# 实验二 配置Web服务器，编写简单页面，分析交互过程

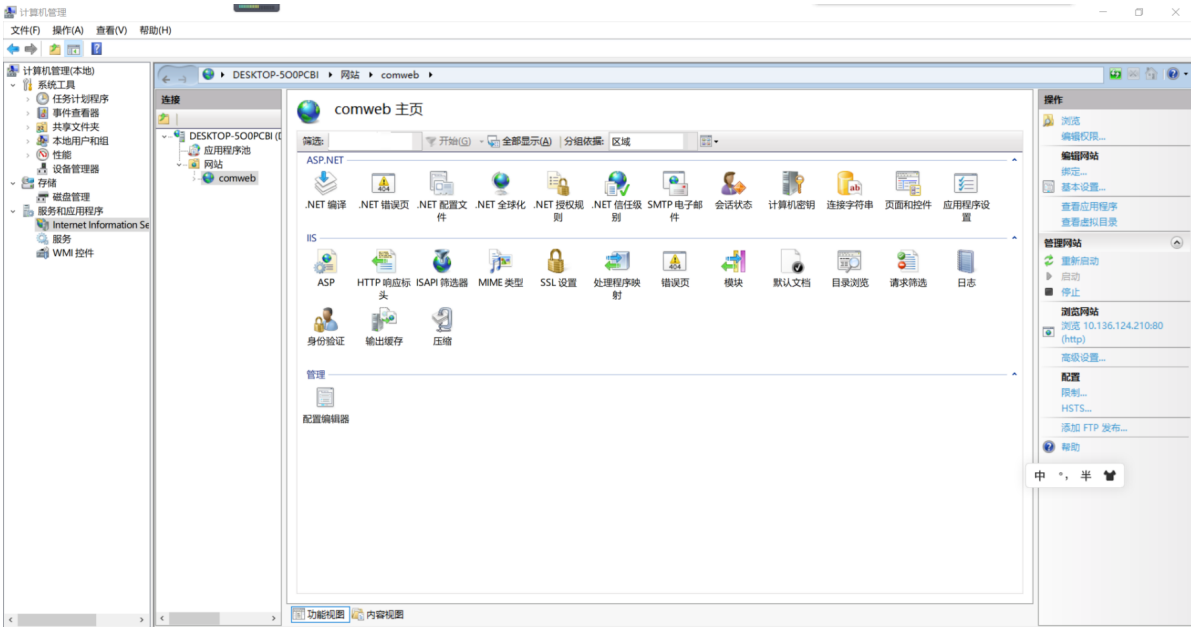

> 姓名：管昀玫
>
> 学号：2013750
>
> 专业：计算机科学与技术


## 实验要求

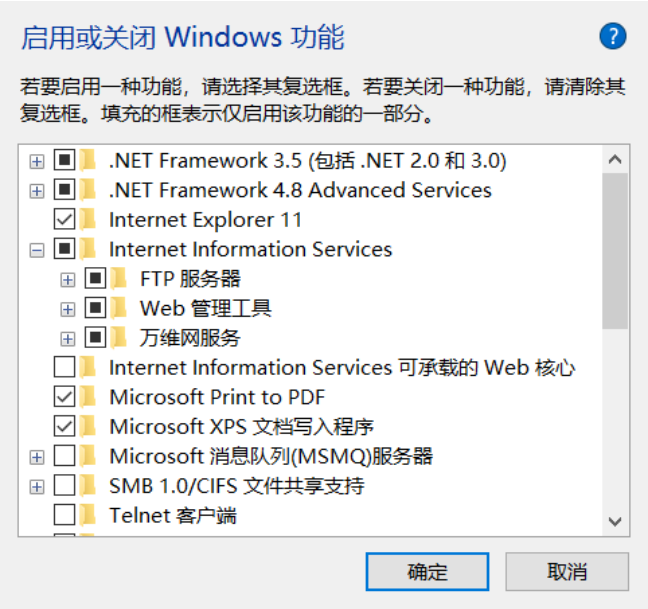
1. 搭建Web服务器（自由选择系统），并制作简单的Web页面，包含简单文本信息（至少包含专业、学号、姓名）和自己的LOGO。
2. 通过浏览器获取自己编写的Web页面，使用Wireshark捕获浏览器与Web服务器的交互过程，并进行简单的分析说明。
3. 提交实验报告。

