1. TCP 协议

TCP（Transmission Control Protocol 传输控制协议）是一种面向连接的、可靠的、基于字节流的传输层通信协议。

udp 通信模型中，在通信开始之前，不需要建立相关的链接，只需要发送数据即可,但是 TCP 通信的话是需要经过创建连接、数据传送、终止连接三个步骤。

TCP 通信模型中，在通信开始之前，一定要先建立相关的链接才能发送数据，类似于生活中打电话

1.1. TCP 特点

1. TCP 采用发送应答机制：

   TCP 发送的每个报文段都必须得到接收方的应答才认为这个 TCP 报文段传输成功。

2. 超时重传：

   发送端发出一个报文段之后就启动定时器，如果在定时时间内没有收到应答就重新发送这个 报文段。

   TCP 为了保证不发生丢包，就给每个包一个序号，同时序号也保证了传送到接收端实体的 包的按序接收。然后接收端实体对已成功收到的包发回一个相应的确认（ACK）；如果发 送端实体在合理的往返时延（RTT）内未收到确认，那么对应的数据包就被假设为已丢失将 会被进行重传。

3. 错误校验：

   TCP 用一个校验和函数来检验数据是否有错误；在发送和接收时都要计算校验和。

4. 流量控制和阻塞管理：

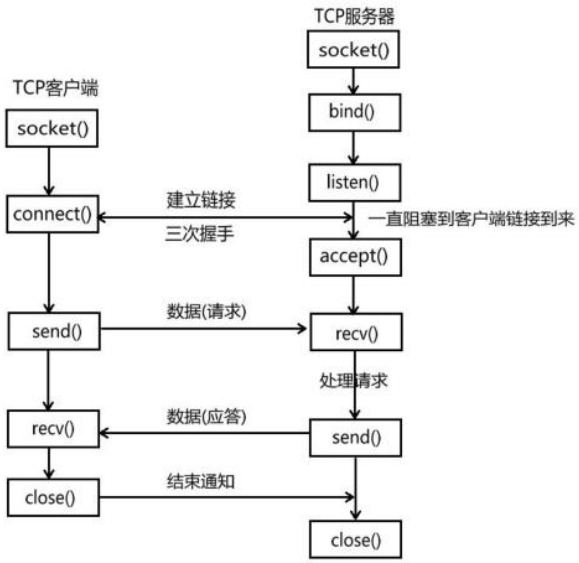
   流量控制用来避免主机发送得过快而使接收方来不及完全收下。

1.2. TCP 与 UDP 的不同点

- 面向连接（确认有创建三方交握，连接已创建才作传输。）
- 有序数据传输
- 重发丢失的数据包
- 舍弃重复的数据包
- 无差错的数据传输

- 阻塞/流量控制

2. TCP 通信模型



3. TCP 客户端

3.1. 流程

对于 TCP 客户端编程流程，有点类似于打电话过程：

1. 找个可以通话的手机（socket()）
2. 拨通对方号码并确定对方是自己要找的人（ connect()）
3. 主动聊天（ send()）
4. 或者，接收对方的回话（ recv()）
5. 通信结束后，双方说再见挂电话（close()）

