计算机网络LAB2

姓名：姚丁钰

班级：智能2103班

学号：202107030125

实验目的

通过本实验，学习采用Socket（套接字）设计简单的网络数据收发程序，理解应用数据包是如何通过传输层进行传送的。

实验内容

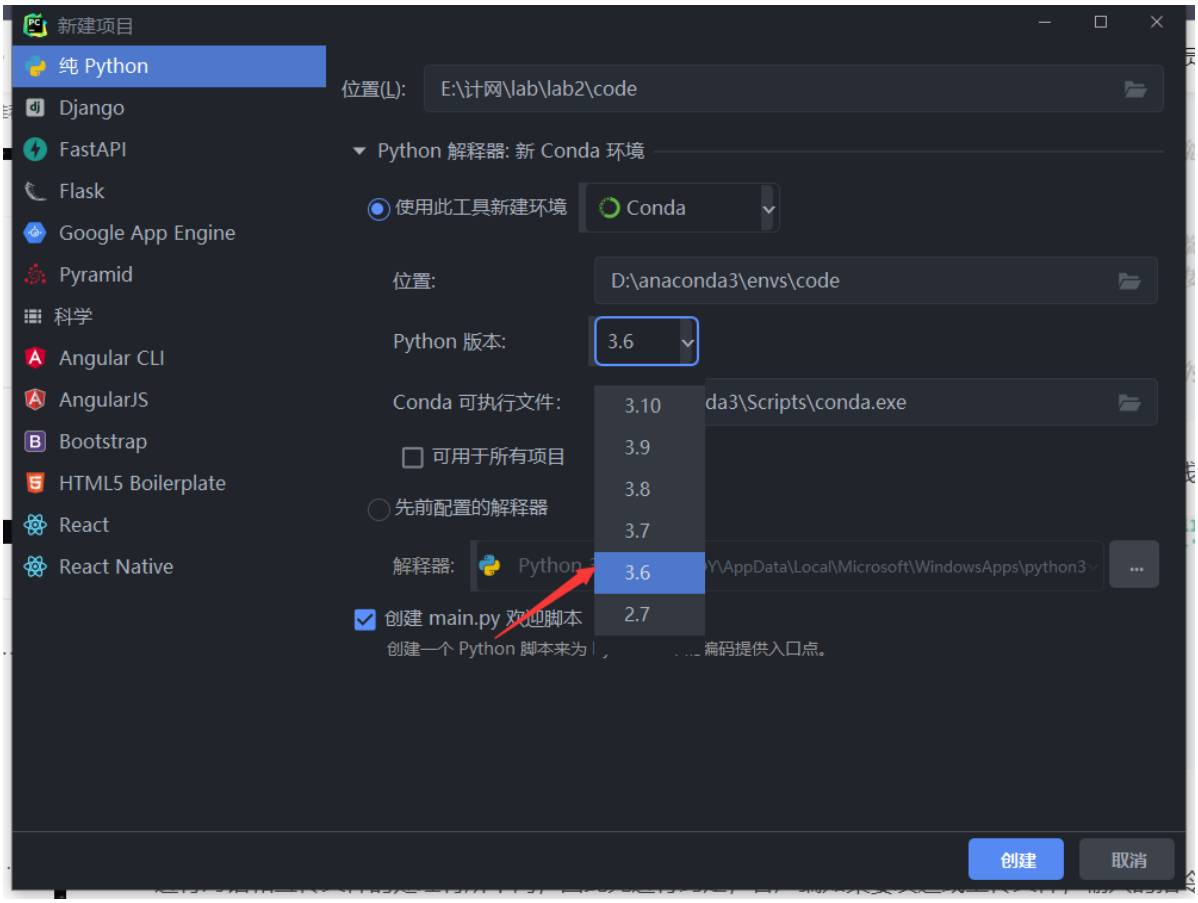
Socket（套接字）是一种抽象层，应用程序通过它来发送和接收数据，就像应用程序打开一个文件句柄，将数据读写到稳定的存储器上一样。一个socket允许应用程序添加到网络中，并与处于同一个网络中的其他应用程序进行通信。一台计算机上的应用程序向socket写入的信息能够被另一台计算机上的另一个应用程序读取，反之亦然。

不同类型的socket与不同类型的底层协议族以及同一协议族中的不同协议栈相关联。现在TCP/IP协议族中的主要socket类型为流套接字（sockets sockets）和数据报套接字（datagram sockets）。流套接字将TCP作为其端对端协议（底层使用IP协议），提供了一个可信赖的字节流服务。一个TCP/IP流套接字代表了TCP连接的一端。数据报套接字使用UDP协议（底层同样使用IP协议），提供了一个"尽力而为"（best-effort）的数据报服务，应用程序可以通过它发送最长65500字节的个人信息。一个TCP/IP套接字由一个互联网地址，一个端对端协议（TCP或UDP协议）以及一个端口号唯一确定。

实验过程

实验环境

- python 3.6



采用TCP进行数据发送的简单程序

> TCP协议，传输控制协议（英语：Transmission Control Protocol，缩写为TCP）是一种面向连接的、可靠的、基于字节流的传输层通信协议，由IETF的RFC 793定义。
>
> TCP通信需要经过创建连接、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。

建设服务端的步骤

1. **导入必要的模块**：导入Python的socket模块，它提供了创建套接字和进行网络通信的功能。

```
import socket
```

2. **配置服务器的IP地址和端口**：确定服务器应该监听的IP地址和端口号。IP地址可以是主机的本地IP（例如，'127.0.0.1'）或公共IP，端口号应该是一个未被占用的整数。

```
server_ip = '127.0.0.1'
server_port = 12345
```

3. **创建服务器套接字**：使用`socket.socket()`函数创建一个TCP socket对象。这将是服务器监听客户端连接的套接字。

```
server_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
```

- `AF_INET` 表示IPv4地址家族。
- `SOCK_STREAM` 表示使用TCP协议。

4. **绑定套接字到IP和端口**：使用`bind()`方法将服务器套接字绑定到指定的IP地址和端口。

```
server_socket.bind((server_ip, server_port))
```

5. **监听客户端连接**：使用`listen()`方法开始监听来自客户端的连接请求。可以指定一个最大连接队列的长度（通常设置为1）。

```
server_socket.listen(1)
```

6. **与客户端通信**：在`client_socket`上与客户端进行通信。使用`recv()`方法接收来自客户端的数据，使用`send()`方法向客户端发送响应数据。
7. **关闭连接**：当通信结束后，确保关闭与客户端的连接以及服务器套接字。

```
client_socket.close()
server_socket.close()
```

建设客户端的步骤

1. **导入必要的模块**：导入Python的socket模块，它提供了创建套接字和进行网络通信的功能。

```
import socket
```

2. **配置服务器的IP地址和端口**：确定要连接的服务器的IP地址和端口号。这些信息应该与服务器端的设置一致。

```
server_ip = '127.0.0.1'  # 服务器IP地址
server_port = 12345      # 服务器端口
```

3. **创建客户端套接字**：使用`socket.socket()`函数创建一个TCP socket对象。这将是客户端用来与服务器通信的套接字。

```
client_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
```