## 搭建Web服务器

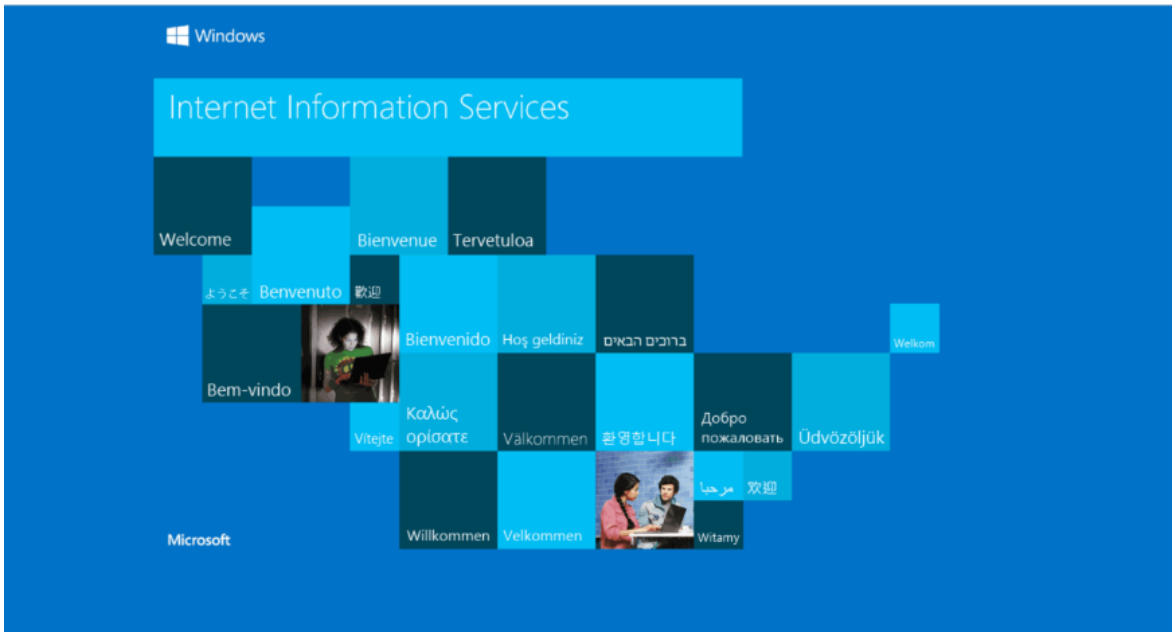
在win11上搭建Web服务器




1. 控制面版-程序-启用或关闭Windows功能-Internet information services的内容全部打钩。




2. 测试：在浏览器中输入localhost，显示下面图片




3. 搜索并打开iis管理工具，在网站上右击添加网站，起一个网站名字，物理路径为自己所建立的网页的目录，端口设置为大于80的数字。


4. 目录浏览->启用
5. 在浏览器打开http://localhost:8080即可使用

# 编写Web页面

本次实验主要目的是观察与服务器的交互过程，而不是注重页面的设计，所以这里就只编写了一个最简单的HTML页面

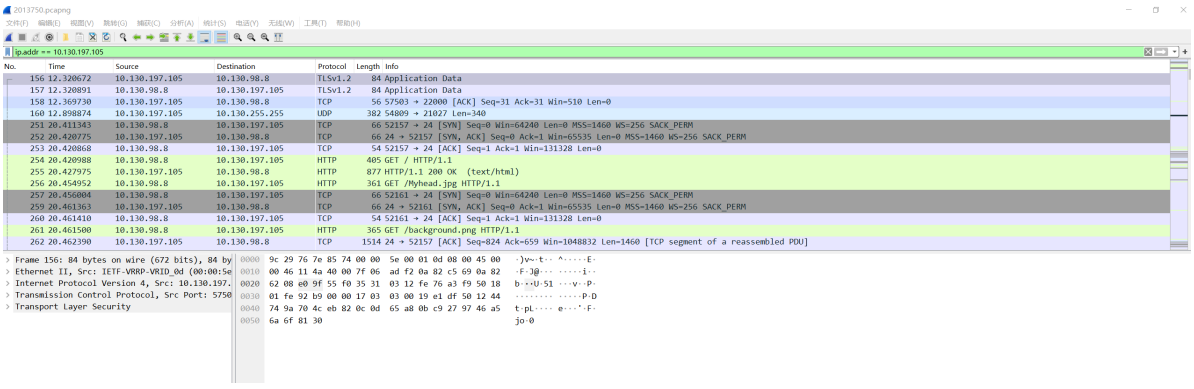
```
<!DOCTYPE HTML>
<html>
 <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <head>
     <title>MooseWeb</title>
  </head>
     <body>
  <body background="background.png">
        <h1><p align="center"><font color="black" face="Microsoft
YaHei">Mildred</font></p ></h1>
   <center><h3>管昀玫</h3>
   <h3>2013750</h3>
   <h3>计算机科学与技术</h3></center>
   <center><img src="Myhead.jpg" width="190" height="190"/></center>
  </body>
</html>
```

