    }
    cout << "[\033[1;33mInfo\033[0m] 服务器成功连接！可以发送数据" << endl;
    return 1;
}
```

1. **第三次握手**：此时，客户端向服务器发送 `ACK_SYN` 标志的数据包，表示连接的确认。
   - `header.flag = ACK_SYN;`：设置标志为 `ACK_SYN`，表示第三次握手。
2. **计算校验和**：在发送前需要重新计算校验和。
3. **发送数据**：通过 `sendto` 函数将构造好的 `ACK_SYN` 数据包发送给服务器，完成第三次握手。

4. **日志输出**：如果成功发送第三次握手数据，打印日志“成功发送第三次握手数据”。之后，输出“服务器成功连接！可以发送数据”，表示连接已经建立，可以开始进行数据交换。

计算16位校验和（checkSum函数）--- 差错检验

同receive.cpp文件

四次挥手（disconnect函数）--- 断开连接

第一次挥手（客户端发起）

```
HEADER header;
char* Buffer = new char[sizeof(header)];
u_short sum;
```

- 创建一个 `HEADER` 类型的结构体 `header`，它用来存储协议头部信息（`flag`，`sum` 等字段）。
- `Buffer` 是一个字符数组，它的大小与 `header` 相同，用来存储要发送的数据。
- `sum` 用来存储校验和，通常是用于校验数据的完整性。

```
// 进行第一次挥手
header.flag = FIN;
header.sum = 0;                                                  // 校验和置0
header.sum = checkSum((u_short*)&header, sizeof(header)); // 计算校验和
memcpy(Buffer, &header, sizeof(header));                    // 将首部放入缓冲区
```

- 第一次挥手

  ：客户端发送一个带有

  ```
  FIN
  ```

  标志的数据包，表示它请求关闭连接。

  - `header.flag = FIN`：`FIN` 标志表示关闭连接。
  - `header.sum = 0`：校验和初始化为 0。
  - `header.sum = checkSum((u_short*)&header, sizeof(header))`：计算校验和。校验和是通过对头部的每个字节进行某种算法计算得到的，用于确保数据完整性。
  - `memcpy(Buffer, &header, sizeof(header))`：将 `header` 复制到缓冲区 `Buffer` 中，准备发送。

```
if (sendto(socketClient, Buffer, sizeof(header), 0, (sockaddr*)&servAddr,
servAddrlen) == -1)
{
    return -1;
}
else
{
    cout << "[\033[1;31mSend\033[0m] 成功发送第一次挥手数据" << endl;
}
```

- 使用 `sendto` 函数将数据包发送给服务器，发送的是包含 `FIN` 标志的请求包。
- 如果发送失败，返回 -1，表示错误。
- 如果发送成功，输出提示信息，表示第一次挥手数据已发送。

超时重传处理（第一次挥手）

```
clock_t start = clock(); // 记录发送第一次挥手时间
u_long mode = 1;
ioctlsocket(socketClient, FIONBIO, &mode); // FIONBIO为命令，允许1/禁止0套接口的非阻塞1/阻塞0模式。
```

- `start = clock()`：记录当前时间，用于计算超时。
- 设置套接字为非阻塞模式，`ioctlsocket(socketClient, FIONBIO, &mode)`，使得 `recvfrom` 函数不会阻塞，允许继续执行而不等待数据。

```
cpp复制代码// 接收第二次挥手
while (recvfrom(socketClient, Buffer, sizeof(header), 0, (sockaddr*)&servAddr, &servAddrlen) <= 0)
{
    // 超时，重新传输第一次挥手
    if (clock() - start > MAX_TIME)
    {
        cout << "[\033[1;33mInfo\033[0m] 第一次挥手超时" << endl;
        header.flag = FIN;
        header.sum = 0;                                          // 校验和置0
        header.sum = checkSum((u_short*)&header, sizeof(header)); // 计算校验和
        memcpy(Buffer, &header, sizeof(header));                 // 将首部放入缓冲区
        sendto(socketClient, Buffer, sizeof(header), 0, (sockaddr*)&servAddr, servAddrlen);
        start = clock();
        cout << "[\033[1;33mInfo\033[0m] 已重传第一次挥手数据" << endl;
    }
}
```

- **等待第二次挥手**：使用 `recvfrom` 等待从服务器接收数据包。由于套接字处于非阻塞模式，`recvfrom` 会立即返回。
- 超时处理
  ：如果超过 MAX_TIME超时时间没有收到响应（即接收失败），则重传第一次挥手的 FIN 数据包。
  - 清空并重新计算校验和，使用 `sendto` 重发数据包。
  - 重设 `start = clock()`，重新记录时间。

第二次挥手（客户端收到服务器的 `ACK` 包）