- `AF_INET` 表示IPv4地址家族。
- `SOCK_STREAM` 表示使用TCP协议。

4. **连接到服务器**：使用`connect()`方法连接到服务器的IP地址和端口。

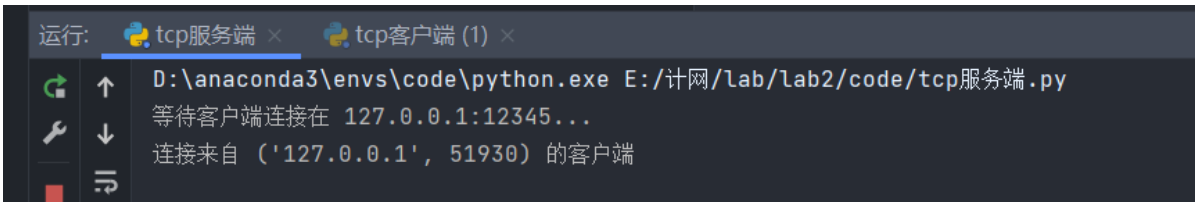
```
client_socket.connect((server_ip, server_port))
```

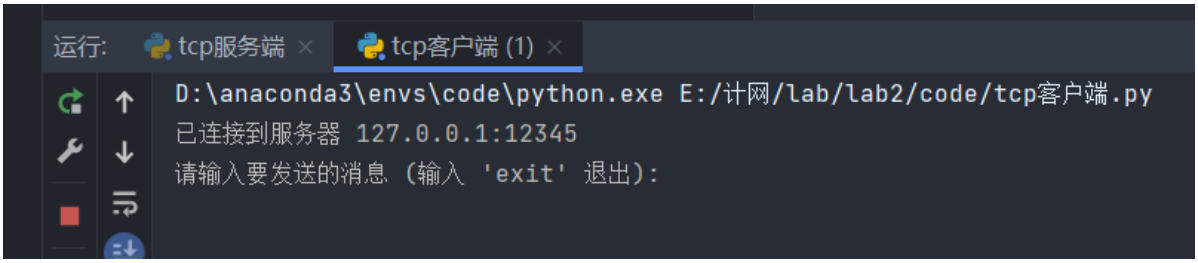
5. **与服务器通信**：在 `client_socket` 上与服务器进行通信。使用 `send()` 方法向服务器发送数据，使用 `recv()` 方法接收服务器的响应数据。
6. **关闭连接**：当通信结束后，确保关闭与服务器的连接。

```
client_socket.close()
```

实验结果

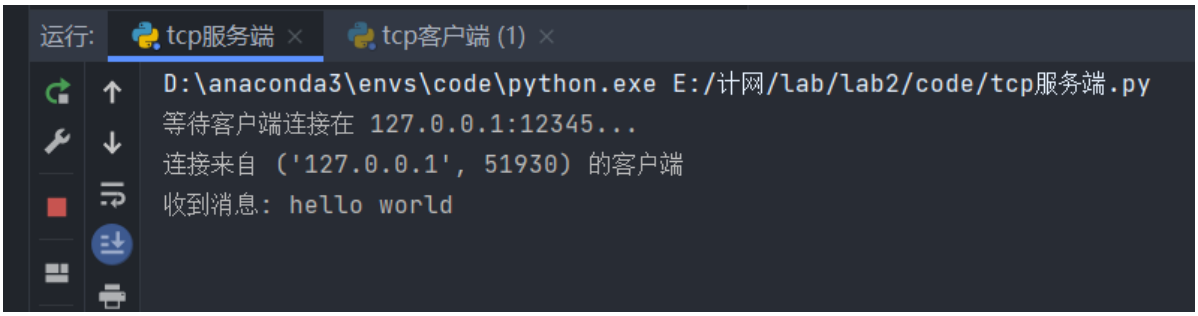
这样客户端和服务端就可以实现简单的通信了，先运行服务端再运行客户端





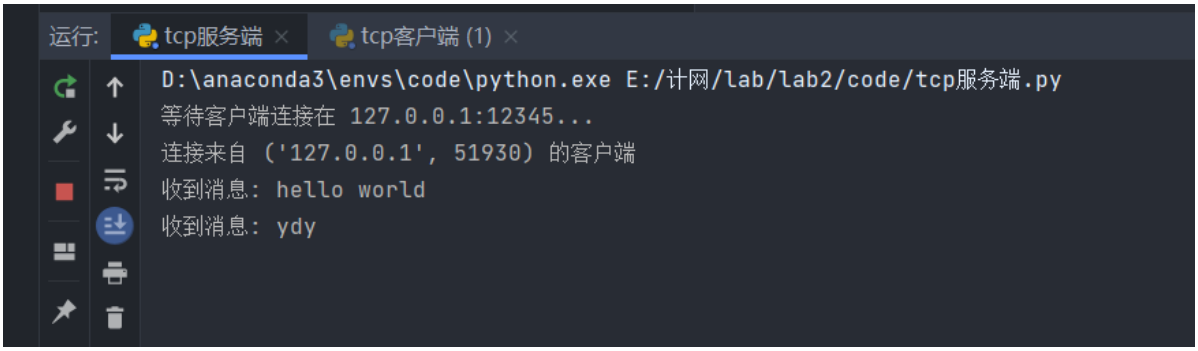
客户端发送hello world，可以在服务端收到“hello world”字符串


运行:
tcp服务端
tcp客户端 (1)
D:\anaconda3\envs\code\python.exe E:/计网/lab/lab2/code/tcp客户端.py
已连接到服务器 127.0.0.1:12345
请输入要发送的消息 (输入 'exit' 退出): hello world
服务器响应: 服务器已收到消息
请输入要发送的消息 (输入 'exit' 退出):

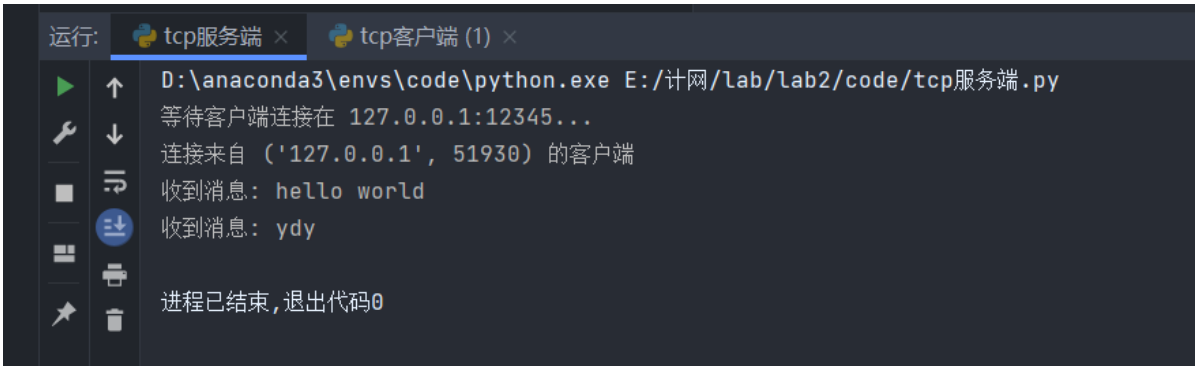

运行:
tcp服务端
tcp客户端 (1)
D:\anaconda3\envs\code\python.exe E:/计网/lab/lab2/code/tcp服务端.py
等待客户端连接在 127.0.0.1:12345...
连接来自 ('127.0.0.1', 51930) 的客户端
收到消息: hello world