3.2. 源码

```
import socket
```

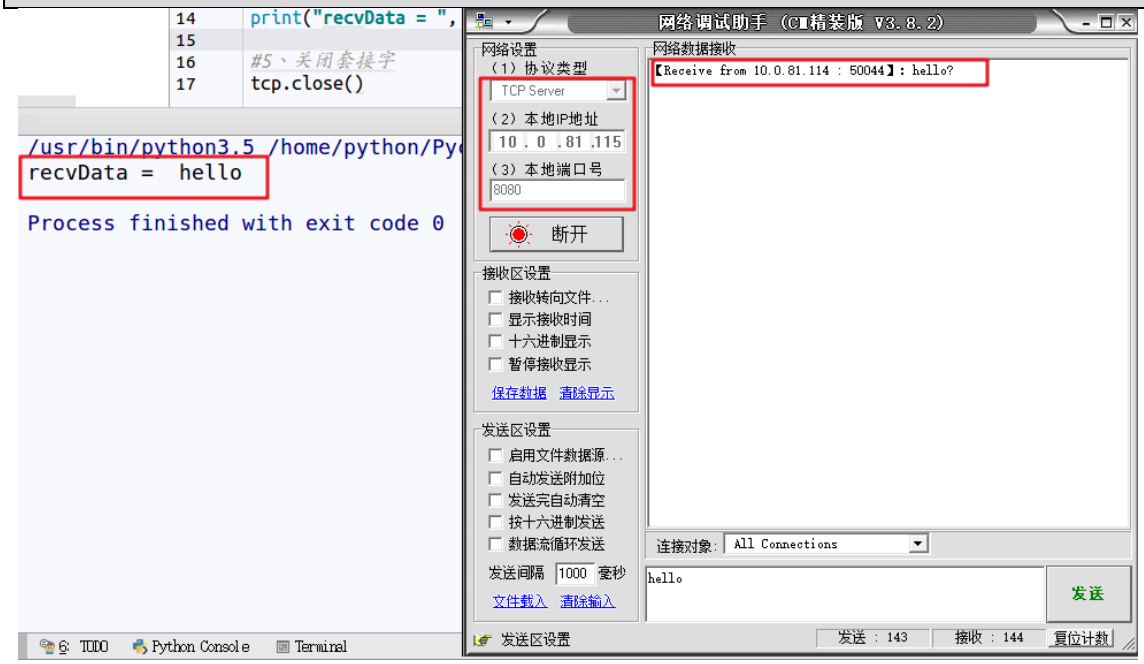
```python
#1、创建 TCP 套接字
tcp = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

#2、链接服务器
tcp.connect(("10.0.81.115", 8080))

#3、发送数据
tcp.send("are u ok?".encode("utf-8"))

#4、接收数据，最大一次接收 1024 个字节
recvData = tcp.recv(1024)
print("recvData = ", recvData.decode("utf-8"))

#5、关闭套接字
tcp.close()
```



4. TCP 编程：服务器

4.1. TCP 服务器需要具备的条件

- 具备一个可以确知的地址（（bind()））：相当于我们要明确知道移动客服的号码，才能给他们电话

- 让操作系统知道是一个服务器， 而不是客户端（ （ listen() ） ）：相当于移动的客服，他们主要的职责是被动接听用户电话，而不是主动打电话骚扰用户
- （ 等待连接的到来（ accept() ） ）：移动客服时刻等待着，来一个客户接听一个

4.2. 举例

对于 TCP 服务器编程流程，有点类似于接电话过程：

1. 找个可以通话的手机（socket() ）
2. 插上电话卡固定一个号码（ bind() ）
3. 职责为被动接听，给手机设置一个铃声来监听是否有来电（ listen() ）
4. 有来电，确定双方的关系后，才真正接通不挂电话（ accept() ）
5. 接听对方的诉说（ recv() ）
6. 适当给些回话（ send() ）
7. 通信结束后，双方说再见挂电话（ close() ）

4.3. 源码

```
import socket

#1、创建 TCP 套接字
tcp = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

#2、绑定本地网络信息
tcp.bind(('', 8080))

#3、使用 socket 创建的套接字默认的属性是主动的，使用 listen 将其变为被动的，这样就可以接收别人的链接了
tcp.listen(128)

#4、如果有新的客户端来链接服务器，那么就产生一个新的套接字专门为这个客户端服务
# clientSocket 用来为这个客户端服务
# tcp 专门等待其他新客户端的链接
clientSocket, clientAddr = tcp.accept()

#5、接收对方发送过来的数据，一次最大接收 1024 个字节
recvData = clientSocket.recv(1024)
```

```
print(clientAddr, " >>>>>>> ", recvData.decode("utf-8"))

#6、发送一些数据到客户端
clientSocket.send("thanks".encode("utf-8"))

#7、关闭为这个客户端服务的套接字，只要关闭了，就意味着为不能再为这个客户端服务了，如果还需要服务，只能再次重新连接
clientSocket.close()
```