效果如下


# WireShark捕获与分析

为了更好地区分不同IP，本次实验使用两台电脑。服务器ip为`10.130.197.105`，客户端ip为`10.130.98.8`

内容经过滤后为




# 三次握手

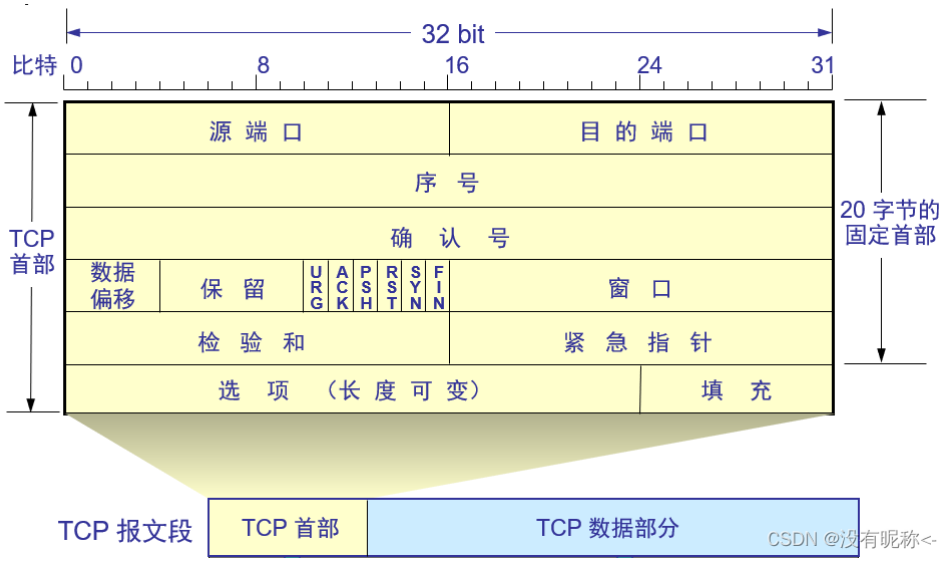
## 原理

1. **IP数据报的结构为**





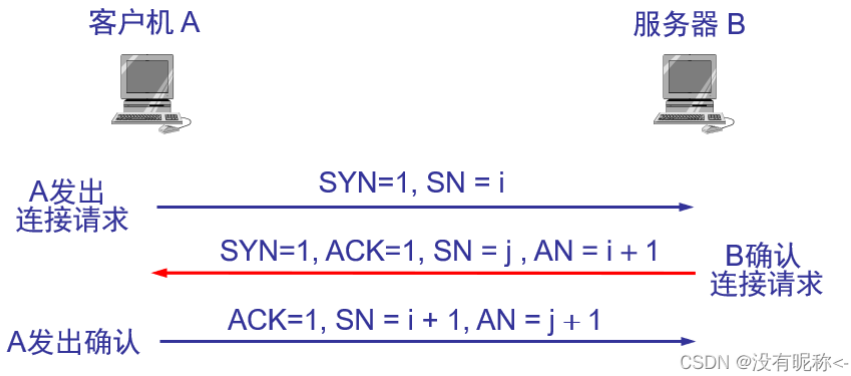
2. **TCP数据报的结构为**





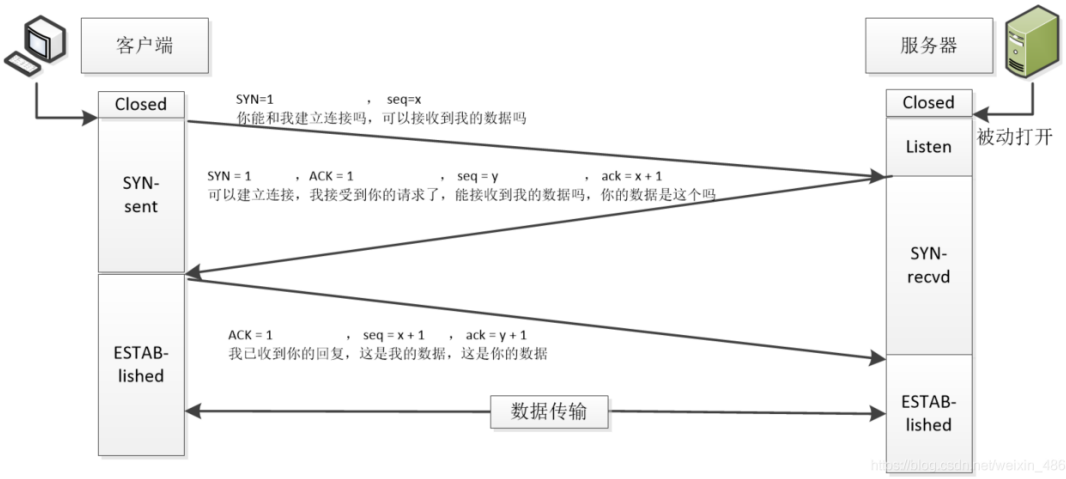

3. **三次握手流程**:

- A 客户进程向B服务器进程发出连接请求报文段，其首部中的SYN=1，并选择一个序列号SN=i，表明传送数据的第一个字节的序号是 i。此时，TCP客户进程进入**SYN-SENT**状态。
- B 服务器收到A的连接请求报文段后，如同意连接，则回答确认报文段。确认报文段首部中的SYN=1，ACK=1，其序列号SN=j，确认号AN=i+1。此时，TCP服务器进程进入**SYN-RCVD**状态。
- A 客户进程收到确认报文段后，还要向 B 回送确认。确认报文段首部中的ACK=1，确认号AN=j+1，序列号SN=i+1。此时，运行客户进程的A告知上层应用进程连接已建立(或打开)，进入**ESTABLISHED**状态。而运行服务器进程的 B 收到 A 的确认后，也通知上层应用进程，同样也进入**ESTABLISHED**状态。




- **序号(sequence number)**：seq序号，占32位，用来标识从TCP源端向目的端发送的字节流，发起方发送数据时对此进行标记。
- **确认号（acknowledgement number）**：ack序号，占32位，只有ACK标志位为1时，确认序号字段才有效，ack=seq+1。
- **标志位（Flags）**：共6个，即URG、ACK、PSH、RST、SYN、FIN等。具体含义如下：

> URG：紧急指针（urgent pointer）有效。
>
> ACK：确认序号有效。（为了与**确认号ack**区分开，我们用大写表示）
>
> PSH：接收方应该尽快将这个报文交给应用层。
>
> RST：重置连接。
>
> SYN：发起一个新连接。
>
> FIN：释放一个连接。




## 分析

### 第一次握手

```
251 20.411343    10.130.98.8    10.130.197.105    TCP    66 52157 → 24 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM
```

第一次握手的时候，从本地客户端发送了一个TCP请求，可以在info里看到端口号。本地端口号为52157，而目的端口号为24。并且序列号Seq设置为0

查看链路层封装的包：

```
> Frame 251: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface \Device
v Ethernet II, Src: IntelCor_7e:85:74 (9c:29:76:7e:85:74), Dst: IETF-VRRP-VRID_0d (00:00:5e
  v Destination: IETF-VRRP-VRID_0d (00:00:5e:00:01:0d)
      Address: IETF-VRRP-VRID_0d (00:00:5e:00:01:0d)
      .... ..0. .... .... .... .... = LG bit: Globally unique address (factory default)
      .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
  v Source: IntelCor_7e:85:74 (9c:29:76:7e:85:74)
      Address: IntelCor_7e:85:74 (9c:29:76:7e:85:74)
      .... ..0. .... .... .... .... = LG bit: Globally unique address (factory default)
      .... ...0 .... .... .... .... = IG bit: Individual address (unicast)
    Type: IPv4 (0x0800)
```

可以发现这里告诉我们上层（也就是网络层）使用的是IP协议。

而在网络层中可以发现：

```
v Internet Protocol Version 4, Src: 10.130.98.8, Dst: 10.130.197.105
    0100 .... = Version: 4
    .... 0101 = Header Length: 20 bytes (5)
  > Differentiated Services Field: 0x00 (DSCP: CS0, ECN: Not-ECT)
    Total Length: 52
    Identification: 0xe5e8 (58856)
  > 010. .... = Flags: 0x2, Don't fragment
    ...0 0000 0000 0000 = Fragment Offset: 0
    Time to Live: 128
    Protocol: TCP (6)
    Header Checksum: 0x0000 [validation disabled]
    [Header checksum status: Unverified]
    Source Address: 10.130.98.8
    Destination Address: 10.130.197.105
```

这里告诉我们上层（也就是传输层）使用的是TCP协议。并且在这一层里可以看见有校验和是0xe5e8，并且标明了发出请求的源IP以及目标IP。

在传输层中：

```
v Transmission Control Protocol, Src Port: 52157, Dst Port: 24, Seq: 0, Len: 0
    Source Port: 52157
    Destination Port: 24
    [Stream index: 50]
    [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 0]
    Sequence Number: 0    (relative sequence number)
    Sequence Number (raw): 1946876011
    [Next Sequence Number: 1    (relative sequence number)]
    Acknowledgment Number: 0
    Acknowledgment number (raw): 0
    1000 .... = Header Length: 32 bytes (8)
  > Flags: 0x002 (SYN)
    Window: 64240
    [Calculated window size: 64240]
    Checksum: 0x3c9c [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  > Options: (12 bytes), Maximum segment size, No-Operation (NOP), Window scale, No-Operat
  > [Timestamps]
```

