



TsingHua University

Department of EE

# 《通信与网路》实验

## socket 网络编程

勾天润 无 03

2020012321

指导老师: 周盛

October 5, 2022

# Contents

<b>1</b>	<b>知识回顾</b>	<b>1</b>
1.1	socket	1
1.1.1	传输层协议	1
1.1.2	IP、端口号	1
1.1.3	服务器、客户端	1
1.2	python 部分语法回顾	1
1.2.1	with 关键字	1
1.2.2	线程	1
<b>2</b>	<b>思路、代码分析</b>	<b>3</b>
2.1	基本流程	3
2.1.1	创建套接字、绑定端口	3
2.1.2	client 请求连接、server 接受连接	4
2.1.3	消息的收发、响应	4
2.1.4	getsockname、getpeername	4
2.2	hub 聊天室	4
2.2.1	等待连接	4
2.2.2	信息的收发	5
<b>3</b>	<b>实验结果</b>	<b>6</b>
3.1	实现客户端聊天程序	6
3.2	实现一对一服务端聊天程序	6
3.3	实现聊天室服务端	6
<b>4</b>	<b>思考题</b>	<b>8</b>
4.1		8
4.2		8
4.3		8

# Chapter 1

## 知识回顾

### 1.1 socket

我们学习的网络体系结构参考模型从下到上依次为：物理层、链路层、网络层、传输层、应用层。socket 将传输层及其以下部分抽象，以便于调用。使用 socket 套接字可进行报文发送、接收。以下为 socket 的一些相关标识

#### 1.1.1 传输层协议

- TCP 可靠、面向连接
- UDP 不可靠、无连接协议

#### 1.1.2 IP、端口号

主机地址唯一标志着主机。IP 地址由 32bit 存储，端口号由 16bit 存储。

#### 1.1.3 服务器、客户端

- 服务端一直在线
- 客户端在有需求时请求与服务器请求通信

### 1.2 python 部分语法回顾

#### 1.2.1 with 关键字

with 在处理文件时很方便。with 语句实现原理建立在上下文管理器之上。上下文管理器是一个实现 `__enter__` 和 `__exit__` 方法的类。使用 with 语句确保在嵌套块的末尾调用 `__exit__` 方法。这个概念类似于 try...finally 块的使用。我有一个朋友在代码中对 socket 对象使用了 with 语句，造成该 socket 自动调用 exit 方法而关闭。

#### 1.2.2 线程

在 thread 库中存在 Thread 类，用于开线程。线程以函数为参数。args 参数必须为 tuple 类型。

```
1 import threading
2 def f(a):
3     while(a!=0):
4         print(a)
5         a=a-1
6
7 threading.Thread(target=f,args=(10,)).start()
```

