

# 乐字节教育高级架构课程

正所谓“授人以鱼不如授人以渔”，你们想要的 **Java 学习资料** 来啦！

不管你是学生，还是已经步入职场的同行，希望你们都要珍惜眼前的学习机会，奋斗没有终点，知识永不过时。

扫描下方二维码即可领取



乐字节官方交流群

## socket 编程

### 一、客户端/服务器架构

C/S=Client/Server

B/S=Brower/Server

一个基于客户端/服务器，一个基于浏览器/服务器

互联网中处处是 C/S 架构，如电影网站是服务端，你的浏览器是客户端（B/S 架构也是 C/S 架构的一种）

C/S 架构与 socket 的关系：

我们学习 socket 就是为了完成 C/S 架构的开发

### 二、osi 五层|七层

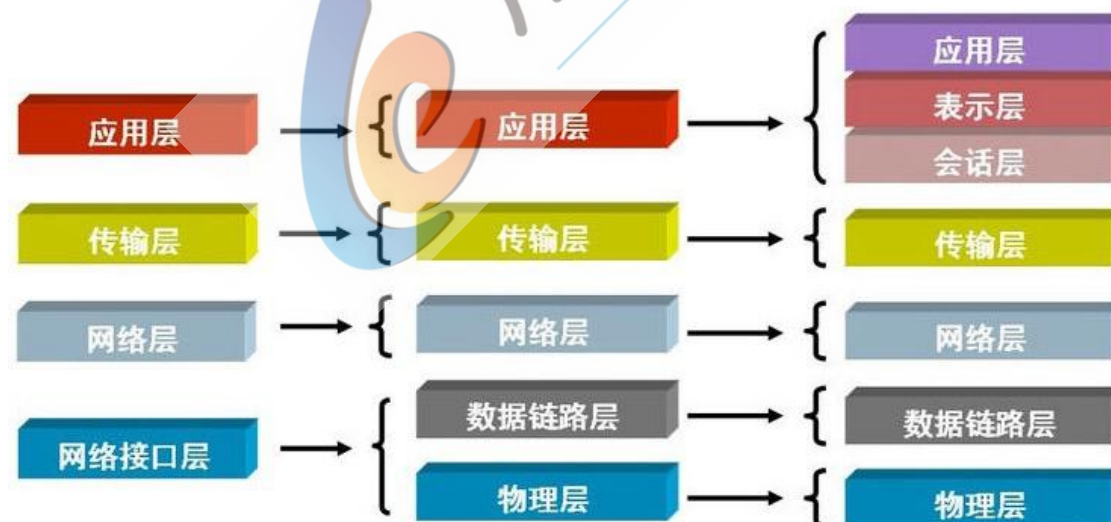
一个完整的计算机系统是由硬件、操作系统、应用软件三者组成，具备了这三个条件，一台计算机系统就可以自己跟自己玩了（打个单机游戏，玩个扫雷啥的）

如果你要跟别人一起玩，那你就需要上网了，什么是互联网？

互联网的核心就是由一堆协议组成，协议就是标准，比如全世界人通信的标准是英语

如果把计算机比作人，互联网协议就是计算机界的英语。所有的计算机都学会了互联网协议，那所有的计算机都就可以按照统一的标准去收发信息从而完成通信了。

人们按照分工不同把互联网协议从逻辑上划分了层级



每层都运行特定的协议，越往上越靠近用户，越往下越靠近硬件

用户感知到的只是最上面一层应用层，自上而下每层都依赖于下一层，所以我们从最下一层开始切入，比较好理解

1. **物理层由来：**上面提到，孤立的计算机之间要想一起玩，就必须接入 internet，言外之意就是计算机之间必须完成组网

**物理层功能：**主要是基于电器特性发送高低电压(电信号)，高电压对应数字 1，低电压对应数字 0

2. **数据链路层由来**：单纯的电信号 0 和 1 没有任何意义，必须规定电信号多少位一组，每组什么意思

**数据链路层的功能**：定义了电信号的分组方式

**以太网协议**：

早期的时候各个公司都有自己的分组方式，后来形成了统一的标准，即以太网协议 ethernet

**mac 地址**：每块网卡出厂时都被烧制上一个世界唯一的 mac 地址，长度为 48 位 2 进制，通常由 12 位 16 进制数表示（前六位是厂商编号，后六位是流水线号）

**广播**：有了 mac 地址，同一网络内的两台主机就可以通信了（一台主机通过 arp 协议获取另外一台主机的 mac 地址）ethernet 采用最原始的方式，广播的方式进行通信，即计算机通信基本靠吼

3. **网络层由来**：有了 ethernet、mac 地址、广播的发送方式，世界上的计算机就可以彼此通信了，问题是世界范围的互联网是由一个个彼此隔离的小的局域网组成的，那么如果所有的通信都采用以太网的广播方式，那么一台机器发送的包全世界都会收到，这就不仅仅是效率低的问题了，这会是一种灾难