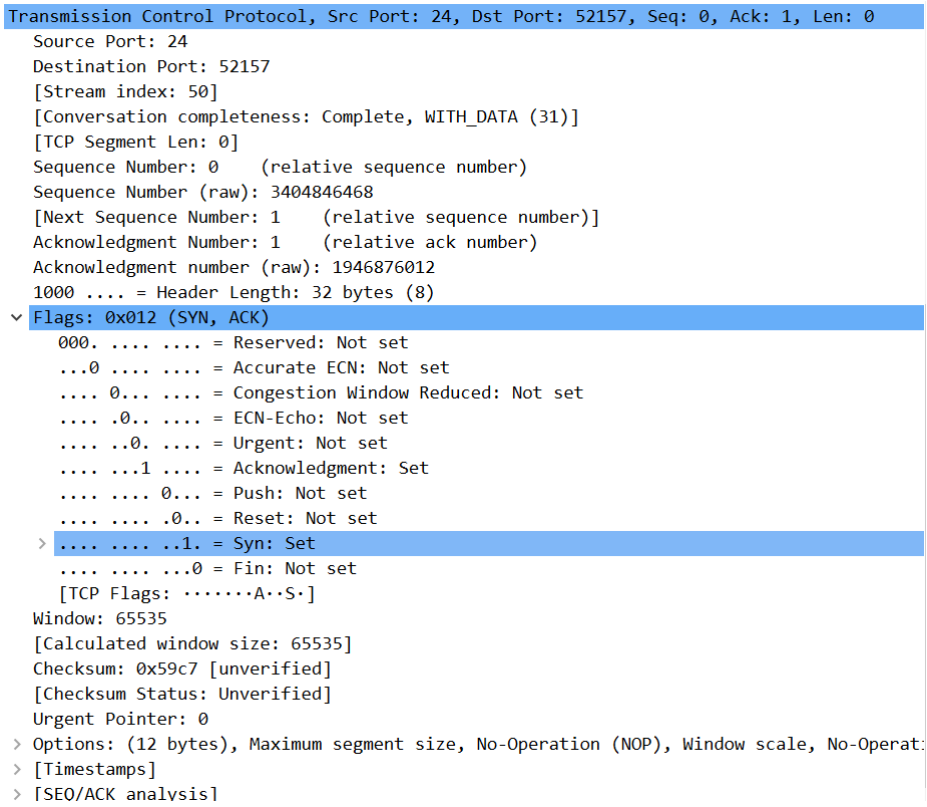
这里标明了目的端口号和源端口号，并且flags是0x002，即SYN。

第一次握手通俗地说，是客户端告诉服务器，说我现在要请求和你建立连接了。

**第二次握手**

```
252 20.420775    10.130.197.105    10.130.98.8    TCP    66 24 → 52157 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=256 SACK_PERM
```

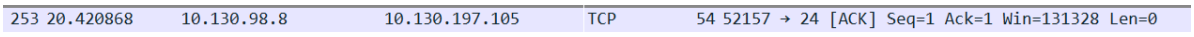
在第二次握手中可以发现，此时是远程给本地发送了一个应答，同时在右边可以看见seq的后面有ack=1




在此处的传输层协议可以看见，序列化依然为0，ACK的标志位设置为了1，同时SYN被置位，下一个Seq为1.

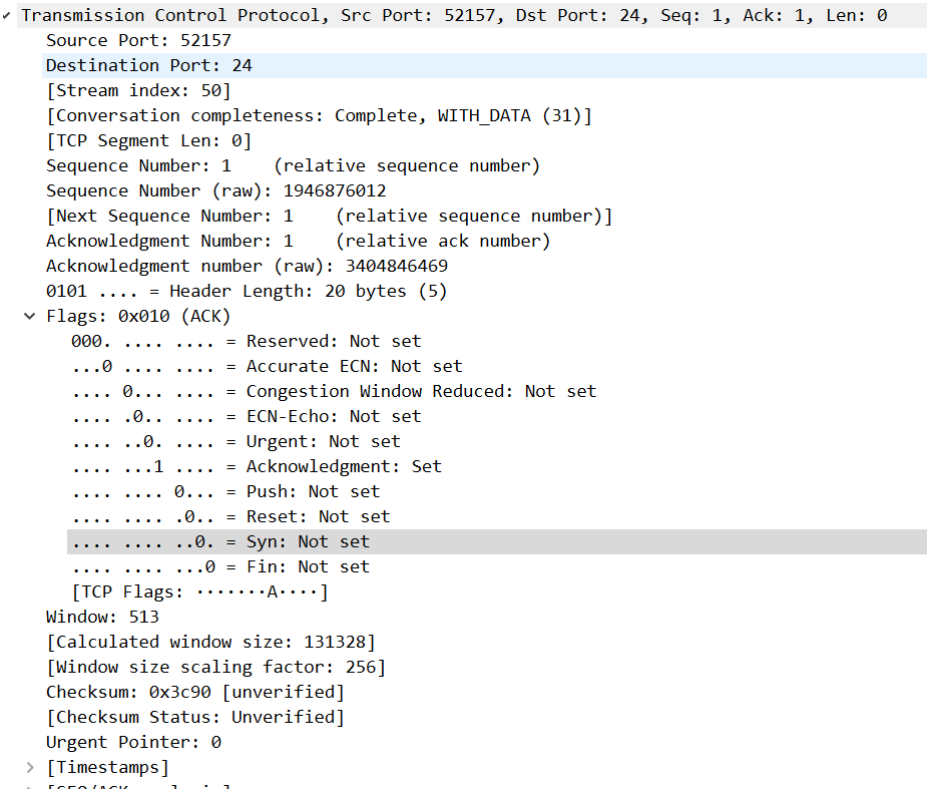
通俗的说，这里是服务器给客户端一个回复，告诉客户端我收到了你要进行连接的请求，并且说现在可以建立连接。