继续发送


运行:
tcp服务端
tcp客户端 (1)
D:\anaconda3\envs\code\python.exe E:/计网/lab/lab2/code/tcp客户端.py
已连接到服务器 127.0.0.1:12345
请输入要发送的消息 (输入 'exit' 退出): hello world
服务器响应: 服务器已收到消息
请输入要发送的消息 (输入 'exit' 退出): ydy
服务器响应: 服务器已收到消息
请输入要发送的消息 (输入 'exit' 退出):


运行:
tcp服务端
tcp客户端 (1)
D:\anaconda3\envs\code\python.exe E:/计网/lab/lab2/code/tcp服务端.py
等待客户端连接在 127.0.0.1:12345...
连接来自 ('127.0.0.1', 51930) 的客户端
收到消息: hello world
收到消息: ydy

退出



完整代码

- 服务端

```python
import socket

# 服务器的IP地址和端口
server_ip = '127.0.0.1'  # 服务器IP地址
server_port = 12345      # 服务器端口

# 创建一个TCP socket
server_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)

# 将服务器socket绑定到IP地址和端口
server_socket.bind((server_ip, server_port))

# 监听来自客户端的连接
server_socket.listen(1)
print(f"等待客户端连接在 {server_ip}:{server_port}...")

# 等待客户端连接
client_socket, client_address = server_socket.accept()
print(f"连接来自 {client_address} 的客户端")
```

```
20
21 while True:
22     # 接收客户端发送的数据
23     data = client_socket.recv(1024)
24     if not data:
25         break
26
27     # 打印接收到的数据
28     print(f"收到消息: {data.decode('utf-8')}")
29
30     # 向客户端发送响应
31     response = "服务器已收到消息"
32     client_socket.send(response.encode('utf-8'))
33
34 # 关闭连接
35 client_socket.close()
36 server_socket.close()
```

- 客户端

```
1  import socket
2
3  # 服务器的IP地址和端口
4  server_ip = '127.0.0.1'  # 服务器IP地址
5  server_port = 12345      # 服务器端口
6
7  # 创建一个TCP socket
8  client_socket = socket.socket(socket.AF_INET,
   socket.SOCK_STREAM)
9
10 # 连接到服务器
11 client_socket.connect((server_ip, server_port))
12 print(f"已连接到服务器 {server_ip}:{server_port}")
13
14 while True:
15     # 从用户输入获取消息
16     message = input("请输入要发送的消息 (输入 'exit' 退出):
   ")
17
18     if message == 'exit':
19         break
20
```

```
    # 发送消息到服务器
    client_socket.send(message.encode('utf-8'))

    # 接收服务器的响应
    response = client_socket.recv(1024)
    print(f"服务器响应: {response.decode('utf-8')}")

# 关闭连接
client_socket.close()
```

采用UDP进行数据发送的简单程序

> UDP——用户数据报协议（User Datagram Protocol），是一个无连接的简单的面向数据报的运输层协议。UDP不提供可靠性，它只是把应用程序传给IP层的数据报发送出去，但是并不能保证它们能到达目的地。由于UDP在传输数据报前不用在客户和服务器之间建立一个连接，且没有超时重发等机制，故而传输速度很快。它不属于连接型协议，因而具有资源消耗小，处理速度快的优点，所以通常音频、视频和普通数据在传送时使用UDP较多，因为它们即使偶尔丢失一两个数据包，也不会对接收结果产生太大影响。

UDP&TCP

使用UDP进行数据的发送和使用TCP进行数据的发送很多地方是相同的，这里列举一些不同的地方:

1. 创建套接字

套接字创建的格式为:

```
udp_socket = socket.socket(参数1, 参数2)
```

- 参数1：family（给定的套接族）一般有两种重要参数
  - `socket.AF_INET`（用于服务器与服务器之间的网络通信）
  - `socket.AF_INET6`（基于IPV6方式的服务器与服务器之间的网络通信）
- 参数2：type（套接字类型），也是一般两个类型
  - `socket.SOCK_STREAM`（基于TCP的流式socket通信）
  - `socket.SOCK_DGRAM`（基于UDP的数据报式socket通信）

由于创建的是UDP套接字，所以选择的参数为:

```
udp_socket =
socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
```

2. 发送数据

UDP发送数据使用的函数为sendto，其格式为：

```
udp_socket.sendto(参数1).encode(参数2,('参数3',参数4))
```

- 参数1：表示发送数据的内容
- 参数2：表示编码格式
- 参数3：表示目的地ip
- 参数4：表示目的地端口

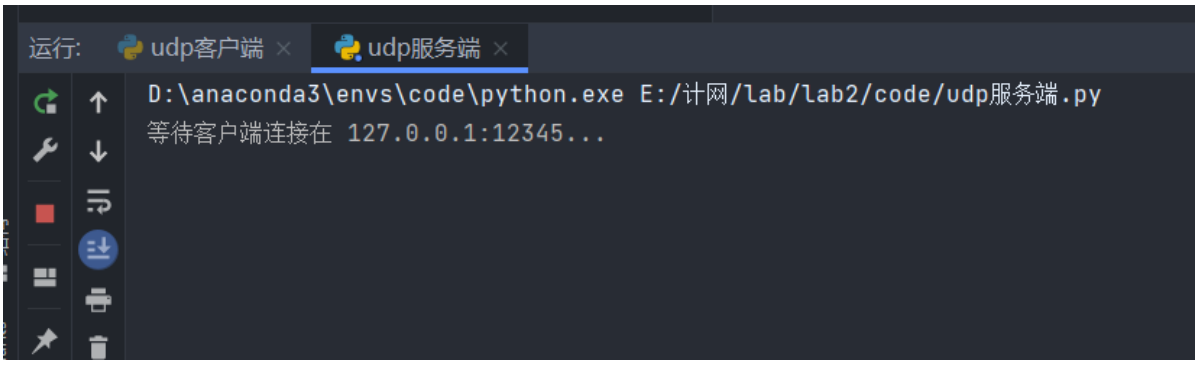
3. 接受数据

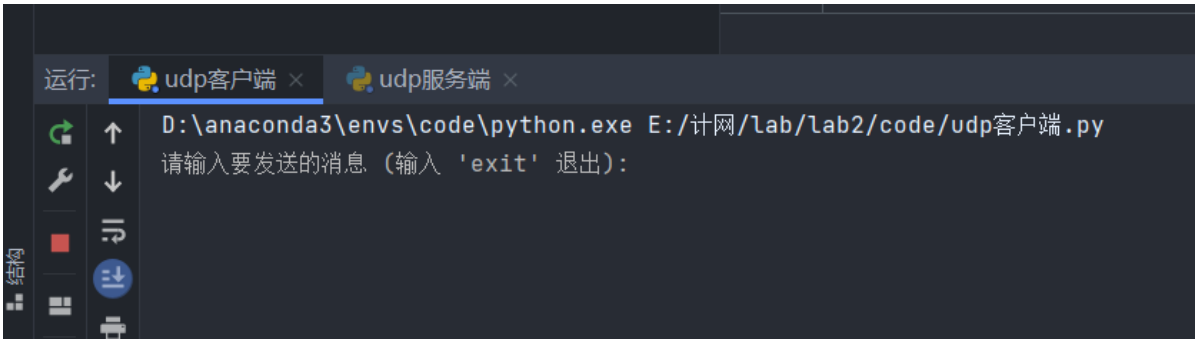
UDP接收数据采用的函数是recvfrom，而TCP采用的是recv，不过函数里面的参数都是要接受的字符的字节长度。recvfrom接收的不仅是数据，也包括发送方的ip地址和端口号信息，可以用于回复信息。