5. TCP 的三次握手和四次挥手

5.1. TCP 连接的建立(三次握手)

在 TCP/IP 协议中，TCP 协议提供可靠的连接服务，采用三次握手建立一个连接。

最开始的时候客户端和服务器都是处于 CLOSED 状态。主动打开连接的为客户端，被动打开连接的是服务器。

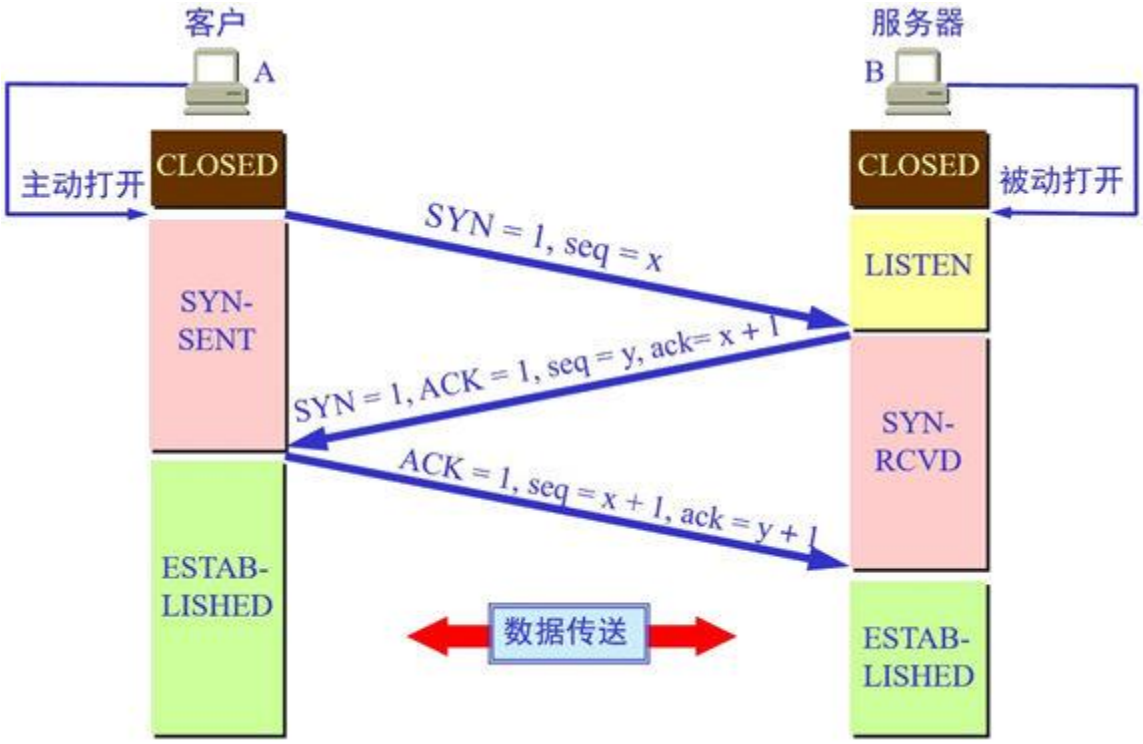
1. TCP 服务器进程先创建传输控制块 TCB，时刻准备接受客户进程的连接请求，此时服务器就进入了 LISTEN（监听）状态；

2. TCP 客户进程也是先创建传输控制块 TCB，然后向服务器发出连接请求报文，这是报文首部中的同部位 SYN=1，同时选择一个初始序列号 seq=x ，此时，TCP 客户端进程进入了 SYN-SENT（同步已发送状态）状态。TCP 规定，SYN 报文段（SYN=1 的报文段）不能携带数据，但需要消耗掉一个序号。

3. TCP 服务器收到请求报文后，如果同意连接，则发出确认报文。确认报文中应该 ACK=1，SYN=1，确认号是 ack=x+1，同时也要为自己初始化一个序列号 seq=y，此时，TCP 服务器进程进入了 SYN-RCVD（同步收到）状态。这个报文也不能携带数据，但是同样要消耗一个序号。

4. TCP 客户进程收到确认后，还要向服务器给出确认。确认报文的 ACK=1，ack=y+1，自己的序列号 seq=x+1，此时，TCP 连接建立，客户端进入 ESTABLISHED（已建立连接）状态。TCP 规定，ACK 报文段可以携带数据，但是如果不携带数据则不消耗序号。

5. 当服务器收到客户端的确认后也进入 ESTABLISHED 状态，此后双方就可以开始通信了。



为什么 TCP 客户端最后还要发送一次确认呢?