```
// 进行校验和检验
memcpy(&header, Buffer, sizeof(header));
if (header.flag == ACK && checkSum((u_short*)&header, sizeof(header)) == 0)
{
    cout << "[\033[1;32mReceive\033[0m] 接收到第二次挥手数据" << endl;
}
else
{
    cout << "[\033[1;33mInfo\033[0m] 错误数据，请重试" << endl;
    return -1;
}
```

- **检查第二次挥手包**：接收服务器返回的 `ACK` 包，表示服务器确认客户端的 `FIN` 包。
- 校验数据包的标志位是否为 `ACK`，并校验和是否正确。如果有错误，则输出错误提示并返回 -1。

第三次挥手（客户端发送 `ACK`）

```
// 进行第三次挥手
start = clock(); // 记录第二次挥手发送时间

// 接收第三次挥手
while (recvfrom(socketClient, Buffer, sizeof(header), 0, (sockaddr*)&servAddr,
&servAddrlen) <= 0)
{
    // 发送第二次挥手等待第三次挥手过程中超时，重传第二次挥手
    if (clock() - start > MAX_TIME)
    {
        cout << "[\033[1;33mInfo\033[0m] 第二次挥手超时 " << endl;
        header.flag = ACK;
        header.sum = 0;
        header.sum = checkSum((u_short*)&header, sizeof(header));
        memcpy(Buffer, &header, sizeof(header));
        if (sendto(socketClient, Buffer, sizeof(header), 0,
(sockaddr*)&servAddr, servAddrlen) == -1)
        {
            return -1;
        }
        cout << "[\033[1;33mInfo\033[0m] 已重传第二次挥手数据 " << endl;
    }
}
```

- **接收第三次挥手包**：客户端等待服务器发送的 `FIN_ACK` 包，表示服务器已经准备好关闭连接。
- **超时重传**：如果没有收到数据包，超过 `MAX_TIME` 超时限制，客户端将重传第二次挥手（`ACK` 包）。

第四次挥手（确认关闭）

```
HEADER temp1;
memcpy(&temp1, Buffer, sizeof(header));
if (temp1.flag == FIN_ACK && checkSum((u_short*)&temp1, sizeof(temp1)) == 0)
{
    cout << "[\033[1;32mReceive\033[0m] 接收到第三次挥手数据 " << endl;
}
else
{
    cout << "[\033[1;33mInfo\033[0m] 错误数据，请重试" << endl;
    return -1;
}
```

- **校验第三次挥手**：收到服务器的 `FIN_ACK` 包，客户端验证该数据包的标志位和校验和，确保数据包没有损坏。

```
// 发送第四次挥手信息
header.flag = FIN_ACK;
header.sum = 0;
header.sum = checkSum((u_short*)&header, sizeof(header));
memcpy(Buffer, &header, sizeof(header));
if (sendto(socketClient, Buffer, sizeof(header), 0, (sockaddr*)&servAddr,
servAddrlen) == -1)
{
    cout << "[\033[1;33mInfo\033[0m] 第四次挥手发送失败 " << endl;
    return -1;
}
else
{
    cout << "[\033[1;31mSend\033[0m] 成功发送第四次挥手数据 " << endl;
}
```

- **发送第四次挥手**：客户端发送最后的 `FIN_ACK` 包，表示客户端已经完成断开连接的所有步骤。
- 重置校验和，构造 `FIN_ACK` 数据包并通过 `sendto` 发送给服务器。