**第三次握手**




在第三次握手时，seq被设置成了1，ack也是1。

- 这里就是客户端在收到服务器发来的包以后检查确认序号ack是否正确：在ACK=1的情况下，ack = 第一次发送的序号seq+1（X+1 = 0+1 =1）。
- 当正确的时候，客户端会再向服务器发送一个数据包，SYN=0，ACK=1，确认序号ack=Y+1=0+1=1，并且把服务器发来ACK的序号seq(Sequence number)加1发送给对方，发送序号seq为X+1= 0+1=1。
- 客户端收到后确认序号值与ACK=1，52157->24。




这里就是当客户端回复服务器，告诉服务器我能收到你的回复，现在开始正式建立连接吧。至此，一次TCP连接就此建立，可以传送数据了。

## 总结

各字段在TCP三次握手中的作用：

- **SYN**：用于**建立连接**
- **ACK**：用于确定**收到了请求。**
- **seq**：发送**自己的**数据。
- **ack**：发送**接收到的对方的**数据。

三次握手的原因：

客户端和服务端通信前要进行连接，“3次握手”的作用就是双方都能明确自己和对方的收、发能力是正常的。

- **第一次握手**：客户端发送网络包，服务端收到了。这样服务端就能得出结论：客户端的发送能力、服务端的接收能力是正常的。
- **第二次握手**：服务端发包，客户端收到了。这样客户端就能得出结论：服务端的接收、发送能力，客户端的接收、发送能力是正常的。从客户端的视角来看，我接到了服务端发送过来的响应数据包，说明服务端接收到了我在第一次握手时发送的网络包，并且成功发送了响应数据包，这就说明，服务端的接收、发送能力正常。而另一方面，我收到了服务端的响应数据包，说明我第一次发送的网络包成功到达服务端，这样，我自己的发送和接收能力也是正常的。
- **第三次握手**：客户端发包，服务端收到了。这样服务端就能得出结论：客户端的接收、发送能力，服务端的发送、接收能力是正常的。第一、二次握手后，服务端并不知道客户端的接收能力以及自己的发送能力是否正常。

而在第三次握手时，服务端收到了客户端对第二次握手作的回应。从服务端的角度，我在第二次握手时的响应数据发送出去了，客户端接收到了。所以，我的发送能力是正常的。而客户端的接收能力也是正常的。

经历了上面的三次握手过程，客户端和服务端都确认了自己的接收、发送能力是正常的。之后就可以正常通信了。

## 中间过程

在建立起TCP之后，客户端会向服务器发送一个HTTP请求，并且请求的时候附带着GET参数

```
254 20.420988    10.130.98.8          10.130.197.105       HTTP      405 GET / HTTP/1.1
```

```
Transmission Control Protocol, Src Port: 52157, Dst Port: 24, Seq: 1, Ack: 1, Len: 351
    Source Port: 52157
    Destination Port: 24
    [Stream index: 50]
    [Conversation completeness: Complete, WITH_DATA (31)]
    [TCP Segment Len: 351]
    Sequence Number: 1    (relative sequence number)
    Sequence Number (raw): 1946876012
    [Next Sequence Number: 352    (relative sequence number)]
    Acknowledgment Number: 1    (relative ack number)
    Acknowledgment number (raw): 3404846469
    0101 .... = Header Length: 20 bytes (5)
  ▾ Flags: 0x018 (PSH, ACK)
      000. .... .... = Reserved: Not set
      ...0 .... .... = Accurate ECN: Not set
      .... 0... .... = Congestion Window Reduced: Not set
      .... .0.. .... = ECN-Echo: Not set
      .... ..0. .... = Urgent: Not set
      .... ...1 .... = Acknowledgment: Set
      .... .... 1... = Push: Set
      .... .... .0.. = Reset: Not set
      .... .... ..0. = Syn: Not set
      .... .... ...0 = Fin: Not set
      [TCP Flags: ·····AP···]
    Window: 513
    [Calculated window size: 131328]
    [Window size scaling factor: 256]
    Checksum: 0x3def [unverified]
    [Checksum Status: Unverified]
    Urgent Pointer: 0
  ▸ [Timestamps]
  ▸ [SEQ/ACK analysis]
    TCP payload (351 bytes)
Hypertext Transfer Protocol
```

可以看到此处传输层中，Push: Set被置为1，Flags为0x018，即PSH,ACK

之后连续的两条就是服务器告诉客户端，text传输完毕；客户端在解析数据报后发现Myhead.jpg资源没有，这个时候客户端会再向服务器发出一个GET请求，表示要请求Myhead.jpg资源；

```
255 20.427975    10.130.197.105    10.130.98.8       HTTP      877 HTTP/1.1 200 OK  (text/html)
256 20.454952    10.130.98.8       10.130.197.105    HTTP      361 GET /Myhead.jpg HTTP/1.1
```

此条同理，请求background.png资源

```
261 20.461500    10.130.98.8          10.130.197.105       HTTP      365 GET /background.png HTTP/1.1
```