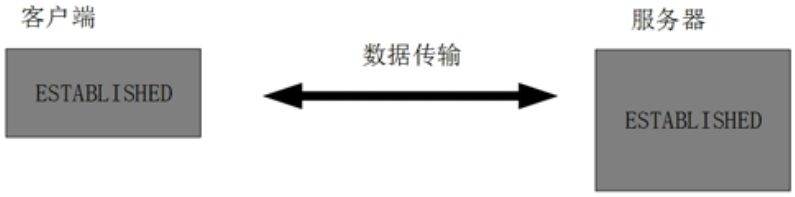
一句话，主要防止已经失效的连接请求报文突然又传送到了服务器，从而产生错误。

如果使用的是两次握手建立连接，假设有这样一种场景，客户端发送了第一个请求连接并且没有丢失，只是因为在网络结点中滞留的时间太长了，由于 TCP 的客户端迟迟没有收到确认报文，以为服务器没有收到，此时重新向服务器发送这条报文，此后客户端和服务器经过两次握手完成连接，传输数据，然后关闭连接。此时此前滞留的那一次请求连接，网络通畅了到达了服务器，这个报文本该是失效的，但是，两次握手的机制将会让客户端和服务器再次建立连接，这将导致不必要的错误和资源的浪费。

如果采用的是三次握手，就算是那一次失效的报文传送过来了，服务端接受到了那条失效报文并且回复了确认报文，但是客户端不会再次发出确认。由于服务器收不到确认，就知道客户端并没有请求连接。

5.2. TCP 连接的释放（四次挥手）

由于 TCP 连接是全双工的，因此每个方向都必须单独进行关闭。这好比，我们打电话（全双工），正常的情况下（出于礼貌），通话的双方都要说再见后才能挂电话，保证通信双方都把话说完了才挂电话。



数据传输完毕后，双方都可释放连接。最开始的时候，客户端和服务器都是处于 ESTABLISHED 状态，然后客户端主动关闭，服务器被动关闭。

1. 客户端进程发出连接释放报文，并且停止发送数据。释放数据报文首部，FIN=1，其序列号为 seq=u（等于前面已经传送过来的数据的最后一个字节的序号加 1），此时，客户端进入 FIN-WAIT-1（终止等待 1）状态。 TCP 规定，FIN 报文段即使不携带数据，也要消耗一个序号。

2. 服务器收到连接释放报文，发出确认报文，ACK=1，ack=u+1，并且带上自己的序列号 seq=v，此时，服务端就进入了 CLOSE-WAIT（关闭等待）状态。TCP 服务器通知高层的应

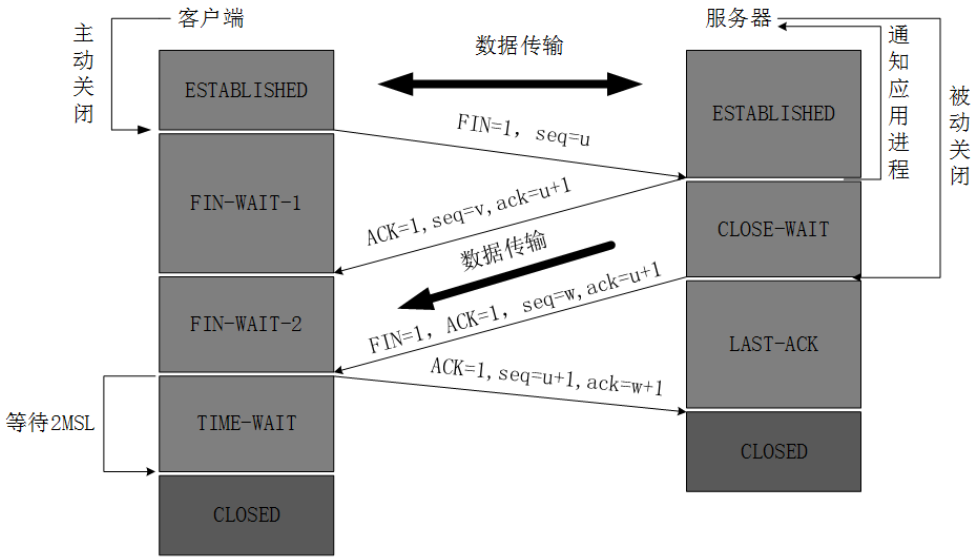
用进程，客户端向服务器的方向就释放了，这时候处于半关闭状态，即客户端已经没有数据要发送了，但是服务器若发送数据，客户端依然要接受。这个状态还要持续一段时间，也就是整个 CLOSE-WAIT 状态持续的时间。

3. 客户端收到服务器的确认请求后，此时，客户端就进入 FIN-WAIT-2（终止等待 2）状态，等待服务器发送连接释放报文（在这之前还需要接受服务器发送的最后的数据）。

4. 服务器将最后的数据发送完毕后，就向客户端发送连接释放报文，FIN=1，ack=u+1，由于在半关闭状态，服务器很可能又发送了一些数据，假定此时的序列号为 seq=w，此时，服务器就进入了 LAST-ACK（最后确认）状态，等待客户端的确认。