```
cout << "[\033[1;33mInfo\033[0m] 四次挥手结束，连接断开！ " << endl;
```

- 输出信息表示四次挥手过程结束，连接成功断开。

计算传输时间和吞吐率

1. 传输时间 (Transmission Time)

传输时间通常指的是从开始发送数据到接收到最后的确认消息所经过的时间。在这段代码中，我已经通过 `clock()` 函数记录了传输过程中的时间。

计算过程：

- 在 `main` 函数中，我使用了 `clock_t start1 = clock();` 和 `clock_t end1 = clock();` 来记录文件传输的开始和结束时间。
- `start1` 是文件发送开始的时间点，`end1` 是文件传输完成后的时间点。

计算传输时间的代码如下：

```
clock_t start1 = clock();  // 记录传输开始的时间
send(server, serverAddr, len, buffer, i);  // 调用传输文件的函数
clock_t end1 = clock();  // 记录传输结束的时间

// 计算传输时间，单位为秒
double transmissionTime = (double)(end1 - start1) / CLOCKS_PER_SEC;
cout << "[\033[1;36mOut\033[0m] 传输总时间为: " << transmissionTime << "s" <<
endl;
```

这里的 `CLOCKS_PER_SEC` 是标准的常量，它表示每秒钟的时钟滴答数，通常是 1000 或 1000000。通过 `end1 - start1` 可以获得传输过程的时钟滴答数，除以 `CLOCKS_PER_SEC` 得到以秒为单位的传输时间。

2. 吞吐率 (Throughput)

吞吐率是单位时间内成功传输的数据量，通常以字节/秒（byte/s）为单位。在这个代码中，吞吐率的计算方式是：

计算过程：

- 数据大小：`i` 表示文件内容的字节数。
- 传输时间：`transmissionTime` 已经计算得到。

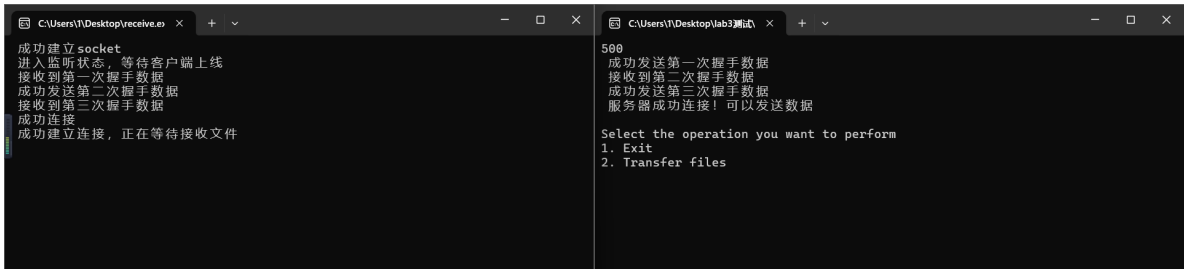
计算吞吐率的代码如下：