## Chapter 2

# 思路、代码分析

### 2.1 基本流程

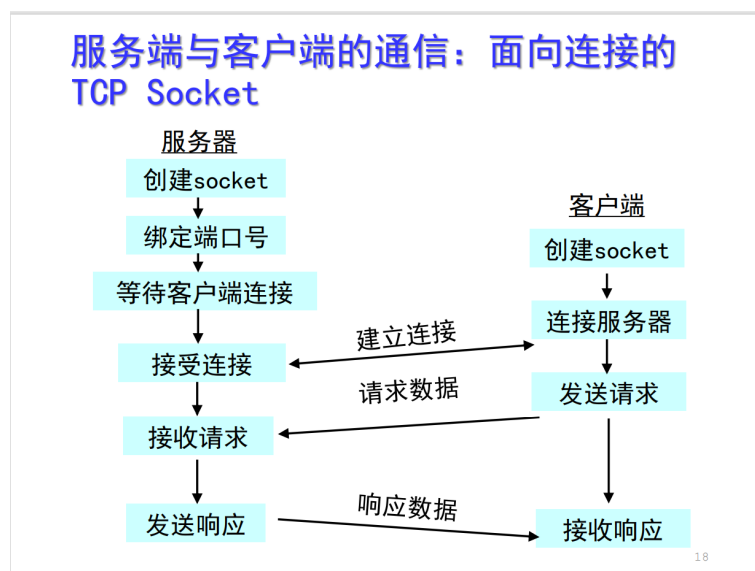


Figure 2.1: 面向连接的 TCP Socket

#### 2.1.1 创建套接字、绑定端口

server 需要固定的、可查询的地址，所以使用 `bind`(捆绑) 将当前套接字绑定到 (ip,port)。而本次实现的客户端只要知道服务器地址就可连接，其本身的端口自动分配。

```
1 class Server():
2 def __init__(self):
3     # TODO: 初始化服务端socket
4     self.server=socket.socket()
5     #-----
6     self.ip = "183.172.237.239" # 服务器IP为local host, 即本机
7     self.port = self.set_port() # 通过命令行标准输入, 设置服务器端口
8     #-----
9     # TODO: 为服务socket绑定IP与端口
10    # self.server.XXXXXXXX(params)
11    self.server.bind((self.ip,self.port))
```

### 2.1.2 client 请求连接、server 接受连接

```

1  def start_connection(self):
2      # TODO: 通过socket连接至对应IP与端口
3      # self.client.XXXXXXX
4      self.client.connect((self.ip, self.port))
5      print("与"+self.ip+"连接建立成功, 可以开始聊天了! (输入q断开连接)")
6      # 为接受消息和发送消息分别开启两个线程, 实现双工聊天
7      Thread(target=self.send_msg).start()
8      Thread(target=self.recv_msg).start()

```

连接由 client 在有通信需求时发起, TCP 为面向连接的协议, 所以 client 发起连接请求, 使用 socket.connect(). 而 server 使用 socket.accept() 被动接受连接。

```

1  cli,add=self.server.accept()

```

accept 有 2 个返回值, 其一为一个新的套接字, 用于和建立连接的 client 通信, 另一个为 client 的地址。如果 server 和 N 个 client 建立了连接, 则 server 的身上插满了 N+1 个套接字, 其中 1 个用于不断接受 client 的连接请求, 剩下的用于和已建立连接的 client 收发消息。

### 2.1.3 消息的收发、响应

使用 send、recv 函数实现消息收发。发前需 encode, 收后需 decode, 进行字符串和字节流之间的转换。

```

1  def recv_msg(self):
2      while(True):
3          try:
4              msg = self.client.recv(BUFFER_SIZE)
5              msg=msg.decode('utf-8')
6              print(msg)
7          except:
8              break

```

client 输入 'q' 时, 套接字的主动关闭, 这一逻辑在 send 线程进行。此时 recv 线程理应也断开, 所以可考虑异常处理 try...except... 方法: 当 recv 出错 (发现自己的 client 关了), 则跳出循环, client 的 recv 线程结束, 该 client.py 运行结束。

### 2.1.4 getsockname、getpeername

在编写代码过程中, 发现 cli 和 address 其实可以不同时保存下来。调用 getsockname 可得到 socket 本机的地址, getpeername 可得到与该 socket 连接的 socket 的地址。最初我认为 accept 返回的是 client 端的套接字, 但通过调用以上两个函数的测试, 最终发现该 socket 是 server 端的, 专门用于通信, 和用于等待连接的 socket 不是一个。

## 2.2 hub 聊天室

该功能需要 server 端维护一个字典, 套接字为 key, 地址为 value。

### 2.2.1 等待连接

```

1  def start_hub_listen(self):
2      while(True):
3          cli,add=self.server.accept()
4          self.ip_client[cli]=add

```

```

5         print(str(addr)+"已成功连接")
6         Thread(target=self.hub_msg_process, args=(cli, addr, self.ip_client
)).start()

```

### 一些问题与解释

- 需要循环的原因：这是 hub 不是 p2p，要连接的 client 很多，所以需要一直被动等待请求，打开多个线程。
- 为什么 hub 模式和每个 client 只开一个线程而不是两个：此模式 server 接收某用户消息后立即广播，而不主动发送消息。所以可以认为 send、recv 成为一体，无须多线程。