之后是一堆TCP segment of a reassembled PDU

```
278 20.472500    10.130.197.105    10.130.98.8       TCP      1514 24 → 52157 [ACK] Seq=16884 Ack=659 Win=1048832 Len=1460 [TCP segment of a reassembled PDU]
279 20.472500    10.130.197.105    10.130.98.8       TCP      1514 24 → 52157 [ACK] Seq=18344 Ack=659 Win=1048832 Len=1460 [TCP segment of a reassembled PDU]
280 20.472543    10.130.98.8       10.130.197.105    TCP        54 52157 → 24 [ACK] Seq=659 Ack=19804 Win=131328 Len=0
281 20.474040    10.130.197.105    10.130.98.8       TCP      1514 24 → 52157 [ACK] Seq=19804 Ack=659 Win=1048832 Len=1460 [TCP segment of a reassembled PDU]
282 20.474040    10.130.197.105    10.130.98.8       TCP      1514 24 → 52157 [ACK] Seq=21264 Ack=659 Win=1048832 Len=1460 [TCP segment of a reassembled PDU]
283 20.474040    10.130.197.105    10.130.98.8       TCP      1514 24 → 52157 [ACK] Seq=22724 Ack=659 Win=1048832 Len=1460 [TCP segment of a reassembled PDU]
284 20.474040    10.130.197.105    10.130.98.8       TCP      1514 24 → 52157 [ACK] Seq=24184 Ack=659 Win=1048832 Len=1460 [TCP segment of a reassembled PDU]
285 20.474040    10.130.197.105    10.130.98.8       TCP      1514 24 → 52157 [ACK] Seq=25644 Ack=659 Win=1048832 Len=1460 [TCP segment of a reassembled PDU]
286 20.474040    10.130.197.105    10.130.98.8       TCP      1514 24 → 52161 [ACK] Seq=1 Ack=312 Win=1049344 Len=1460 [TCP segment of a reassembled PDU]
287 20.474040    10.130.197.105    10.130.98.8       TCP      1514 24 → 52161 [ACK] Seq=1461 Ack=312 Win=1049344 Len=1460 [TCP segment of a reassembled PDU]
288 20.474040    10.130.197.105    10.130.98.8       TCP      1514 24 → 52161 [ACK] Seq=2921 Ack=312 Win=1049344 Len=1460 [TCP segment of a reassembled PDU]
289 20.474040    10.130.197.105    10.130.98.8       TCP      1514 24 → 52161 [ACK] Seq=4381 Ack=312 Win=1049344 Len=1460 [TCP segment of a reassembled PDU]
290 20.474040    10.130.197.105    10.130.98.8       TCP      1514 24 → 52161 [ACK] Seq=5841 Ack=312 Win=1049344 Len=1460 [TCP segment of a reassembled PDU]
291 20.474040    10.130.197.105    10.130.98.8       TCP      1514 24 → 52161 [ACK] Seq=7301 Ack=312 Win=1049344 Len=1460 [TCP segment of a reassembled PDU]
292 20.474040    10.130.197.105    10.130.98.8       TCP      1514 24 → 52161 [ACK] Seq=8761 Ack=312 Win=1049344 Len=1460 [TCP segment of a reassembled PDU]
```

这是因为，在基于TCP传输消息是，对于上面的应用层如果出于某些原因（如超过MSS）TCP Segment不能一次包含全部的应用层PDU，而要把一个完整消息分成多个段，就会将除了最后一个分段（segment）的所有其他分段都打上“TCP segment of a reassembled PDU”。

下面这两句是服务器在告诉客户端，图片传输完毕

| | | | | | |
|---|---|---|---|---|---|
| 295 | 20.474988 | 10.130.197.105 | 10.130.98.8 | HTTP | 596 HTTP/1.1 200 OK (PNG) |

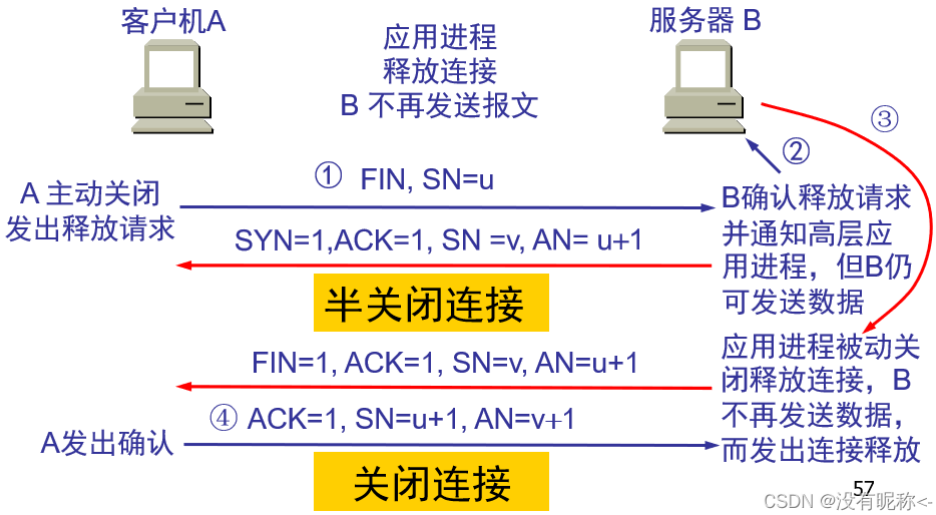
| | | | | | |
|---|---|---|---|---|---|
| 345 | 20.486712 | 10.130.197.105 | 10.130.98.8 | HTTP | 315 HTTP/1.1 200 OK (JPEG JFIF image) |

# 四次挥手

## 原理

1. 若由A向B发出连接释放报文段，其首部中的FIN=1，选择一个序列号 SN=u，它是前面已传送过的数据的最后一个字节的序列号加1，表示发送数据已告结束，主动关闭TCP连接。此时A进入**FIN-WAIT-1**状态，等待来自B的确认。
2. B收到释放连接报文后，如同意连接，则回答确认报文段。确认报文段首部中的SYN=1，ACK=1，其序列号SN=v，确认号AN=u+1。然后B进入**CLOSED-WAIT**状态，同时通知高层应用进程。这样，从A到B的连接就释放了，连接处于半关闭(half-close)状态。A收到来自B的确认报文段后，就进入**FIN-WAIT-2**状态，等待B再发来连接释放报文段。
3. 此后，B不再接收来自A的数据，但B若有数据要发往A，仍可继续发送。若B向A的数据发送完毕后，就向A发出连接释放报文段。在此报文段中应将FIN=1，SN=v (它是前面已传送过的数据的最后一个字节的序号加1)。另外，必须重复上次已发送过的确认号AN=u+1。此时B进入**LAST-ACK**状态，等待A发来的确认报文段。
4. A收到连接释放报文段后，必须对此发出确认，其确认号为AN=v+1，而序列号SN=u+1。然后进入到**TIME-WAIT**状态。B收到了来自A的确认报文段后，就进入**CLOSED**状态，并撤消相应的传输控制块TCB，就结束了本次的TCP连接。

注意：进入到TIME-WAIT状态后，本次TCP接还没有完全释放掉，必须再经过时间等待计时器设置的时间(=2MSL)后，A才进入到CLOSED状态，此时整个连接才全部释放。