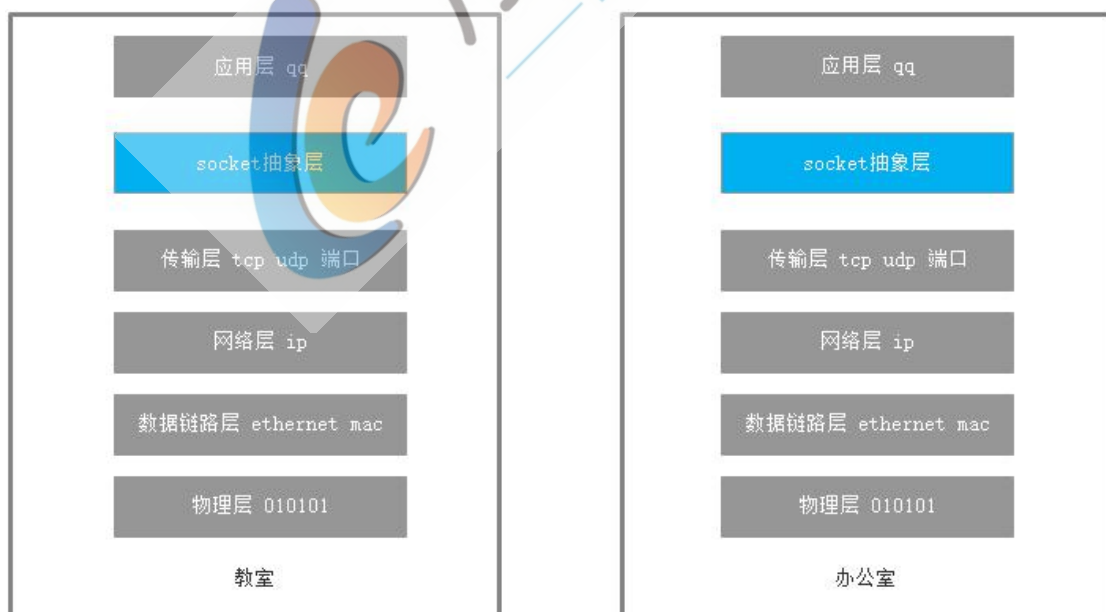
**网络层功能**：引入一套新的地址用来区分不同的广播域 / 子网，这套地址即网络地址

4. **传输层的由来**：网络层的 ip 帮我们区分子网，以太网层的 mac 帮我们找到主机，然后大家使用的都是应用程序，你的电脑上可能同时开启 qq，暴风影音，等多个应用程序，那么我们通过 ip 和 mac 找到了一台特定的主机，如何标识这台主机上的应用程序，答案就是端口，端口即应用程序与网卡关联的编号。

**传输层功能**：建立端口到端口的通信

5. **应用层由来**：用户使用的都是应用程序，均工作于应用层，互联网是开发的，大家都可以开发自己的应用程序，数据多种多样，必须规定好数据的组织形式

**应用层功能**：规定应用程序的数据格式



### 三、socket 是什么？

我们经常把 socket 翻译为**套接字**，Socket 是应用层与 TCP/IP 协议族通信的中间软件抽象层，它是一组接口。在设计模式中，Socket 其实就是一个门面模式，它把复杂的

TCP/IP 协议族隐藏在 Socket 接口后面, 对用户来说, 一组简单的接口就是全部, 让 Socket 去组织数据, 以符合指定的协议。

所以, 我们无需深入理解 tcp/udp 协议, socket 已经为我们封装好了, 我们只需要遵循 socket 的规定去编程, 写出的程序自然就是遵循 tcp/udp 标准的。

也有人将 socket 说成 ip+port, ip 是用来标识互联网中的一台主机的位置, 而 port 是用来标识这台机器上的一个应用程序, ip 地址是配置到网卡上的, 而 port 是应用程序开启的, ip 与 port 的绑定就标识了互联网中独一无二的一个应用程序。而程序的 pid 是同一台机器上不同进程或者线程的标识

## 四、套接字

套接字起源于 20 世纪 70 年代加利福尼亚大学伯克利分校版本的 Unix。一开始, 套接字被设计用在同一台主机上多个应用程序之间的通讯。这也被称进程间通讯, 或 IPC。套接字有两种 (或者称为有两个种族, 分别是基于文件型的和基于网络型的)。

基于文件类型的套接字家族

套接字家族的名字: AF\_UNIX

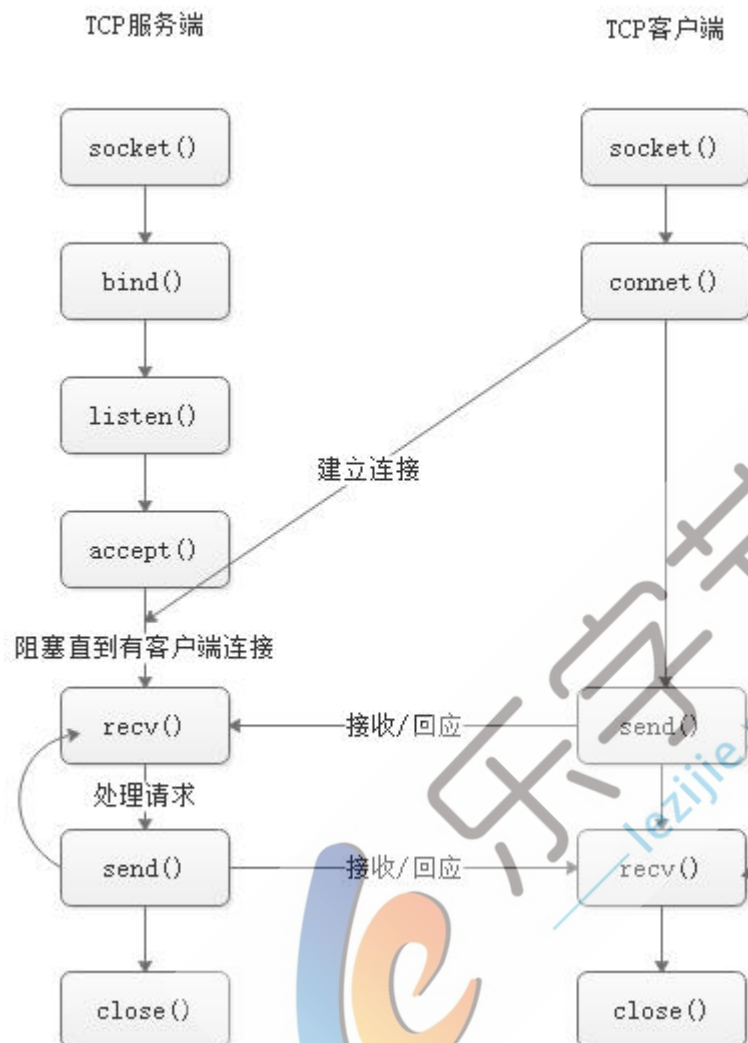
unix 一切皆文件, 基于文件的套接字调用的就是底层的文件系统来取数据, 两个套接字进程运行在同一机器, 可以通过访问同一个文件系统间接完成通信