### 2.2.2 信息的收发

```

1  def hub_msg_process(self, current_client, current_address, ip_client):
2      while(current_client in ip_client):
3          msg=current_client.recv(BUFFER_SIZE)
4          msg=msg.decode('utf-8')
5          if(msg=='q'):
6              msg=str(current_address)+"已退出"
7              self.hub_close_client(current_client,current_address)
8          else:
9              msg=str(current_address)+': '+msg
10             for key, value in ip_client.items():
11                 if(key!=current_client):
12                     key.send(msg.encode('utf-8'))

```

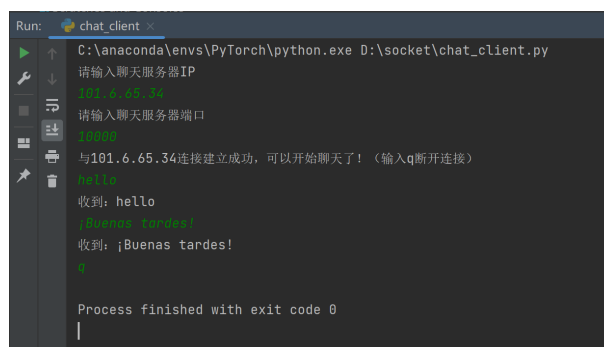
### 一些问题与解答：

- 线程循环运行的条件：current\_client 还在 key 里。  
字典用于维护 server 和各 client 的连接情况，如果该 client 说'q'要退出，则 server 端关闭与其通信的 socket，并从 key 中将其删除。
- 发送消息的 for 循环也可只遍历 ip\_client.keys()

## Chapter 3

# 实验结果

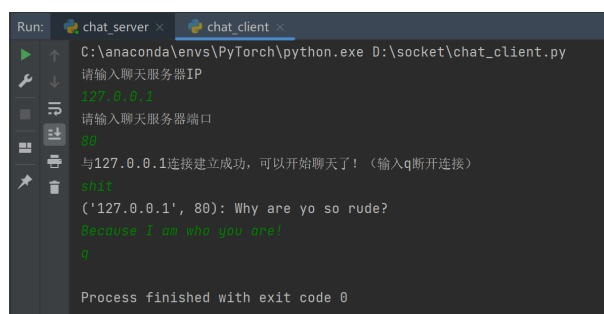
### 3.1 实现客户端聊天程序



```
Run: chat_client x
C:\anaconda\envs\PyTorch\python.exe D:\socket\chat_client.py
请输入聊天服务器IP
101.6.65.34
请输入聊天服务器端口
10000
与101.6.65.34连接建立成功，可以开始聊天了！（输入q断开连接）
hello
收到: hello
¡Buenas tardes!
收到: ¡Buenas tardes!
q
Process finished with exit code 0
```

Figure 3.1: echo from server constructed by the tutor

### 3.2 实现一对一服务端聊天程序

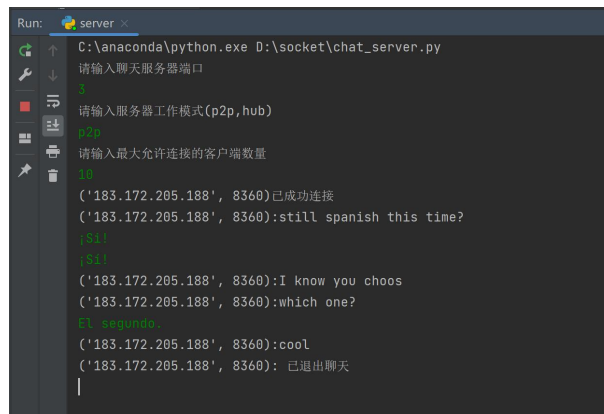


```
Run: chat_server x chat_client x
C:\anaconda\envs\PyTorch\python.exe D:\socket\chat_client.py
请输入聊天服务器IP
127.0.0.1
请输入聊天服务器端口
80
与127.0.0.1连接建立成功，可以开始聊天了！（输入q断开连接）
Why?
('127.0.0.1', 80): Why are yo so rude?
Because I am who you are!
q
Process finished with exit code 0
```