## 分析

挥手的过程可以由任何一方发起，此处为服务器端发起。

### 第一次挥手

服务器端发出FIN，用来断开服务器端到客户端的数据传送，进入FIN-WAIT-1状态

| | | | | | |
|---|---|---|---|---|---|
| 501 | 25.499721 | 10.130.197.105 | 10.130.98.8 | TCP | 56 24 → 52157 [FIN, ACK] Seq=76140 Ack=967 Win=1048576 Len=0 |

```
0101 .... = Header Length: 20 bytes (5)
Flags: 0x011 (FIN, ACK)
    000. .... .... = Reserved: Not set
    ...0 .... .... = Accurate ECN: Not set
    .... 0... .... = Congestion Window Reduced: Not set
    .... .0.. .... = ECN-Echo: Not set
    .... ..0. .... = Urgent: Not set
    .... ...1 .... = Acknowledgment: Set
    .... .... 0... = Push: Not set
    .... .... .0.. = Reset: Not set
    .... .... ..0. = Syn: Not set
  > .... .... ...1 = Fin: Set
  > [TCP Flags: ·····A···F]
Window: 4099
[Calculated window size: 1049344]
[Window size scaling factor: 256]
Checksum: 0x8718 [unverified]
[Checksum Status: Unverified]
```

可以看到，此处Flags中的FIN为被设置为1，告诉服务器这个数据包的功能是要准备关闭连接。

**第二次挥手**

客户端收到服务器端的FIN后，发送ACK确认报文，进入CLOSE-WAIT状态

```
502 25.499775    10.130.98.8     10.130.197.105    TCP    54 52157 → 24 [ACK] Seq=967 Ack=76141 Win=130304 Len=0
```

可以看见，此处ACK = 1，ack = 76141 = 76140 + 1 = 上一次Seq + 1。Seq = 上一次的ack = 967.

```
Flags: 0x010 (ACK)
    000. .... .... = Reserved: Not set
    ...0 .... .... = Accurate ECN: Not set
    .... 0... .... = Congestion Window Reduced: Not set
    .... .0.. .... = ECN-Echo: Not set
    .... ..0. .... = Urgent: Not set
    .... ...1 .... = Acknowledgment: Set
    .... .... 0... = Push: Not set
    .... .... .0.. = Reset: Not set
    .... .... ..0. = Syn: Not set
    .... .... ...0 = Fin: Not set
    [TCP Flags: ·····A····]
Window: 4099
[Calculated window size: 1049344]
[Window size scaling factor: 256]
Checksum: 0x8717 [unverified]
[Checksum Status: Unverified]
Urgent Pointer: 0
```

此处FIN=0，单纯是回复客户端，告诉客户端：我（服务器）收到了你（客户端）要关闭连接的请求。

**第三次挥手**

客户端发出FIN，用来断开客户端到服务器端的数据传送，进入LAST-ACK状态

```
503 25.499817    10.130.98.8     10.130.197.105    TCP    54 52157 → 24 [FIN, ACK] Seq=967 Ack=76141 Win=130304 Len=0
```

现在是由服务器给客户端发送数据包，并把FIN设置为1，告诉客户端现在服务器要要关闭连接了。

```
0101 .... = Header Length: 20 bytes (5)
Flags: 0x011 (FIN, ACK)
    000. .... .... = Reserved: Not set
    ...0 .... .... = Accurate ECN: Not set
    .... 0... .... = Congestion Window Reduced: Not set
    .... .0.. .... = ECN-Echo: Not set
    .... ..0. .... = Urgent: Not set
    .... ...1 .... = Acknowledgment: Set
    .... .... 0... = Push: Not set
    .... .... .0.. = Reset: Not set
    .... .... ..0. = Syn: Not set
  > .... .... ...1 = Fin: Set
  > [TCP Flags: ·····A···F]
Window: 511
[Calculated window size: 130816]
[Window size scaling factor: 256]
Checksum: 0x3c90 [unverified]
[Checksum Status: Unverified]
```

可以看到此处FIN=1，告诉服务器要关闭连接了。而且，Seq=76141=上一次的Seq，Ack=967 = 上一次的Ack。

**第四次挥手**

服务器端收到客户端的FIN后，发送ACK确认报文，进入TIME-WAIT状态，服务器端等待2个最长报文段寿命后进入Close状态；客户端收到确认后，立刻进入Close状态。

```
504 25.502746    10.130.197.105    10.130.98.8    TCP    56 24 → 52157 [ACK] Seq=76141 Ack=968 Win=1048576 Len=0
```

此处，Seq = 76141 = 上一次的Ack，Ack = 968 = 967 + 1 = 上一次的Seq + 1.

```
Flags: 0x010 (ACK)
    000. .... .... = Reserved: Not set
    ...0 .... .... = Accurate ECN: Not set
    .... 0... .... = Congestion Window Reduced: Not set
    .... .0.. .... = ECN-Echo: Not set
    .... ..0. .... = Urgent: Not set
    .... ...1 .... = Acknowledgment: Set
    .... .... 0... = Push: Not set
    .... .... .0.. = Reset: Not set
    .... .... ..0. = Syn: Not set
    .... .... ...0 = Fin: Not set
    [TCP Flags: ·····A····]
  Window: 4099
  [Calculated window size: 1049344]
  [Window size scaling factor: 256]
  Checksum: 0x8717 [unverified]
  [Checksum Status: Unverified]
```

服务器收到客户端的关闭连接的数据包，回复FIN = 0，告诉客户端收到了要关闭的操作。

至此，4次挥手过程结束，连接被关闭。

## 总结

四次握手的原因：TCP协议是一种面向连接的、可靠的、基于字节流的运输层通信协议。TCP是全双工模式，这就意味着，当主机1发出`FIN`报文段时，只是表示主机1已经没有数据要发送了，主机1告诉主机2，它的数据已经全部发送完毕了；但是，这个时候主机1还是可以接受来自主机2的数据；当主机2返回`ACK`报文段时，表示它已经知道主机1没有数据发送了，但是主机2还是可以发送数据到主机1的；当主机2也发送了`FIN`报文段时，这个时候就表示主机2也没有数据要发送了，就会告诉主机1，我也没有数据要发送了，之后彼此就会愉快的中断这次TCP连接。