```
// 计算吞吐率，单位为字节每秒
double throughput = (double)i / transmissionTime;
cout << "[\033[1;36mOut\033[0m] 吞吐率为: " << fixed << setprecision(2) <<
throughput << " byte/s" << endl;
```

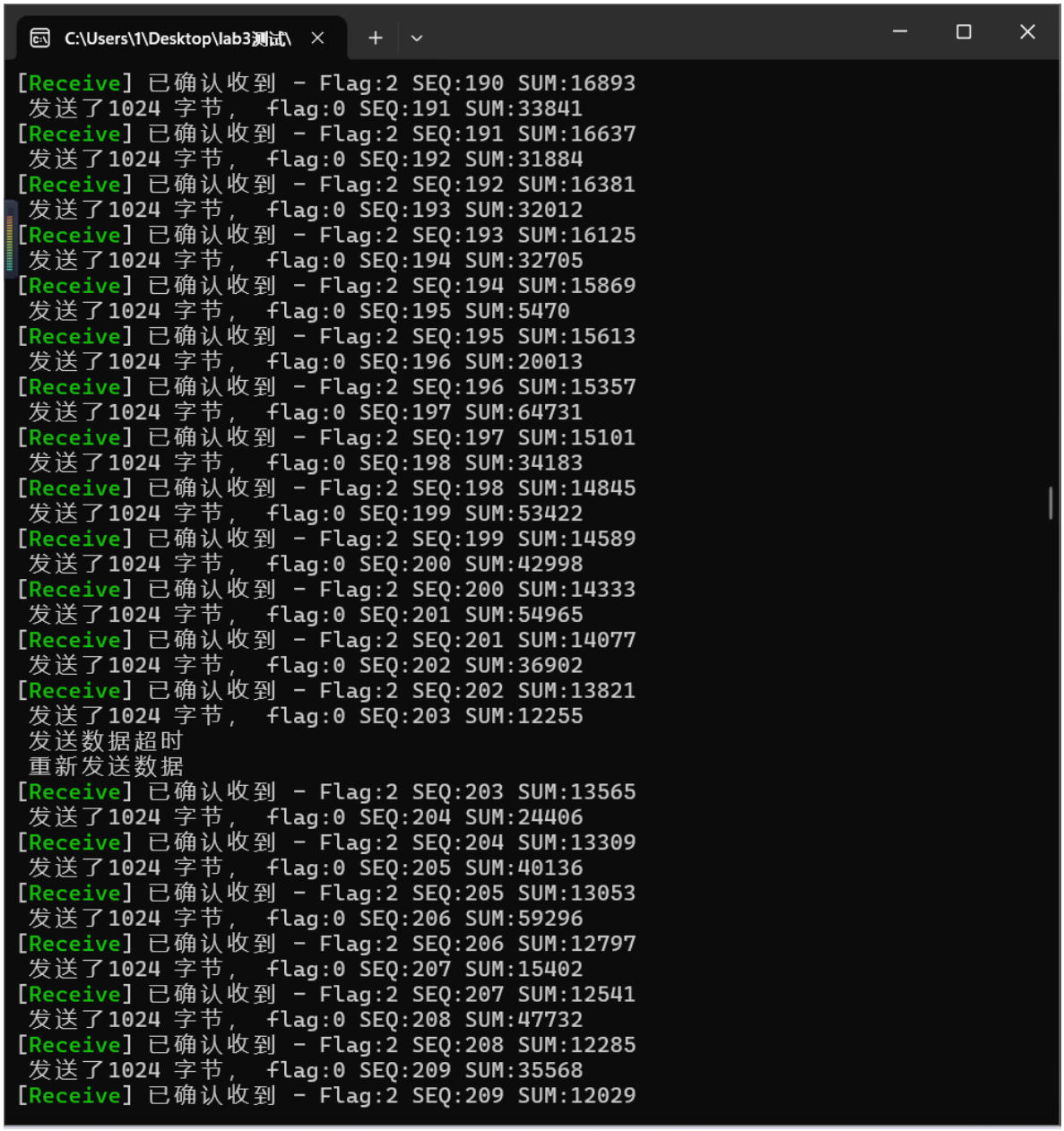
这里的 `i` 是文件的字节数，`transmissionTime` 是传输文件所需要的时间（单位：秒）。将字节数除以传输时间，就可以得到吞吐率。

实验结果

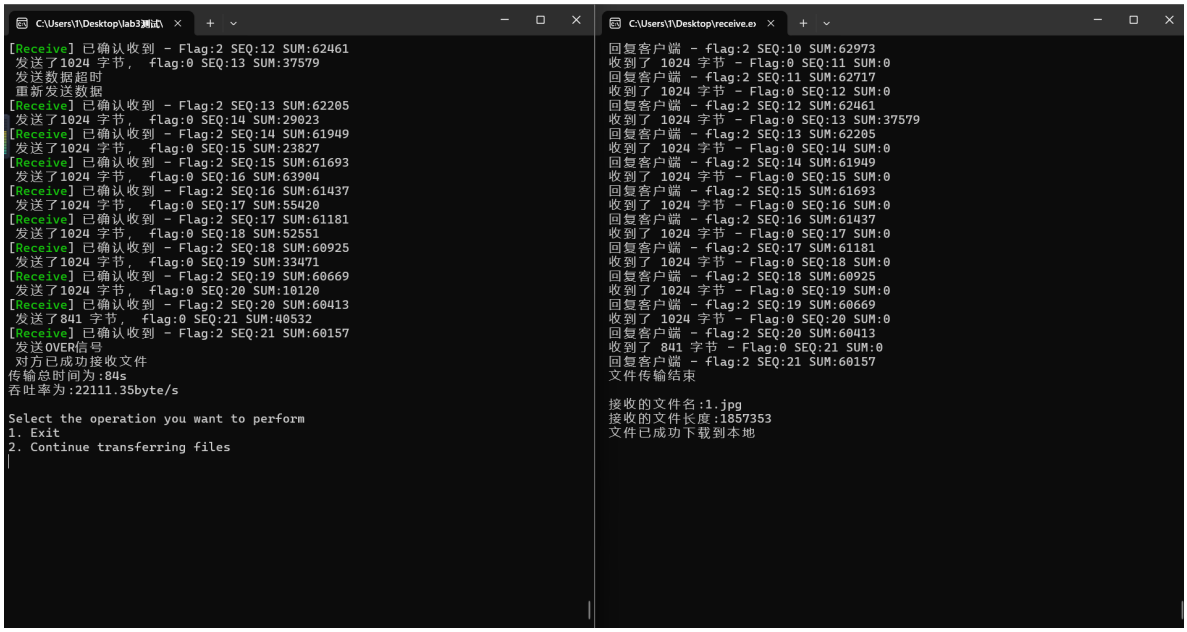
建立连接



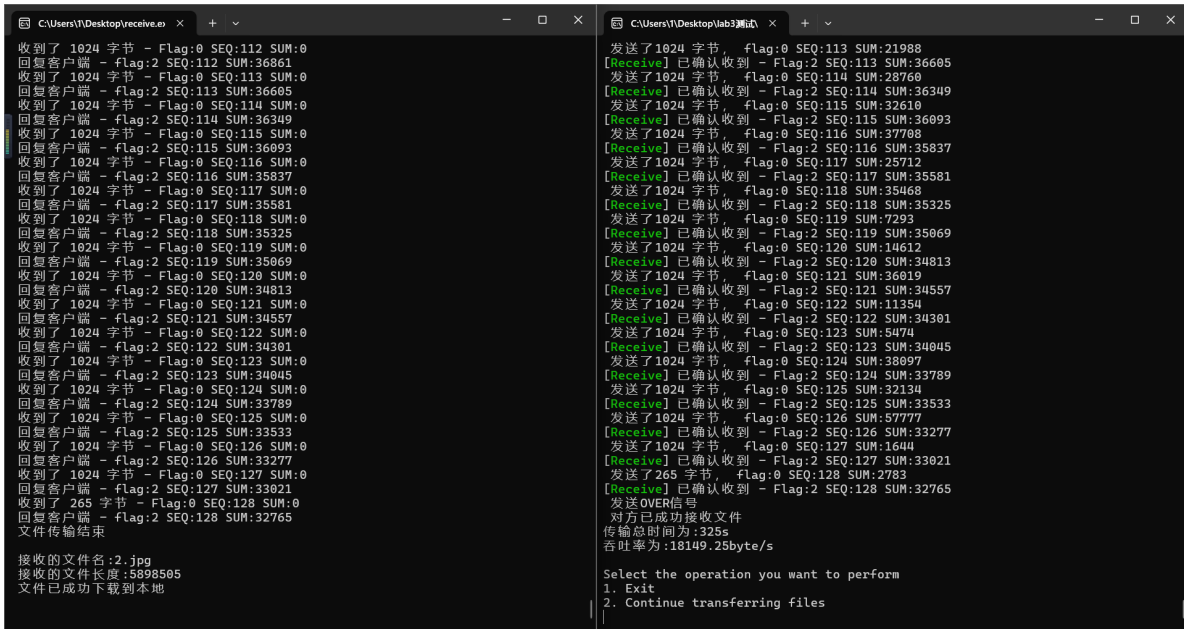
超时重传


C:\Users\1\Desktop\lab3测试\
[Receive] 已确认收到 - Flag:2 SEQ:190 SUM:16893
发送了1024 字节， flag:0 SEQ:191 SUM:33841
[Receive] 已确认收到 - Flag:2 SEQ:191 SUM:16637
发送了1024 字节， flag:0 SEQ:192 SUM:31884
[Receive] 已确认收到 - Flag:2 SEQ:192 SUM:16381
发送了1024 字节， flag:0 SEQ:193 SUM:32012
[Receive] 已确认收到 - Flag:2 SEQ:193 SUM:16125
发送了1024 字节， flag:0 SEQ:194 SUM:32705
[Receive] 已确认收到 - Flag:2 SEQ:194 SUM:15869