基于网络类型的套接字家族

套接字家族的名字: AF\_INET

(还有 AF\_INET6 被用于 ipv6, 还有一些其他的地址家族, 不过, 他们要么是只用于某个平台, 要么就是已经被废弃, 或者是很少被使用, 或者是根本没有实现, 所有地址家族中, AF\_INET 是使用最广泛的一个, python 支持很多种地址家族, 但是由于我们只关心网络编程, 所以大部分时候我只使用 AF\_INET)

## 1、套接字工作流程



工作流程：服务器端先初始化 Socket，然后与端口绑定(bind)，对端口进行监听(listen)，调用 `accept()` 阻塞，等待客户端连接。在这时如果有个客户端初始化一个 Socket，然后连接服务器(connect)，如果连接成功，这时客户端与服务器端的连接就建立了。客户端发送数据请求，服务器端接收请求并处理请求，然后把回应数据发送给客户端，客户端读取数据，最后关闭连接，一次交互结束。

tcp 的整个流程类似打电话的一个过程：

服务端：

- 买手机 `tcp_server=socket.socket(socket.AF_INET, socket.SOCK_STREAM)`
- 绑定电话卡 `tcp_server.bind(("ip", 端口))`
- 待机 `tcp_server.listen(最大链接数)`
- 接听电话 `conn, addr = tcp_server.accept()` 得到链接和对方地址
- 接收消息，听话 `data = conn.recv(接收字节数)`
- 发消息，说话 `conn.send(发送的是字节数据需要编码)`
- 挂电话 `conn.close()`

h. 手机关机 `tcp_server.close()`

客户端:

- 买手机 `tcp_client=socket.socket(socket.AF_INET, socket.SOCK_STREAM)`
- 拨号 `tcp_client.connect(("服务端 ip", 服务端端口))`
- 发消息, 说话 `tcp_client.send(发送的是字节数据需要编码)`
- 接收消息, 听话 `data = tcp_client.recv(1024)`
- 挂电话 `tcp_client.close()`

## 2、基于 tcp 协议的套接字编程

tcp 是基于链接的, 必须先启动服务端, 然后再启动客户端去链接服务端  
文件名不可以是 `socket.py`

`tcp_server.py`

```
# -*- coding:utf-8 -*-
import socket # 导入工具

# 建立 socket 连接 socket.AF_INET -> 基于网络类型 socket.SOCK_STREAM -> tcp 协议
tcp_server = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # 买手机

# 声明 socket 服务端的 ip 和端口
tcp_server.bind(("127.0.0.1", 8000)) # 类型必须为元组 # SIM 卡

# 最大连接数
tcp_server.listen(5) # 待机中

# 等待客户端的连接
conn, addr = tcp_server.accept() # 三次握手 # 接听电话

# 接收客户端的消息
data = conn.recv(1024) # 听消息, 听话
print('客户端发来的消息是: ', data.decode('utf-8'))

# 服务端发送消息给客户端
conn.send('我是服务端...'.encode('utf-8')) # 发消息, 说话

# 关闭连接
conn.close() # 四次挥手 # 挂电话
tcp_server.close() # 手机关机
```