- `FIN_WAIT_1`：其实`FIN_WAIT_1`和`FIN_WAIT_2`状态的真正含义都是表示等待对方的FIN报文。而这两种状态的区别是：
  - `FIN_WAIT_1`状态实际上是当SOCKET在ESTABLISHED状态时，它想主动关闭连接，向对方发送了`FIN`报文，此时该SOCKET即进入到`FIN_WAIT_1`状态。
  - 当对方回应ACK报文后，则进入到`FIN_WAIT_2`状态，当然在实际的正常情况下，无论对方何种情况下，都应该马上回应ACK报文，所以`FIN_WAIT_1`状态一般是比较难见到的，而`FIN_WAIT_2`状态还有时常常可以用netstat看到。（主动方）
- `FIN_WAIT_2`：`FIN_WAIT_2`状态下的SOCKET，表示半连接，也即有一方要求close连接，但另外还告诉对方，我暂时还有点数据需要传送给你(ACK信息)，稍后再关闭连接。（主动方）
- `CLOSE_WAIT`：这种状态的含义是表示在等待关闭。当对方close一个SOCKET后发送`FIN`报文给自己，系统会回应一个ACK报文给对方，此时则进入到`CLOSE_WAIT`状态。接下来，真正需要考虑的事情是察看是否还有数据发送给对方，如果没有的话，就可以close这个SOCKET，发送`FIN`报文给对方，也即关闭连接。（被动方）
- `LAST_ACK`：被动关闭一方在发送`FIN`报文后，最后等待对方的ACK报文。当收到ACK报文后，也即可以进入到CLOSED可用状态。（被动方）
- `TIME_WAIT`：表示收到了对方的FIN报文，并发送出了ACK报文，就等2MSL后即可回到CLOSED可用状态了。如果FINWAIT1状态下，收到了对方同时带FIN标志和ACK标志的报文时，可以直接进入到`TIME_WAIT`状态，而无须经过`FIN_WAIT_2`状态。（主动方）
- `CLOSED`：表示连接中断。

# 其他问题

## 两个“三次握手，资源传递，四次挥手”

在抓包的时候，正常来说只会建立一个连接，也就是只有一个“三次握手，资源传递，四次挥手”。但是我在实际上抓的时候发现，使用Internet Explorer的老版本时，只会有一次连接；而在使用Chorme的时候，却建立了两次连接，也就是总共进行了两个“三次握手，资源传递，四次挥手”。

两个三次握手：

```
56 58567 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM=1
56 80 → 58567 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=65495 WS=256 SACK_PERM=1
44 58567 → 80 [ACK] Seq=1 Ack=1 Win=2619648 Len=0
56 64801 → 80 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM=1
56 80 → 64801 [SYN, ACK] Seq=0 Ack=1 Win=8192 Len=0 MSS=65495 WS=256 SACK_PERM=1
44 64801 → 80 [ACK] Seq=1 Ack=1 Win=2619648 Len=0
```

两个四次挥手：

```
44 64801 → 80 [FIN, ACK] Seq=1 Ack=1 Win=2619648 Len=0
44 80 → 64801 [ACK] Seq=1 Ack=2 Win=262400 Len=0
44 80 → 64801 [FIN, ACK] Seq=1 Ack=2 Win=262400 Len=0
44 64801 → 80 [ACK] Seq=2 Ack=2 Win=2619648 Len=0
44 58567 → 80 [FIN, ACK] Seq=1817 Ack=296894 Win=2585856 Len=0
44 80 → 58567 [ACK] Seq=296894 Ack=1818 Win=2100992 Len=0
44 80 → 58567 [FIN, ACK] Seq=296894 Ack=1818 Win=2100992 Len=0
44 58567 → 80 [ACK] Seq=1818 Ack=296895 Win=2585856 Len=0
```

经过网上的查询，我发现这个其实是现在浏览器对访问加速的一个机制，因为HTTP/1.1有Head-of-line blocking，在同一个TCP连接上，新的HTTP请求一定要等到收到上一次的HTTP响应才能发生 (先不考虑HTTP pipelining)。如果一次请求有多个资源链接返回，多个HTTP请求需要并发的话，是需要多个TCP连接的，从而会引入TCP连接建立3次握手的延迟。现代浏览器一般都对一个Host建立多个TCP连接。

如果所有TCP连接上都有ongoing HTTP request/response, 这个时候只能等上一个HTTP请求完成再性能下一次请求了，性能会大打折扣。在HTTP/2中一个重要的概念就是Multiplexing, 在一个TCP连接中可以实现多个HTTP请求和响应的复用，所以一个TCP连接上service多个HTTP请求和响应，性能提升很明显。

同时在抓包的时候，有的时候四次挥手的数据包并不是4个而是3个（这个是在查阅相关资料的时候看见的，但是自己多次尝试并没有发现这个现象），根据网上查询到的叙述表示，这个是服务端的一个优化。因为在正常的四次挥手过程中，第二次和第三次都是由服务器向客户端发送数据包，第一个是表示收到要关闭的信息，第二个是告诉客户端现在要关闭。而正常情况下，服务器给客户端发送信息告诉客户端要关闭连接的前提就是收到了来自客户端要关闭连接的数据包（假设服务器是不会主动关闭连接的），所以是没有必要分为两次发送数据包，直接发送带有FIN的数据包即可。即，服务器收到请求后，如果没有更多数据需要发送，则将第二次挥手与第三次挥手合为一个数据包同时发送给本机。第二个挥手数据包中FIN和ACK标志同时被置1。所以这里只有“三次挥手”。

## 四次挥手与资源释放

四次挥手后，最终还需要等待2MSL。因为第四次挥手的时候，客户端向服务器端发送确认报文可能丢失，等待一定的时间是为了重发可能丢失的ACK报文。由于超时数据包可能有也可能没有，所以约定等待2MSL，有则接受，无则释放资源。

## 第二次挥手和第三次挥手之间的关闭等待时间

这是因为服务器端可能有未发送完毕的数据，这个关闭等待时间是用来继续发送数据的。