4. UDP的服务端中不需要设置监听模式，也不需要建立连接

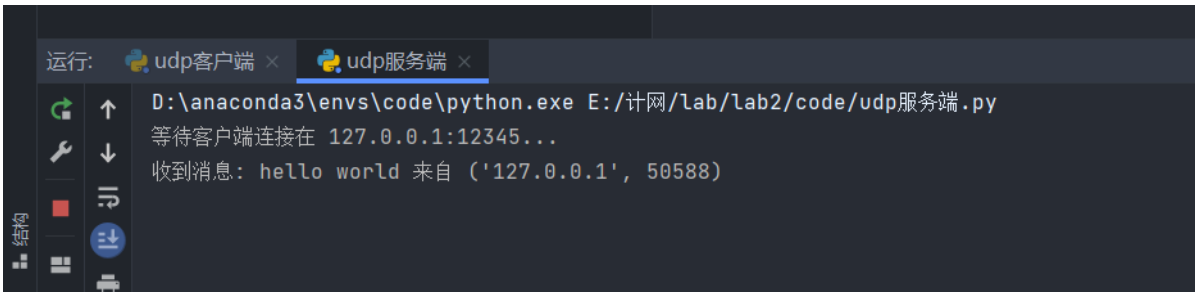
实验结果

这样客户端和服务端就可以实现简单的通信了，先运行服务端再运行客户端

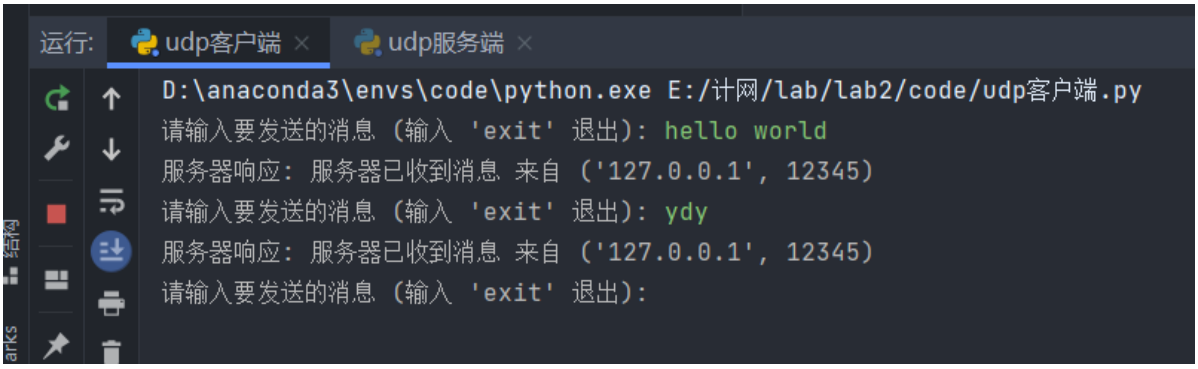


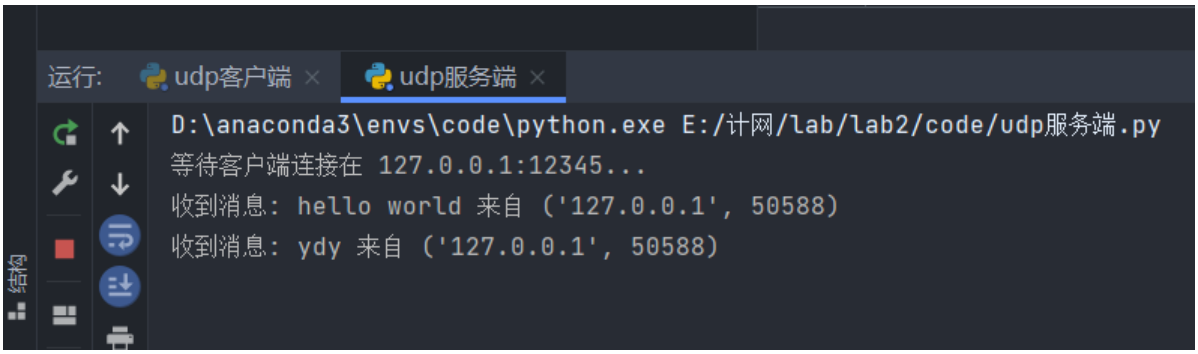


客户端发送hello world，可以在服务端收到“hello world”字符串

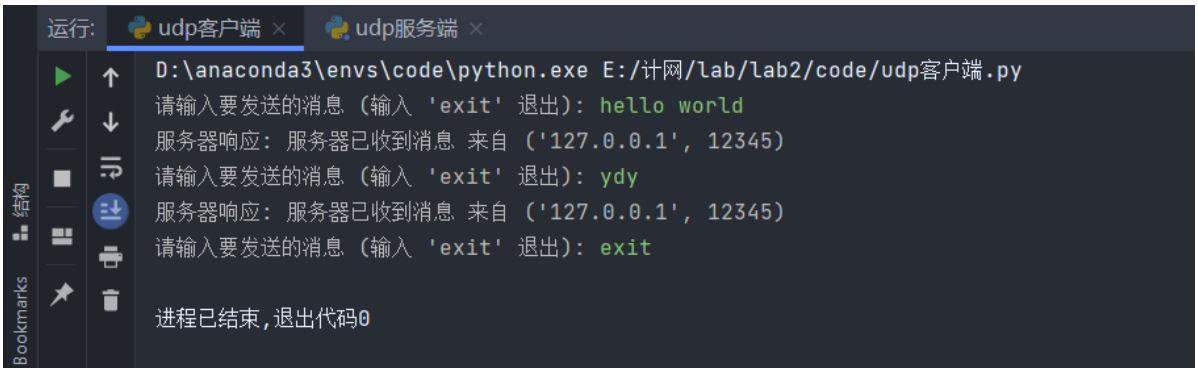


继续发送





退出



完整代码

- 服务端

```python
import socket

# 服务器的IP地址和端口
server_ip = '127.0.0.1'  # 服务器IP地址
server_port = 12345      # 服务器端口

# 创建一个UDP socket
server_socket = socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)

# 将服务器socket绑定到IP地址和端口
server_socket.bind((server_ip, server_port))
print(f"等待客户端连接在 {server_ip}:{server_port}...")

while True:
    # 接收客户端发送的数据
    data, client_address = server_socket.recvfrom(1024)

    # 打印接收到的数据和客户端地址
    print(f"收到消息: {data.decode('utf-8')} 来自
{client_address}")

    # 向客户端发送响应
    response = "服务器已收到消息"
    server_socket.sendto(response.encode('utf-8'),
client_address)
```