5. 客户端收到服务器的连接释放报文后，必须发出确认，ACK=1，ack=w+1，而自己的序列号是 seq=u+1，此时，客户端就进入了 TIME-WAIT（时间等待）状态。注意此时 TCP 连接还没有释放，必须经过 2*MSL（最长报文段寿命）的时间后，当客户端撤销相应的 TCB 后，才进入 CLOSED 状态。

6. 服务器只要收到了客户端发出的确认，立即进入 CLOSED 状态。同样，撤销 TCB 后，就结束了这次的 TCP 连接。可以看到，服务器结束 TCP 连接的时间要比客户端早一些。



5.3. 为什么客户端最后还要等待 2MSL?

MSL（Maximum Segment Lifetime），TCP 允许不同的实现可以设置不同的 MSL 值。

第一，保证客户端发送的最后一个 ACK 报文能够到达服务器，因为这个 ACK 报文可能丢失，站在服务器的角度看来，我已经发送了 FIN+ACK 报文请求断开了，客户端还没有给我回应，应该是我发送的请求断开报文它没有收到，于是服务器又会重新发送一次，而客户端就能在这个 2MSL 时间段内收到这个重传的报文，接

着给出回应报文，并且会重启 2MSL 计时器。
第二，防止类似与“三次握手”中提到了的“已经失效的连接请求报文段”出现在本连接中。客户端发送完最后一个确认报文后，在这个 2MSL 时间中，就可以使本连接持续的时间内所产生的所有报文段都从网络中消失。这样新的连接中不会出现旧连接的请求报文。

5.4. 为什么建立连接是三次握手，关闭连接确是四次挥手呢?

建立连接的时候， 服务器在 LISTEN 状态下，收到建立连接请求的 SYN 报文后，把 ACK 和 SYN 放在一个报文里发送给客户端。
而关闭连接时，服务器收到对方的 FIN 报文时，仅仅表示对方不再发送数据了但是还能接收数据，而自己也未必全部数据都发送给对方了，所以己方可以立即关闭，也可以发送一些数据给对方后，再发送 FIN 报文给对方来表示同意现在关闭连接，因此，己方 ACK 和 FIN 一般都会分开发送，从而导致多了一次。

6. 案例：文件下载

6.1. 客户端

```
import socket

def main():
    # 1. 创建套接字
    tcpSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # 2. 链接服务器
    serverIP = input("请输入服务器的 ip:")
    serverPort = input("请输入服务器的 port:")
    tcpSocket.connect((serverIP, int(serverPort)))

    # 3. 发送下载文件的请求
    fileName = input("请输入要下载的文件名字：")
    tcpSocket.send(fileName.encode("utf-8"))

    # 4. 接收文件的数据并且保存到文件中, 假设文件的长度不会超过 1024
    recvData = tcpSocket.recv(1024)
    if recvData:
        with open("[新]" + fileName, "wb") as f:
            #不管在这里是否产生异常，那么 with 这个语句，一定会保证调用 f.close()
            f.write(recvData)
        print("下载成功...ok")
    else:
        print("下载失败...error")
```

```
    # 5. 关闭套接字
    tcpSocket.close()


if __name__ == "__main__":
    main()
```

6.2. 服务器

```
import socket

def getFileContent(fileName):
    try:
        with open(fileName, "rb") as f:
            fileContent = f.read()
        return fileContent
    except Exception as ret:
        print("下载的文件%s 不存在...." % fileName)

def main():
    # 1. 创建套接字
    tcpSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # 2. 绑定
    tcpSocket.bind(("", 8080))

    # 3. 监听套接字改为被动套接字
    tcpSocket.listen(128)

    # 4. 等待客户端的到来
    while True:
        clientSocket, clientAddr = tcpSocket.accept()
        print("一个新的客户端进行了链接: %s" % str(clientAddr))

        # 5. 接收客户端发送过来的文件下载请求（文件名）
        fileName = clientSocket.recv(1024)
        fileName = fileName.decode("utf-8")
        print("需要下载的文件名是：%s" % fileName)

        # 6. 获取文件中的数据
```

```
        fileContent = getFileContent(fileName)

        if fileContent:
            # 7. 发送文件数据给客户端
            clientSocket.send(fileContent)
            print("发送成功...ok")
        else:
            print("发送失败...error")

        # 8. 关闭套接字
        clientSocket.close()

    tcpSocket.close()

if __name__ == "__main__":
    main()
```