发送了1024 字节， flag:0 SEQ:195 SUM:5470
[Receive] 已确认收到 - Flag:2 SEQ:195 SUM:15613
发送了1024 字节， flag:0 SEQ:196 SUM:20013
[Receive] 已确认收到 - Flag:2 SEQ:196 SUM:15357
发送了1024 字节， flag:0 SEQ:197 SUM:64731
[Receive] 已确认收到 - Flag:2 SEQ:197 SUM:15101
发送了1024 字节， flag:0 SEQ:198 SUM:34183
[Receive] 已确认收到 - Flag:2 SEQ:198 SUM:14845
发送了1024 字节， flag:0 SEQ:199 SUM:53422
[Receive] 已确认收到 - Flag:2 SEQ:199 SUM:14589
发送了1024 字节， flag:0 SEQ:200 SUM:42998
[Receive] 已确认收到 - Flag:2 SEQ:200 SUM:14333
发送了1024 字节， flag:0 SEQ:201 SUM:54965
[Receive] 已确认收到 - Flag:2 SEQ:201 SUM:14077
发送了1024 字节， flag:0 SEQ:202 SUM:36902
[Receive] 已确认收到 - Flag:2 SEQ:202 SUM:13821
发送了1024 字节， flag:0 SEQ:203 SUM:12255
发送数据超时
重新发送数据
[Receive] 已确认收到 - Flag:2 SEQ:203 SUM:13565
发送了1024 字节， flag:0 SEQ:204 SUM:24406
[Receive] 已确认收到 - Flag:2 SEQ:204 SUM:13309