- 客户端

```python
import socket

# 服务器的IP地址和端口
server_ip = '127.0.0.1'  # 服务器IP地址
server_port = 12345      # 服务器端口

# 创建一个UDP socket
client_socket = socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)

while True:
    # 从用户输入获取消息
```

```
    message = input("请输入要发送的消息 (输入 'exit' 退出): ")

    if message == 'exit':
        break

    # 发送消息到服务器
    client_socket.sendto(message.encode('utf-8'), (server_ip, server_port))

    # 接收服务器的响应
    response, server_address = client_socket.recvfrom(1024)
    print(f"服务器响应: {response.decode('utf-8')} 来自 {server_address}")

# 关闭连接
client_socket.close()
```

多线程\线程池对比

当一个客户端向一个已经被其他客户端占用的服务器发送连接请求时，虽然其在连接建立后即可向服务器端发送数据，服务器端在处理完已有客户端的请求前，却不会对新的客户端作出响应。

并行服务器：可以单独处理每一个连接，且不会产生干扰。并行服务器分为两种：**一客户一线程和线程池。**

每个新线程都会消耗系统资源：创建一个线程将占用CPU周期，而且每个线程都自己的数据结构（如，栈）也要消耗系统内存。另外，当一个线程阻塞（block）时，JVM将保存其状态，选择另外一个线程运行，并在上下文转换（context switch）时恢复阻塞线程的状态。随着线程数的增加，线程将消耗越来越多的系统资源。这将最终导致系统花费更多的时间来处理上下文转换和线程管理，更少的时间来对连接进行服务。那种情况下，加入一个额外的线程实际上可能增加客户端总服务时间。

我们可以通过限制总线程数并重复使用线程来避免这个问题。与为每个连接创建一个新的线程不同，服务器在启动时创建一个由固定数量线程组成的线程池（thread pool）。当一个新的客户端连接请求传入服务器，它将交给线程池中的一个线程处理。当该线程处理完这个客户端后，又返回线程池，并为下一次请

求处理做好准备。如果连接请求到达服务器时，线程池中的所有线程都已经被占用，它们则在一个队列中等待，直到有空闲的线程可用。

多线程实现TCP客户端/服务器

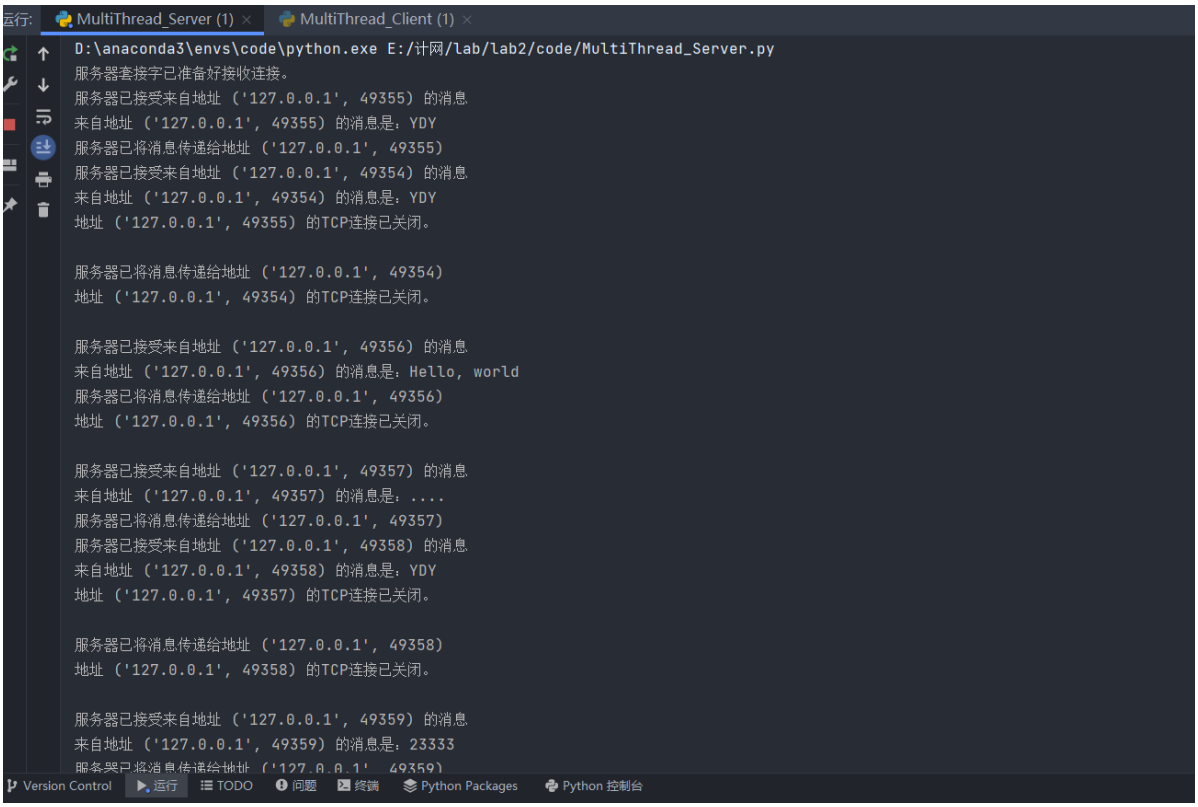
服务器端的实现基于多线程。Worker为线程的工作函数。服务器端的设计思路：

首先，服务器端建立serverSocket套接字，该套接字类似欢迎之门，用于等待和聆听用户的敲门。在while循环中，当serverSocket感知到用户敲门时，将调用accept()方法，创建一个新套接字connectionSocket，由这个特定的客户专用，此时服务器端将创建线程，调用工作函数。工作函数的核心作用为利用该用户专用的connectionSocket套接字完成数据接受、处理、回传操作，同时，由该线程主动关闭该connectionSocket。概括整体流程：**该工作函数的核心作用为维护特定用户的connectionSocket。**由于connectionSocket的管理交由线程负责，因此，主线程(即执行while循环的线程)可以在满足最大接受连接数约束条件的前提下，在主进程占用CPU的过程中，在任意时刻接受来自不同PC主机端的请求连接，从而实现了多线程维护服务器端的正常运作。

客户端的实现基于多线程。Worker为线程的工作函数，**其核心功能为创建clientSocket**，使该clientSocket与服务器建立TCP连接，向服务器的connectionSocket发送数据，接受connectionSocket传输的数据。客户端基于多线程实现数据发送可以有效地模拟真实生产环境下，不同PC主机向服务器发送数据的状态，从而对多线程/线程池的工作效果加以分析。