`tcp_client.py`

```
# -*- coding:utf-8 -*-
import socket
```



```
# 建立 socket 连接
tcp_client = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # 买手机

# 与服务端建立连接
tcp_client.connect(("127.0.0.1", 8000)) # 类型必须为元组 # 拨号

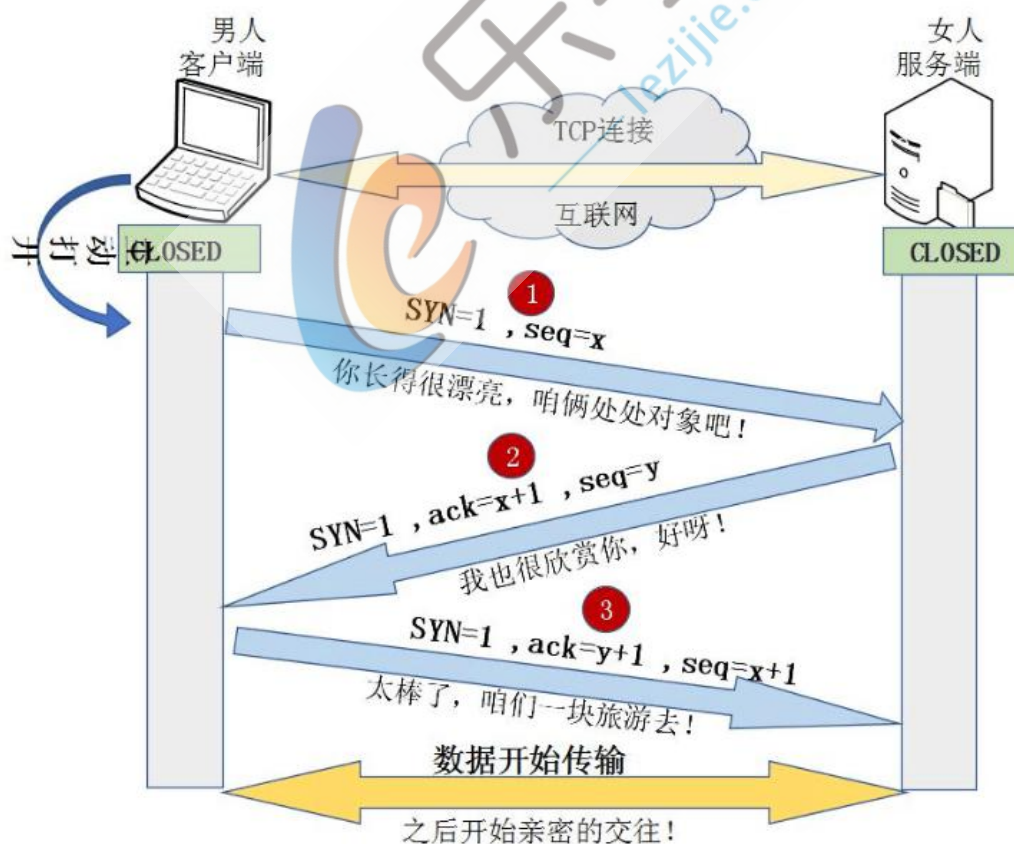
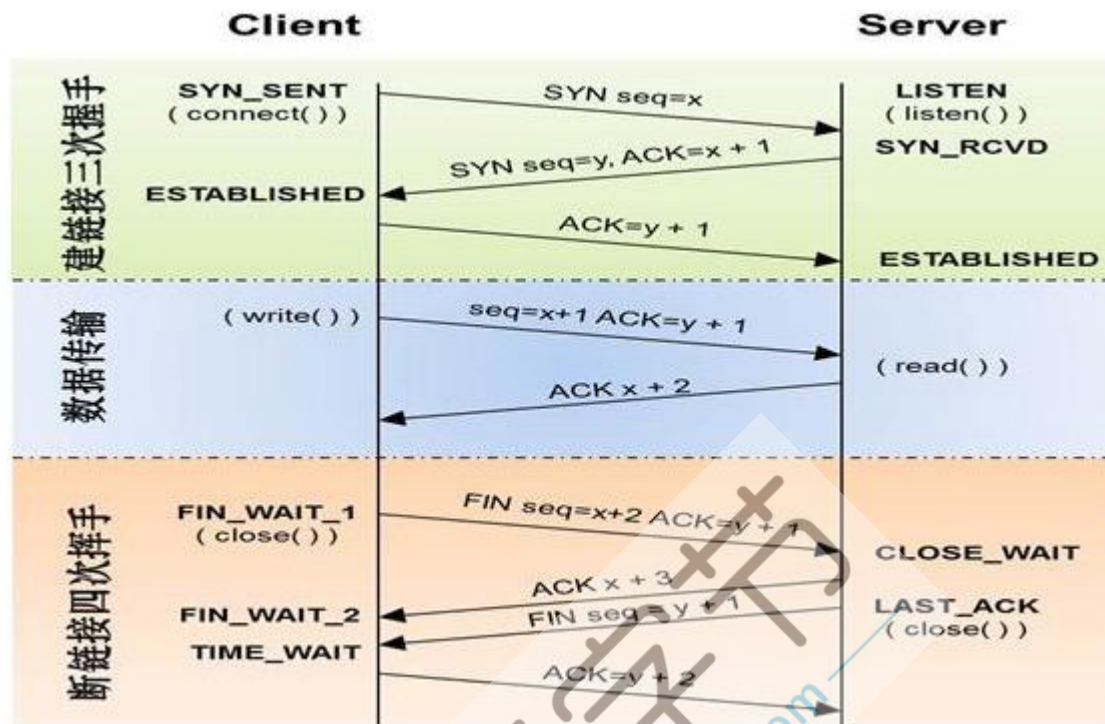
# 客户端发送消息给服务端
tcp_client.send('hello'.encode('utf-8')) # 发消息, 说话

# 接收服务端的消息
data = tcp_client.recv(1024) # 听消息, 听话
print('服务端发来的消息是: ', data.decode('utf-8'))

# 关闭连接
tcp_client.close() # 挂电话
```



## 2.1、tcp 协议的三次握手和四次挥手



三次握手过程说明:

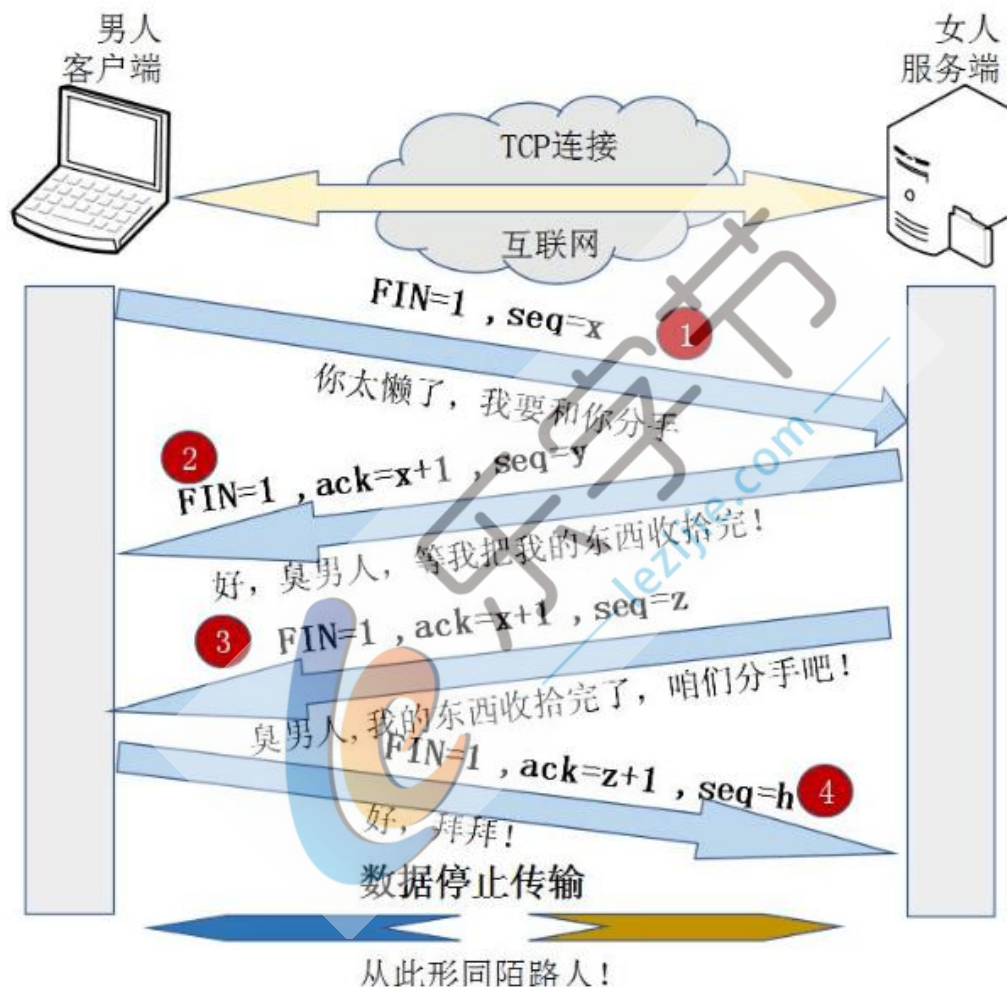
1、由客户端发送建立 TCP 连接的请求报文, 其中报文中包含 seq 序列号, 是由发送端随机



生成的，并且将报文中的 SYN 字段置为 1，表示需要建立 TCP 连接。(SYN=1, seq=x, x 为随机生成数值)

2、由服务端回复客户端发送的 TCP 连接请求报文，其中包含 seq 序列号，是由回复端随机生成的，并且将 SYN 置为 1，而且会产生 ACK 字段，ACK 字段数值是在客户端发送过来的序列号 seq 的基础上加 1 进行回复，以便客户端收到信息时，知晓自己的 TCP 建立请求已得到验证。(SYN=1, ACK=x+1, seq=y, y 为随机生成数值) 这里的 ack 加 1 可以理解为是确认和谁建立连接。

3、客户端收到服务端发送的 TCP 建立验证请求后，会使自己的序列号加 1 表示，并且再次回复 ACK 验证请求，在服务端发过来的 seq 上加 1 进行回复。(SYN=1, ACK=y+1, seq=x+1)



#### 四次挥手过程说明:

1、客户端发送断开 TCP 连接请求的报文，其中报文中包含 seq 序列号，是由发送端随机生成的，并且还将报文中的 FIN 字段置为 1，表示需要断开 TCP 连接。(FIN=1, seq=x, x 由客户端随机生成)

2、服务端会回复客户端发送的 TCP 断开请求报文，其包含 seq 序列号，是由回复端随机生成的，而且会产生 ACK 字段，ACK 字段数值是在客户端发过来的 seq 序列号基础上加 1 进行回复，以便客户端收到信息时，知晓自己的 TCP 断开请求已经得到验证。(FIN=1, ACK=x+1, seq=y, y 由服务端随机生成)

3、服务端在回复完客户端的 TCP 断开请求后，不会马上进行 TCP 连接的断开，服务端会先确保断开前，所有传输到 A 的数据是否已经传输完毕，一旦确认传输数据完毕，就会将回复

报文的 FIN 字段置 1，并且产生随机 seq 序列号。(FIN=1, ACK=x+1, seq=z, z 由服务端随机生成)

4、客户端收到服务端的 TCP 断开请求后，会回复服务端的断开请求，包含随机生成的 seq 字段和 ACK 字段，ACK 字段会在服务端的 TCP 断开请求的 seq 基础上加 1，从而完成服务端请求的验证回复。(FIN=1, ACK=z+1, seq=h, h 为客户端随机生成)