发送了1024 字节， flag:0 SEQ:205 SUM:40136
[Receive] 已确认收到 - Flag:2 SEQ:205 SUM:13053
发送了1024 字节， flag:0 SEQ:206 SUM:59296
[Receive] 已确认收到 - Flag:2 SEQ:206 SUM:12797
发送了1024 字节， flag:0 SEQ:207 SUM:15402
[Receive] 已确认收到 - Flag:2 SEQ:207 SUM:12541
发送了1024 字节， flag:0 SEQ:208 SUM:47732
[Receive] 已确认收到 - Flag:2 SEQ:208 SUM:12285
发送了1024 字节， flag:0 SEQ:209 SUM:35568
[Receive] 已确认收到 - Flag:2 SEQ:209 SUM:12029

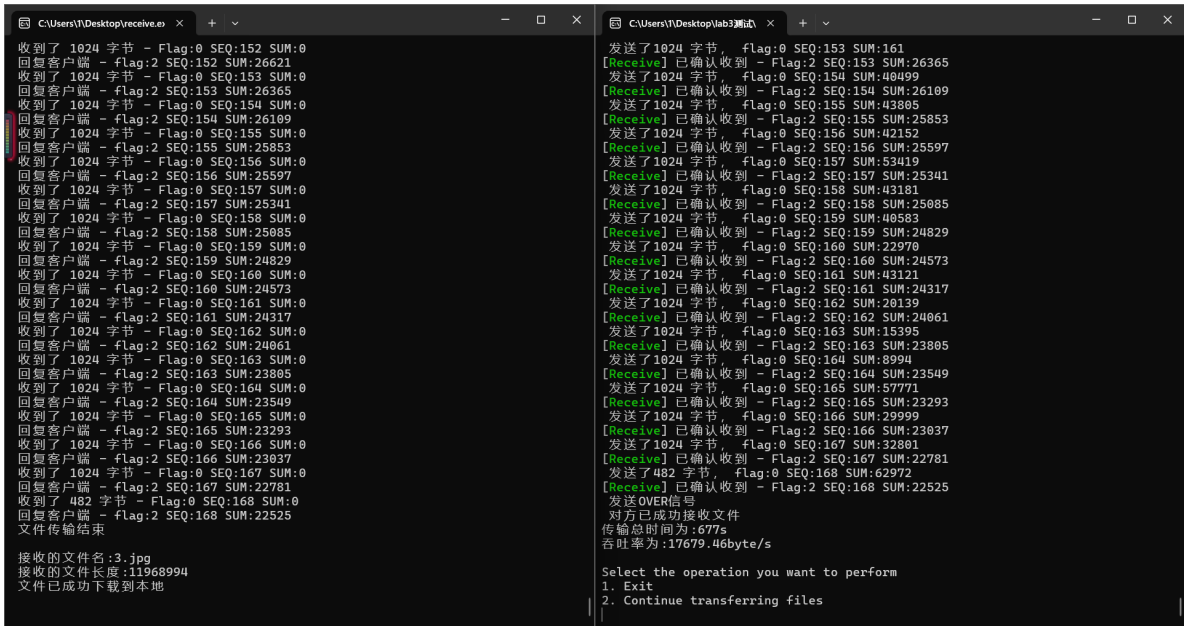
1.jpg


[Receive] 已确认收到 - Flag:2 SEQ:12 SUM:62461
发送了1024 字节， flag:0 SEQ:13 SUM:37579
发送数据超时
重新发送数据
[Receive] 已确认收到 - Flag:2 SEQ:13 SUM:62205
发送了1024 字节， flag:0 SEQ:14 SUM:29023
[Receive] 已确认收到 - Flag:2 SEQ:14 SUM:61949
发送了1024 字节， flag:0 SEQ:15 SUM:23827
[Receive] 已确认收到 - Flag:2 SEQ:15 SUM:61693
发送了1024 字节， flag:0 SEQ:16 SUM:63904
[Receive] 已确认收到 - Flag:2 SEQ:16 SUM:61437