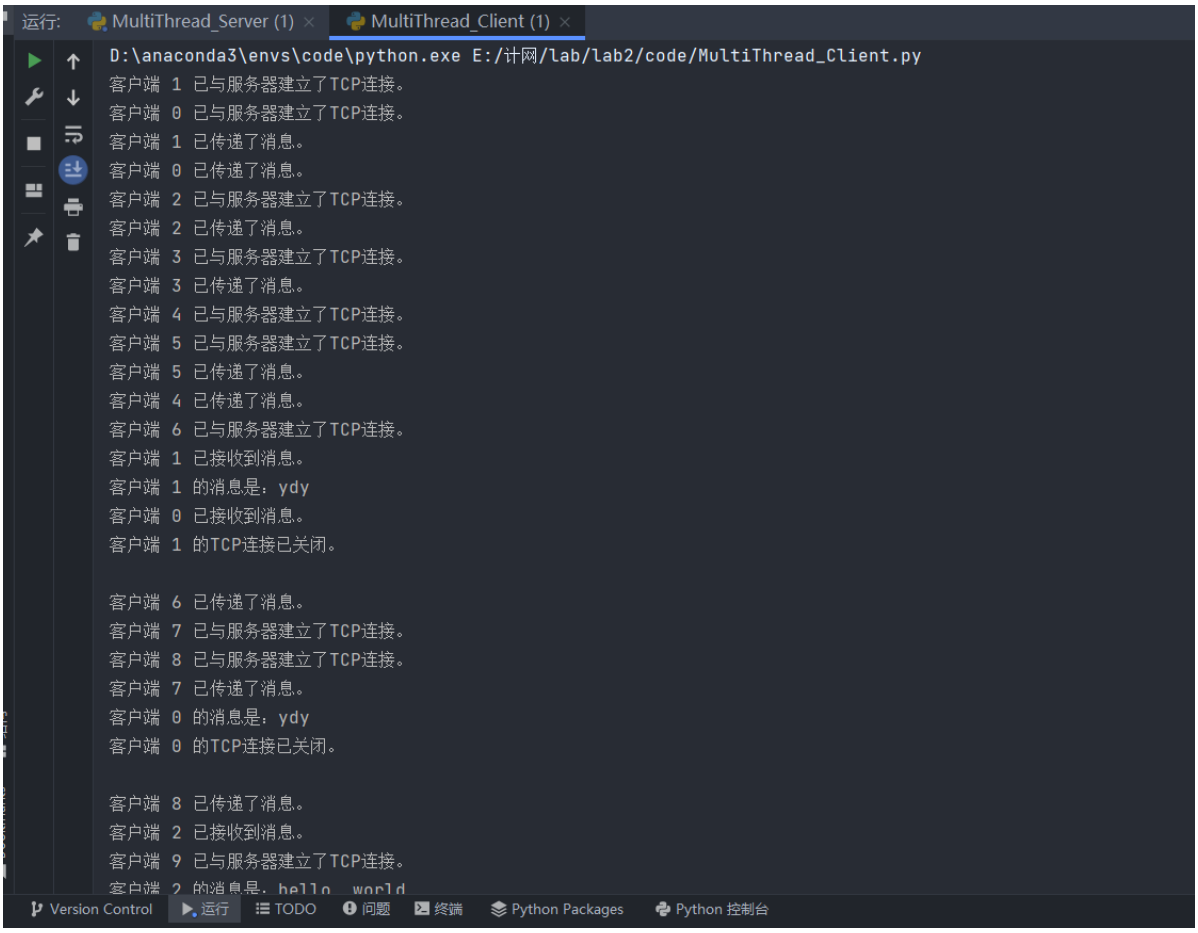
实验结果

- 服务端


运行:
MultiThread_Server (1)
MultiThread_Client (1)
D:\anaconda3\envs\code\python.exe E:/计网/lab/lab2/code/MultiThread_Server.py
服务器套接字已准备好接收连接。
服务器已接受来自地址 ('127.0.0.1', 49355) 的消息
来自地址 ('127.0.0.1', 49355) 的消息是：YDY
服务器已将消息传递给地址 ('127.0.0.1', 49355)
服务器已接受来自地址 ('127.0.0.1', 49354) 的消息
来自地址 ('127.0.0.1', 49354) 的消息是：YDY
地址 ('127.0.0.1', 49355) 的TCP连接已关闭。
服务器已将消息传递给地址 ('127.0.0.1', 49354)
地址 ('127.0.0.1', 49354) 的TCP连接已关闭。
服务器已接受来自地址 ('127.0.0.1', 49356) 的消息
来自地址 ('127.0.0.1', 49356) 的消息是：Hello, world
服务器已将消息传递给地址 ('127.0.0.1', 49356)
地址 ('127.0.0.1', 49356) 的TCP连接已关闭。
服务器已接受来自地址 ('127.0.0.1', 49357) 的消息
来自地址 ('127.0.0.1', 49357) 的消息是：....
服务器已将消息传递给地址 ('127.0.0.1', 49357)
服务器已接受来自地址 ('127.0.0.1', 49358) 的消息
来自地址 ('127.0.0.1', 49358) 的消息是：YDY
地址 ('127.0.0.1', 49357) 的TCP连接已关闭。
服务器已将消息传递给地址 ('127.0.0.1', 49358)
地址 ('127.0.0.1', 49358) 的TCP连接已关闭。
服务器已接受来自地址 ('127.0.0.1', 49359) 的消息
来自地址 ('127.0.0.1', 49359) 的消息是：23333
Version Control
运行
TODO
问题
终端
Python Packages
Python 控制台

!

- 客户端


运行:
MultiThread_Server (1)
MultiThread_Client (1)
D:\anaconda3\envs\code\python.exe E:/计网/lab/lab2/code/MultiThread_Client.py
客户端 1 已与服务器建立了TCP连接。
客户端 0 已与服务器建立了TCP连接。
客户端 1 已传递了消息。
客户端 0 已传递了消息。
客户端 2 已与服务器建立了TCP连接。
客户端 2 已传递了消息。
客户端 3 已与服务器建立了TCP连接。
客户端 3 已传递了消息。
客户端 4 已与服务器建立了TCP连接。
客户端 5 已与服务器建立了TCP连接。
客户端 5 已传递了消息。
客户端 4 已传递了消息。
客户端 6 已与服务器建立了TCP连接。
客户端 1 已接收到消息。
客户端 1 的消息是：ydy
客户端 0 已接收到消息。
客户端 1 的TCP连接已关闭。
客户端 6 已传递了消息。
客户端 7 已与服务器建立了TCP连接。
客户端 8 已与服务器建立了TCP连接。
客户端 7 已传递了消息。
客户端 0 的消息是：ydy
客户端 0 的TCP连接已关闭。
客户端 8 已传递了消息。
客户端 2 已接收到消息。
客户端 9 已与服务器建立了TCP连接。
Version Control
运行
TODO
问题
终端
Python Packages
Python 控制台

完整代码

- 服务端

```python
from socket import *
import threading

serverPort = 12346

def Worker(connSocket, addr):
    sentence = connSocket.recv(1024).decode()
    print("服务器已接受来自地址 " + str(addr) + " 的消息")
    print("来自地址 " + str(addr) + " 的消息是：" + sentence)
    sentence = sentence.lower()
    connSocket.send(sentence.encode())
    print("服务器已将消息传递给地址 " + str(addr))
    connSocket.close()
    print("地址 " + str(addr) + " 的TCP连接已关闭。\n")

serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(('', serverPort))
serverSocket.listen(128)
print("服务器套接字已准备好接收连接。")

while True:
    connectionSocket, addr = serverSocket.accept()
    thread = threading.Thread(target=Worker, args=(connectionSocket, addr))
    thread.start()
```