至此 TCP 断开的 4 次挥手过程完毕

#### 11 种状态：

LISTEN：等待从任何远端 TCP 和端口的连接请求。

SYN\_SENT：发送完一个连接请求后等待一个匹配的连接请求。

SYN\_RECEIVED：发送连接请求并且接收到匹配的连接请求以后等待连接请求确认。

ESTABLISHED：表示一个打开的连接，接收到的数据可以被投递给用户。连接的数据传输阶段的正常状态。

FIN\_WAIT\_1：等待远端 TCP 的连接终止请求，或者等待之前发送的连接终止请求的确认。

FIN\_WAIT\_2：等待远端 TCP 的连接终止请求。

CLOSE\_WAIT：等待本地用户的连接终止请求。

CLOSING：等待远端 TCP 的连接终止请求确认。

LAST\_ACK：等待先前发送给远端 TCP 的连接终止请求的确认（包括它字节的连接终止请求的确认）

TIME\_WAIT：等待足够的时间过去以确保远端 TCP 接收到它的连接终止请求的确认。

TIME\_WAIT 两个存在的理由：

1. 可靠的实现 tcp 全双工连接的终止；
2. 允许老的重复分节在网络中消逝。

CLOSED：不在连接状态（这是为方便描述假想的状态，实际不存在）

## 2.2、服务端客户端循环收发消息

tcp\_server\_while.py

```
# -*- coding:utf-8 -*-  
import socket # 导入工具  
  
# 建立 socket 连接 socket.AF_INET -> 基于网络类型 socket.SOCK_STREAM -> tcp 协议  
tcp_server = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # 买手机
```

```
# 声明 socket 服务端的 ip 和端口
tcp_server.bind(("127.0.0.1", 8000)) # 类型必须为元组 # SIM 卡

back_log = 5
buffer_size = 1024
# 最大连接数
tcp_server.listen(back_log) # 待机中

# 外层循环保证服务端无限循环(链接循环)
while True:
    # 等待客户端的连接 阻塞
    print('服务端开始运行了')
    conn, addr = tcp_server.accept() # 三次握手 # 接听电话
    print(conn)
    print(addr)

    # 内层循环保证通讯无限循环(通信循环)
    while True:
        # windows 强制关闭客户端会抛出异常, 主动解决异常
        # linux 强制关闭客户端会不停返回空, 服务端进入死循环
        try:
            # 接收客户端的消息
            data = conn.recv(buffer_size) # 听消息, 听话

            # 如果客户端发来的是空字符则表示客户端已关闭, 那么关闭服务端与该客户端连接
            if len(data) == 0: break
            print('客户端发来的消息是: ', data.decode('utf-8'))

            # 如果客户端发来的是指定字符则关闭
            if 'exit'.encode('utf-8') == data:
                conn.send('exit'.encode('utf-8'))
                break

            # 服务端发送消息给客户端
            msg = input('>>>: ').strip()
            conn.send(msg.encode('utf-8')) # 发消息, 说话
        except ConnectionResetError:
            print('客户端异常断开连接')
            break

    # 关闭连接
    conn.close() # 四次挥手 # 挂电话

tcp_server.close() # 手机关机
```

```
tcp_client_while.py
# -*- coding:utf-8 -*-
import socket

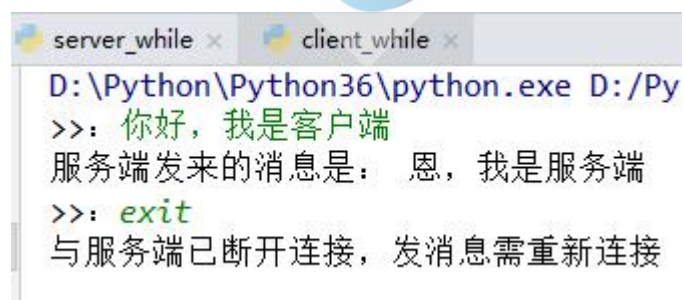
# 建立 socket 连接
tcp_client = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # 买手机

# 与服务端建立连接
tcp_client.connect(("127.0.0.1", 8000)) # 类型必须为元组 # 拨号

buffer_size = 1024
while True:
    # 客户端发送消息给服务端
    msg = input('>>: ').strip()
    # 如果客户端发送空字符则重新发送
    if 0 == len(msg): continue
    tcp_client.send(msg.encode('utf-8')) # 发消息, 说话

    # 接收服务端的消息
    data = tcp_client.recv(buffer_size) # 听消息, 听话
    # 如果服务端发送指定字符则关闭
    if 'exit'.encode('utf-8') == data:
        print('与服务端已断开连接, 发消息需重新连接')
        break
    print('服务端发来的消息是: ', data.decode('utf-8'))

# 关闭连接
tcp_client.close() # 挂电话
```



```
server_while x client_while x
D:\Python\Python36\python.exe D:/Py
>>: 你好, 我是客户端
服务端发来的消息是: 恩, 我是服务端
>>: exit
与服务端已断开连接, 发消息需重新连接
```

```
server_while x client_while x
D:\Python\Python36\python.exe D:/Pycl
服务端开始运行了
<socket.socket fd=124, family=Address
('127.0.0.1', 5830)
客户端发来的消息是: 你好, 我是客户端
>>: 恩, 我是服务端
```

## 2.3、总结

**tcp** 是基于链接的, 必须先启动服务端, 然后再启动客户端去链接服务端  
**tcp** 的 `recv()` 和 `send()` 没有对应关系, 都是从各自的缓冲区进行操作

### tcp 服务端

```
ss = socket() #创建服务器套接字
ss.bind()     #把地址绑定到套接字
ss.listen()   #监听链接
inf_loop:    #服务器无限循环
    cs = ss.accept() #接受客户端链接
    comm_loop:      #通讯循环
        cs.recv()/cs.send() #对话(接收与发送)
    cs.close()      #关闭客户端套接字
ss.close()         #关闭服务器套接字(可选)
```

### tcp 客户端

```
cs = socket() # 创建客户套接字
cs.connect()  # 尝试连接服务器
comm_loop:   # 通讯循环
    cs.send()/cs.recv() # 对话(发送/接收)
cs.close()   # 关闭客户套接字
```