发送了1024 字节， flag:0 SEQ:17 SUM:55420
[Receive] 已确认收到 - Flag:2 SEQ:17 SUM:61181
发送了1024 字节， flag:0 SEQ:18 SUM:52551
[Receive] 已确认收到 - Flag:2 SEQ:18 SUM:60925
发送了1024 字节， flag:0 SEQ:19 SUM:33471
[Receive] 已确认收到 - Flag:2 SEQ:19 SUM:60669
发送了1024 字节， flag:0 SEQ:20 SUM:10120
[Receive] 已确认收到 - Flag:2 SEQ:20 SUM:60413
发送了841 字节， flag:0 SEQ:21 SUM:40532
[Receive] 已确认收到 - Flag:2 SEQ:21 SUM:60157
发送OVER信号
对方已成功接收文件
传输总时间为:84s
吞吐率为:22111.35byte/s
Select the operation you want to perform
1. Exit
2. Continue transferring files
回复客户端 - flag:2 SEQ:10 SUM:62973
收到了 1024 字节 - Flag:0 SEQ:11 SUM:0
回复客户端 - flag:2 SEQ:11 SUM:62717
收到了 1024 字节 - Flag:0 SEQ:12 SUM:0
回复客户端 - flag:2 SEQ:12 SUM:62461
收到了 1024 字节 - Flag:0 SEQ:13 SUM:37579
回复客户端 - flag:2 SEQ:13 SUM:62205
收到了 1024 字节 - Flag:0 SEQ:14 SUM:0