- 客户端

```python
from socket import *
import threading
import random

serverName = '127.0.0.1'
serverPort = 12346
sentencePool = ["YDY", "Hello, world", "23333", "...."]

def Worker(number):
    clientSocket = socket(AF_INET, SOCK_STREAM)
    clientSocket.connect((serverName, serverPort))
    print("客户端 " + str(number) + " 已与服务器建立了TCP连接。")
```

```
    sentence = sentencePool[random.randint(0, 3)]
    clientSocket.send(sentence.encode())
    print("客户端 " + str(number) + " 已传递了消息。")
    modifiedSentence = clientSocket.recv(1024).decode()
    print("客户端 " + str(number) + " 已接收到消息。")
    print("客户端 " + str(number) + " 的消息是: " +
modifiedSentence)
    clientSocket.close()
    print("客户端 " + str(number) + " 的TCP连接已关闭。\n")

for number in range(10):
    thread = threading.Thread(target=Worker, args=
(number,))
    thread.start()
```

线程池

有关线程池的定义：

线程池是一种多线程处理形式，处理过程中将任务添加到队列，然后在创建线程后自动启动这些任务。线程池线程都是后台线程。每个线程都使用默认的堆栈大小，以默认的优先级运行，并处于多线程单元中。如果某个线程在托管代码中空闲（如正在等待某个事件）,则线程池将插入另一个辅助线程来使所有处理器保持繁忙。如果所有线程池线程都始终保持繁忙，但队列中包含挂起的工作，则线程池将在一段时间后创建另一个辅助线程但线程的数目永远不会超过最大值。超过最大值的线程可以排队，但他们要等到其他线程完成后才启动。

从 Python3.2 开始，标准库为我们提供了 concurrent.futures 模块，它提供了 ThreadPoolExecutor 和 ProcessPoolExecutor两个类，实现了对 threading 和 multiprocessing 的进一步抽象，不仅可以自动调度线程，还可以做到：

1. 主线程可以获取某一个线程（或者任务的）的状态，以及返回值；
2. 当一个线程完成的时候，主线程能够立即知道；
3. 让多线程和多进程的编码接口一致。

我们可以使用 ThreadPoolExecutor 来实例化线程池对象。传入max_workers 参数来设置线程池中最多能同时运行的线程数目。我们可以使用 submit 函数来提交线程需要执行的任务（函数名和参数）到线程池中，并返回该任务的抽象对象，注意 submit() 不是阻塞的，而是立即返回。通过 submit 函数返回的任务抽象对象，能够使用其 done() 方法判断任务是否结束。

在ThreadPool_Server的设计中，我们的核心工作函数Worker()与在MultiThread_Server中的Worker()保持一致。我们通过ThreadPoolExecutor()创建了一个最大工作线程数为10的线程池，并且当serverSocket感知到有新用户敲门时，

若将**服务器端线程池最大工作线程数设置为比客户端请求线程总数小**，由于线程池只会维护最大工作线程数的线程进行工作，因此，当线程池的线程已满时，**后到的任务需要排队等待线程池对其进行工作调度。**

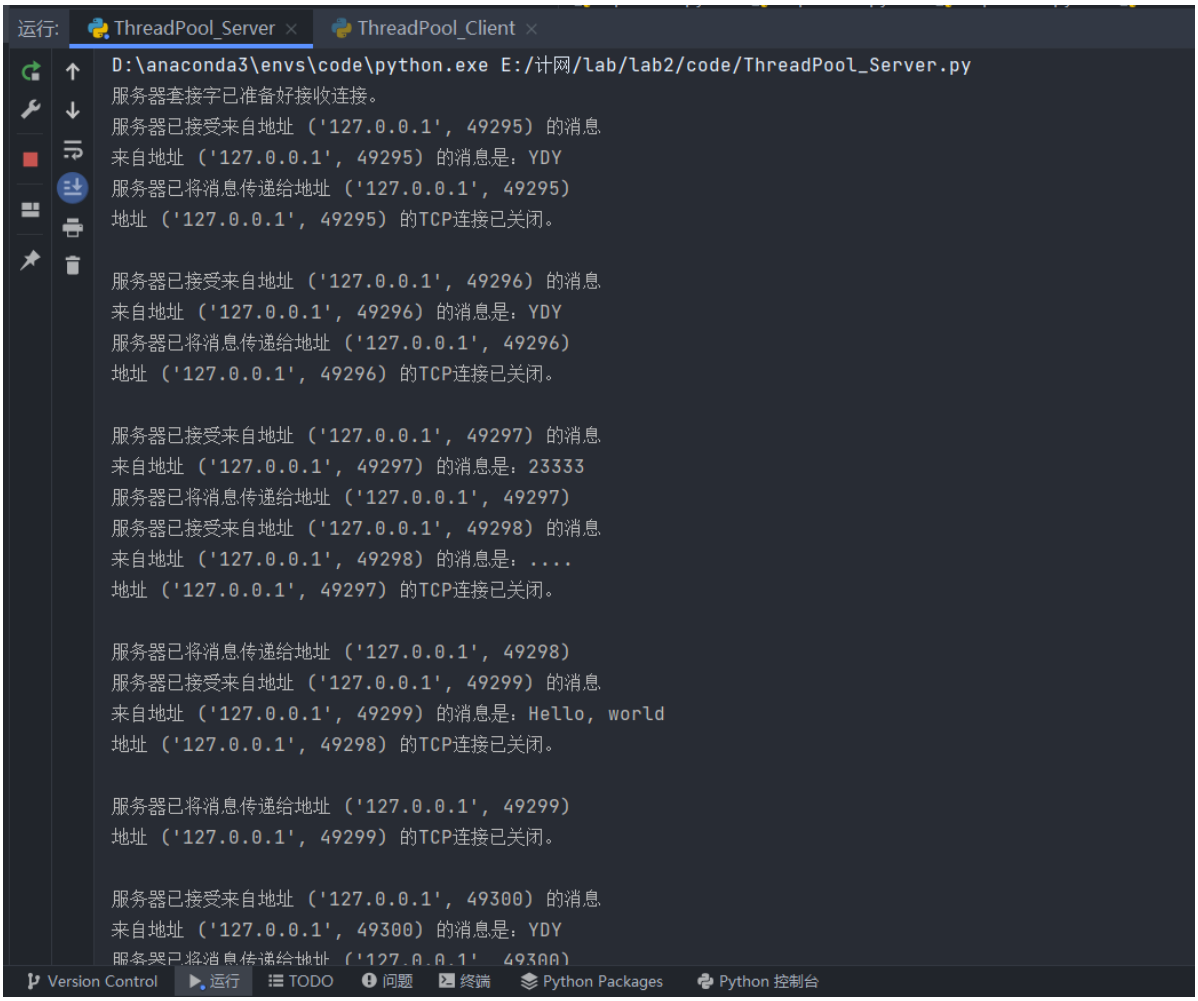
多线程/线程池的比较

多线程并**不会阻塞一个新连接的处理**，对于多线程而言，它会一直消耗资源来进行连接处理，直到计算机资源的耗尽。随着线程数的增多，系统资源将被消耗地更多，这将导致系统花费大量的时间处理上下文切换和线程管理，主线程中与其它客户建立connectionSocket进行服务的时间将会相对应的减少。