## 3、基于 udp 协议的套接字编程

**udp** 是无链接的, 先启动哪一端都不会报错, QQ 就是 **udp** 协议为主开发的  
**udp\_server.py**

```
# -*- coding:utf-8 -*-
from socket import *

ip_port = ("127.0.0.1", 8000)
buffer_size = 1024
```



```
# 建立 socket 连接 socket.AF_INET -> 基于网络类型 socket.SOCK_DGRAM -> udp 协议
udp_server = socket(AF_INET, SOCK_DGRAM)

# 声明 socket 服务端的 ip 和端口
udp_server.bind(ip_port)

while True:
    data, addr = udp_server.recvfrom(buffer_size) # 接收的是元组
    print(data.decode('utf-8'), addr)

    udp_server.sendto(data, addr)

udp_server.close()
```

udp\_client.py

```
# -*- coding:utf-8 -*-
from socket import *

ip_port = ("127.0.0.1", 8000)
buffer_size = 1024

# 建立 socket 连接
udp_client = socket(AF_INET, SOCK_DGRAM)

while True:
    msg = input('>>: ').strip()
    udp_client.sendto(msg.encode('utf-8'), ip_port)

    data, addr = udp_client.recvfrom(buffer_size)
    print(data.decode('utf-8'), addr)

udp_client.close()
```



```
D:\Python\Python36\python.exe D:/Pycho
>>: 你好我是张三
你好我是张三 ('127.0.0.1', 8000)
>>:
```

```

udp_server x udp_client x udp_client2 x
D:\Python\Python36\python.exe D:/PycharmProj
>>: 我是李四, 很高兴认识大家
我是李四, 很高兴认识大家 ('127.0.0.1', 8000)
>>:

udp_server x udp_client x udp_client2 x
D:\Python\Python36\python.exe D:/PycharmProj
你好我是张三 ('127.0.0.1', 64603)
我是李四, 很高兴认识大家 ('127.0.0.1', 64604)

```

### 3.1、udp 实现 ntp 时间服务

```

ntp_server.py
# -*- coding:utf-8 -*-
from socket import *
import time

ip_port = ("127.0.0.1", 8000)
buffer_size = 1024

# 建立 socket 连接 socket.AF_INET -> 基于网络类型 socket.SOCK_DGRAM -> udp 协议
udp_server = socket(AF_INET, SOCK_DGRAM)

# 声明 socket 服务端的ip 和端口
udp_server.bind(ip_port)

while True:
    data, addr = udp_server.recvfrom(buffer_size) # 接收的是元组

    if not data:
        fmt = '%Y-%m-%d %X'
    else:
        fmt = data.decode('utf-8')

    time_msg = 'ntp 服务器的标准时间是: ' + time.strftime(fmt)
    udp_server.sendto(time_msg.encode('utf-8'), addr)

udp_server.close()

ntp_client.py
# -*- coding:utf-8 -*-

```

```
from socket import *

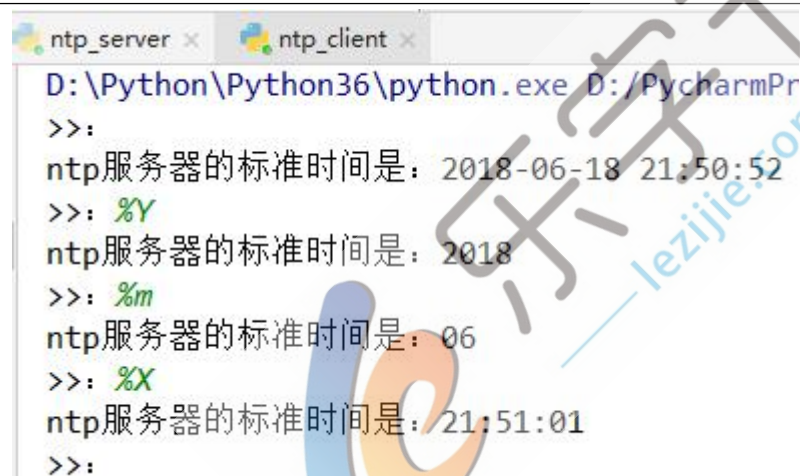
ip_port = ("127.0.0.1", 8000)
buffer_size = 1024

# 建立 socket 连接
udp_client = socket(AF_INET, SOCK_DGRAM)

while True:
    msg = input('>>: ').strip()
    udp_client.sendto(msg.encode('utf-8'), ip_port)

    data, addr = udp_client.recvfrom(buffer_size)
    print(data.decode('utf-8'))

udp_client.close()
```



```
D:\Python\Python36\python.exe D:/PycharmPr
>>:
ntp服务器的标准时间是: 2018-06-18 21:50:52
>>: %Y
ntp服务器的标准时间是: 2018
>>: %m
ntp服务器的标准时间是: 06
>>: %X
ntp服务器的标准时间是: 21:51:01
>>:
```

## 3.2、总结

udp 是无链接的，先启动哪一端都不会报错，QQ 就是 udp 协议为主开发的  
udp 一次 recvfrom() 对应一个 sendto()，缓冲区大小不一致会丢失数据或报错

### udp 服务端

```
ss = socket() # 创建一个服务器的套接字
ss.bind()    # 绑定服务器套接字
inf_loop:   # 服务器无限循环
    cs = ss.recvfrom()/ss.sendto() # 对话(接收与发送)
ss.close()  # 关闭服务器套接字
```

### udp 客户端

```
cs = socket() # 创建客户套接字
comm_loop:   # 通讯循环
    cs.sendto()/cs.recvfrom() # 对话(发送/接收)
cs.close()   # 关闭客户套接字
```