回复客户端 - flag:2 SEQ:14 SUM:61949
收到了 1024 字节 - Flag:0 SEQ:15 SUM:0
回复客户端 - flag:2 SEQ:15 SUM:61693
收到了 1024 字节 - Flag:0 SEQ:16 SUM:0
回复客户端 - flag:2 SEQ:16 SUM:61437
收到了 1024 字节 - Flag:0 SEQ:17 SUM:0
回复客户端 - flag:2 SEQ:17 SUM:61181
收到了 1024 字节 - Flag:0 SEQ:18 SUM:0
回复客户端 - flag:2 SEQ:18 SUM:60925
收到了 1024 字节 - Flag:0 SEQ:19 SUM:0
回复客户端 - flag:2 SEQ:19 SUM:60669
收到了 1024 字节 - Flag:0 SEQ:20 SUM:0
回复客户端 - flag:2 SEQ:20 SUM:60413
收到了 841 字节 - Flag:0 SEQ:21 SUM:0
回复客户端 - flag:2 SEQ:21 SUM:60157
文件传输结束
接收的文件名:1.jpg
接收的文件长度:1857353
文件已成功下载到本地

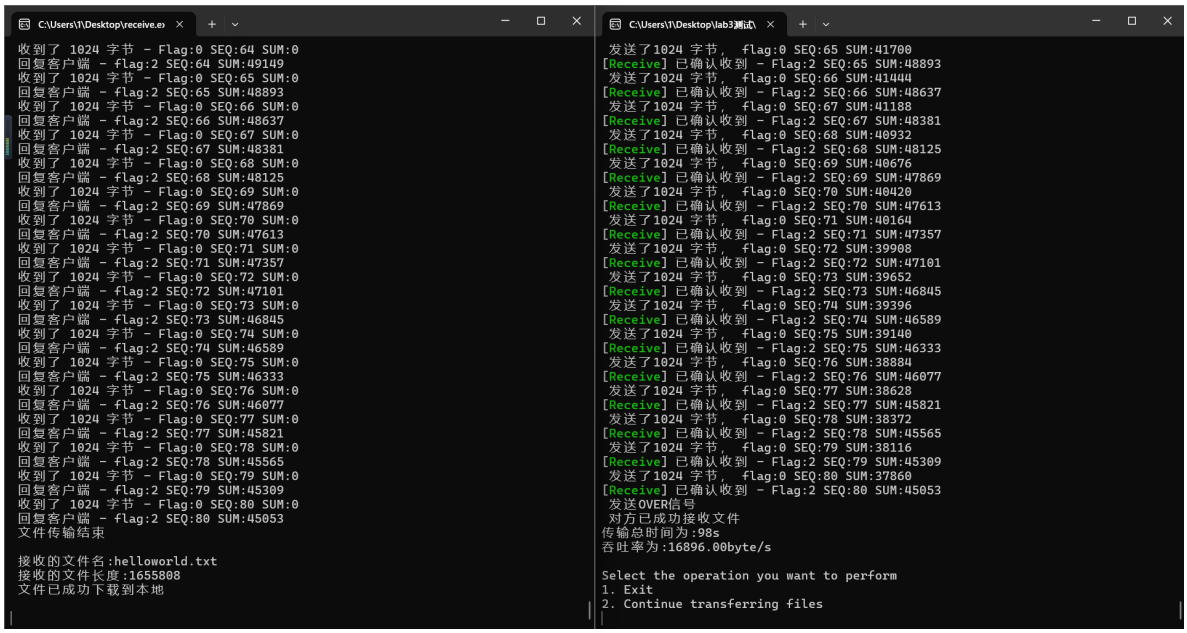
2.jpg



3.jpg



helloworld.txt



文件名	文件大小	传输时间	吞吐率
1.jpg	1857353byte	84s	22111.35byte/s
2.jpg	5898505byte	325s	18149.25byte/s
3.jpg	11968994byte	677s	17679.46byte/s
helloworld.txt	1655808byte	98s	16896.00byte/s