Figure 3.2: communicate with native host

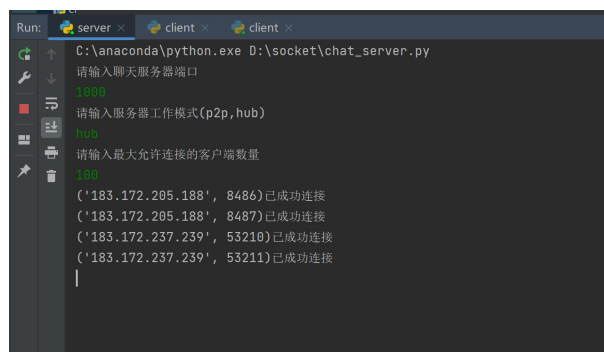
### 3.3 实现聊天室服务端





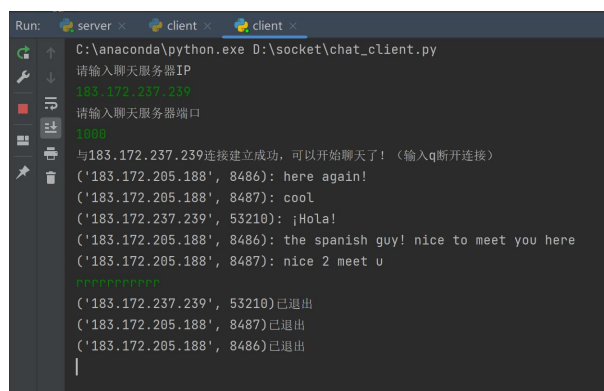
```
Run: server
C:\anaconda\python.exe D:\socket\chat_server.py
请输入聊天服务器端口
3
请输入服务器工作模式(p2p,hub)
p2p
请输入最大允许连接的客户端数量
10
('183.172.205.188', 8360)已成功连接
('183.172.205.188', 8360):still spanish this time?
{83}
{83}
('183.172.205.188', 8360):I know you choos
('183.172.205.188', 8360):which one?
El segundo.
('183.172.205.188', 8360):cool
('183.172.205.188', 8360): 已退出聊天
|
```

Figure 3.3: communicate with my schoolmate as server



```
Run: server
C:\anaconda\python.exe D:\socket\chat_server.py
请输入聊天服务器端口
1000
请输入服务器工作模式(p2p,hub)
hub
请输入最大允许连接的客户端数量
100
('183.172.205.188', 8486)已成功连接
('183.172.205.188', 8487)已成功连接
('183.172.237.239', 53210)已成功连接
('183.172.237.239', 53211)已成功连接
|
```

Figure 3.4: hub server



```
Run: server
C:\anaconda\python.exe D:\socket\chat_client.py
请输入聊天服务器IP
183.172.237.239
请输入聊天服务器端口
1000
与183.172.237.239连接建立成功, 可以开始聊天了! (输入q断开连接)
('183.172.205.188', 8486): here again!
('183.172.205.188', 8487): cool
('183.172.237.239', 53210): ¡Hola!
('183.172.205.188', 8486): the spanish guy! nice to meet you here
('183.172.205.188', 8487): nice 2 meet u
qqqqqqqq
('183.172.237.239', 53210)已退出
('183.172.205.188', 8487)已退出
('183.172.205.188', 8486)已退出
|
```

Figure 3.5: hub client

## Chapter 4

# 思考题

### 4.1

本实验中提供的代码框架使用多线程分别处理消息接收与消息发送，若取消代码中的多线程部分，会出现什么现象？请分析现象原因。

答：无法同时收发。多线程能使 send、recv 两部分同时运行。若取消多线程，如下代码，则只能按照顺序运行一行行代码。

```
1         self.send_msg()
2         self.recv_msg()
```

### 4.2

除多线程外，有无其他方式实现 Socket 双工通信？

答：若不使用多线程，则需将 send、recv 串行处理。可采用计时方法，在同一函数中，给 send、recv 不同的运行时间，交替循环进行。

### 4.3

若使用基于 UDP 的 socket，聊天软件是否能正常工作？二者在使用上有何不同？

答：可正常工作。

不同点

- UDP 不建立连接，不使用 connect、accept 等函数，只需初始化 socket。
- UDP 消息收发，每次都需要规定目标 address。在 python 中体现在 recvfrom、sendto 的 address 参数。