## 4、基于 tcp 实现远程执行命令

cmd\_server.py

```
# -*- coding:utf-8 -*-
from socket import *
import subprocess # 与系统交互模块

ip_port = ('127.0.0.1', 8000)
back_log = 5
buffer_size = 1024

# 创建 socket 链接 tcp 协议
cmd_server = socket(AF_INET, SOCK_STREAM)

# 绑定 ip 和端口
cmd_server.bind(ip_port)

# 声明最大链接数
cmd_server.listen(back_log)

# 外层循环_链接循环: 保证服务端永久执行
while True:
    conn, addr = cmd_server.accept() # 得到一个持续链接和客户端的地址

    # 内层循环_通信循环: 保持和客户端的持续连接
    while True:
        # 防止客户端异常中断, try 处理
        try:
            cmd = conn.recv(buffer_size) # 得到客户端的 cmd 指定

            if not cmd: break # 客户端输入空 中断该连接

            print('客户端的指令是: ', cmd.decode('utf-8'))

            # -----与系统交互代码块-----
            ...

            args 参数为: 指令序列或者字符串指令
            shell=True 参数为: True 就是指令为字符串, False 就是指令为序列
            stderr=None 参数为: 返回错误通道信息
```

```
        subprocess.PIPE 返回一个 file，可以直接给客户端展示
    stdin=None 参数为：标准输入通道
    stdout=None 参数为：返回标准通道信息
    ...
    cmd_result = subprocess.Popen(cmd.decode('utf-8'), shell=True,
                                   stderr=subprocess.PIPE,
                                   stdin=subprocess.PIPE,
                                   stdout=subprocess.PIPE)

    err_msg = cmd_result.stderr.read() # 从错误通道获取信息

    ...

    因为我们从 cmd 发送命令后，可能会返回两种结果：
    1. 错误指令，返回错误通道信息
    2. 正确指令，返回标准通道信息
        a. 指令执行成功，返回空，比如 cd . cd ..
        b. 指令执行成功，返回信息
    ...

    # 如果错误通道有消息，返回客户端显示
    if err_msg:
        msg = err_msg
    else:
        # 如果错误通道没有消息，获取标准通道
        msg = cmd_result.stdout.read() # 从标准通道获取信息

    # 执行 cd . 或者 cd .. 这样的指令，返回空，所以要处理
    if not msg:
        msg = '执行成功'.encode('gbk') # 返回客户端友好提示

    # -----与系统交互代码块-----

    # 返回标准通道信息
    # 因为返回的信息已经是 cmd 的 gbk 格式信息，所以无需再添加 encode('gbk')
    conn.send(msg)

except Exception as e:
    print('程序发生异常，异常信息为：', e)
    break

# 与客户端断开链接
conn.close()

# 关闭 socket 链接
cmd_server.close()
```



cmd\_client.py

```
# -*- coding:utf-8 -*-
from socket import *

ip_port = ('127.0.0.1', 8000)
back_log = 5
buffer_size = 1024

# 创建 socket 链接 tcp 协议
cmd_client = socket(AF_INET, SOCK_STREAM)

# 建立链接
cmd_client.connect(ip_port)

# 实现通讯循环
while True:
    cmd_msg = input('>>>: ').strip()
    if not cmd_msg: continue

    # 如果发送 exit 中断与服务端链接
    if 'exit' == cmd_msg: break
    print(cmd_msg)

    # 发送指令给服务端
    cmd_client.send(cmd_msg.encode('utf-8'))

    data = cmd_client.recv(300)
    print(data.decode('gbk'))

# 关闭链接
cmd_client.close()
```



```
cmd_server x cmd_client x
D:\Python\Python36\python.exe D:/PycharmProjects/0523/day15
>>: asd
asd
'asd' 不是内部或外部命令，也不是可运行的程序
或批处理文件。

>>: dir
dir
驱动器 D 中的卷没有标签。
卷的序列号是 E85B-4BF0

D:\PycharmProjects\0523\day15\tcp 的目录

2018/06/21 17:26 <DIR> .
2018/06/21 17:26 <DIR> ..
2018/06/21 17:26 636 cmd_client.py
2018/06/21 17:18 3,160 cmd_server.py
2018/06/21 15:12 563 tcp_client.py
2018/06/21 11:14 819 tcp_client_forever.py
2018/06/21 11:34 821 tcp_client_while.py
2018/06/21 13:44 821 tcp_client_while2.py
2018/06/21 14:46 853 tcp_server.py
2018/06/21 11:31 1,162 tcp_server_forever.py
2018/06/21 11:40 1,662 tcp_server_while.py
9 个文件 10,497 字节
2 个目录 21,947,920,384 可用字节

>>:
```

```
C:\WINDOWS\system32\cmd.exe

D:\PycharmProjects\0523\day15\tcp>asd
'asd' 不是内部或外部命令，也不是可运行的程序
或批处理文件。

D:\PycharmProjects\0523\day15\tcp>dir
驱动器 D 中的卷没有标签。
卷的序列号是 E85B-4BF0

D:\PycharmProjects\0523\day15\tcp 的目录

2018/06/21  17:26    <DIR>          .
2018/06/21  17:26    <DIR>          ..
2018/06/21  17:26                636 cmd_client.py
2018/06/21  17:18            3,160 cmd_server.py
2018/06/21  15:12            563 tcp_client.py
2018/06/21  11:14            819 tcp_client_forever.py
2018/06/21  11:34            821 tcp_client_while.py
2018/06/21  13:44            821 tcp_client_while2.py
2018/06/21  14:46            853 tcp_server.py
2018/06/21  11:31          1,162 tcp_server_forever.py
2018/06/21  11:40          1,662 tcp_server_while.py
                9 个文件             10,497 字节
                2 个目录  21,947,920,384 可用字节

D:\PycharmProjects\0523\day15\tcp>
```