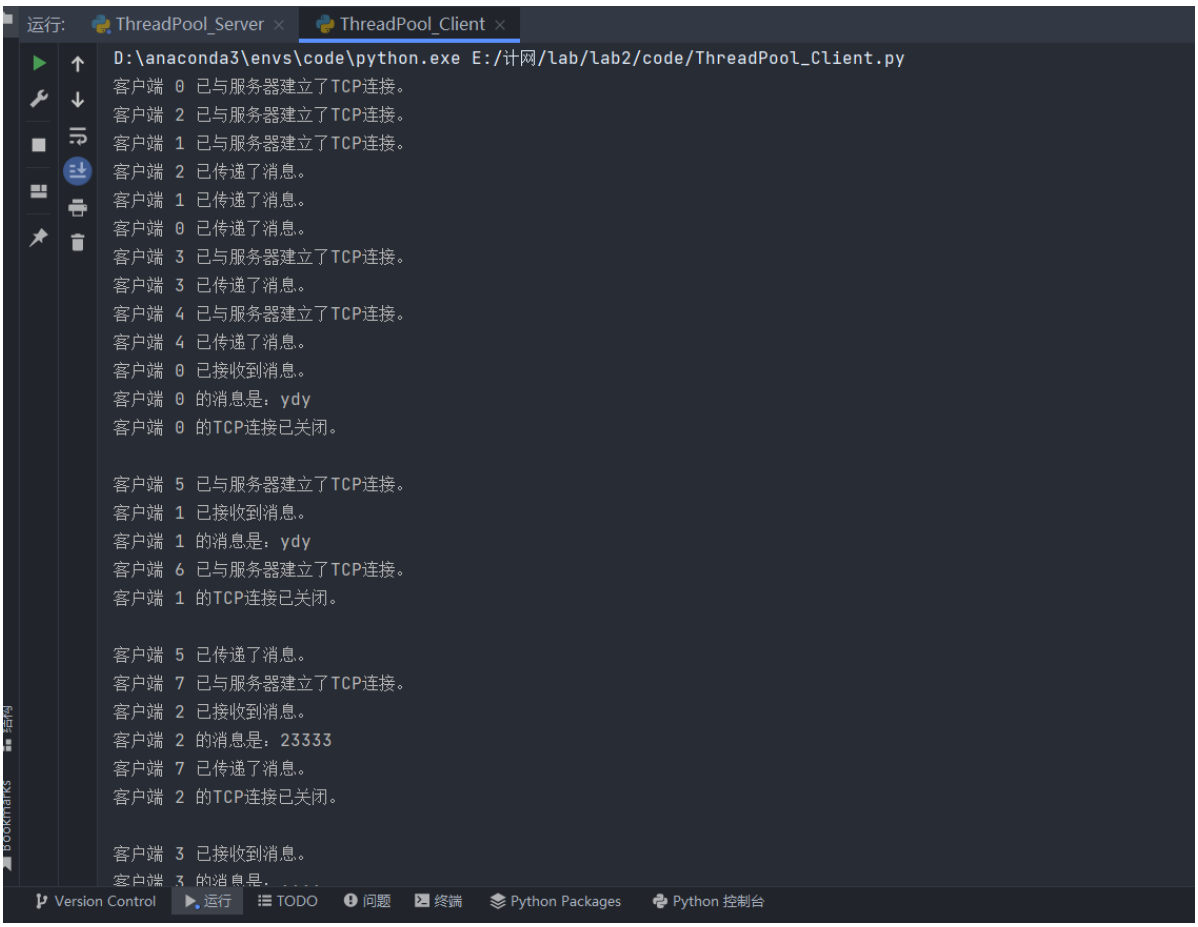
线程池**仅在有空闲线程的时候才会处理新的连接**，否则将进行阻塞。当新的客户端连接请求传入服务器时，它将交给线程池中的一个线程处理。当该线程处理完这个客户端后，线程将返回线程池，为下一次请求处理做好准备。

实验结果

- 服务端


运行:
ThreadPool_Server
ThreadPool_Client
D:\anaconda3\envs\code\python.exe E:/计网/lab/lab2/code/ThreadPool_Server.py
服务器套接字已准备好接收连接。
服务器已接受来自地址 ('127.0.0.1', 49295) 的消息
来自地址 ('127.0.0.1', 49295) 的消息是：YDY
服务器已将消息传递给地址 ('127.0.0.1', 49295)
地址 ('127.0.0.1', 49295) 的TCP连接已关闭。
服务器已接受来自地址 ('127.0.0.1', 49296) 的消息
来自地址 ('127.0.0.1', 49296) 的消息是：YDY
服务器已将消息传递给地址 ('127.0.0.1', 49296)
地址 ('127.0.0.1', 49296) 的TCP连接已关闭。
服务器已接受来自地址 ('127.0.0.1', 49297) 的消息
来自地址 ('127.0.0.1', 49297) 的消息是：23333
服务器已将消息传递给地址 ('127.0.0.1', 49297)
服务器已接受来自地址 ('127.0.0.1', 49298) 的消息
来自地址 ('127.0.0.1', 49298) 的消息是：....
地址 ('127.0.0.1', 49297) 的TCP连接已关闭。
服务器已将消息传递给地址 ('127.0.0.1', 49298)
服务器已接受来自地址 ('127.0.0.1', 49299) 的消息
来自地址 ('127.0.0.1', 49299) 的消息是：Hello, world
地址 ('127.0.0.1', 49298) 的TCP连接已关闭。
服务器已将消息传递给地址 ('127.0.0.1', 49299)
地址 ('127.0.0.1', 49299) 的TCP连接已关闭。
服务器已接受来自地址 ('127.0.0.1', 49300) 的消息
来自地址 ('127.0.0.1', 49300) 的消息是：YDY
Version Control
运行
TODO
问题
终端
Python Packages
Python 控制台

- 客户端


运行:
ThreadPool_Server
ThreadPool_Client
D:\anaconda3\envs\code\python.exe E:/计网/lab/lab2/code/ThreadPool_Client.py
客户端 0 已与服务器建立了TCP连接。
客户端 2 已与服务器建立了TCP连接。
客户端 1 已与服务器建立了TCP连接。
客户端 2 已传递了消息。
客户端 1 已传递了消息。
客户端 0 已传递了消息。
客户端 3 已与服务器建立了TCP连接。
客户端 3 已传递了消息。
客户端 4 已与服务器建立了TCP连接。
客户端 4 已传递了消息。
客户端 0 已接收到消息。
客户端 0 的消息是：ydy
客户端 0 的TCP连接已关闭。
客户端 5 已与服务器建立了TCP连接。
客户端 1 已接收到消息。
客户端 1 的消息是：ydy
客户端 6 已与服务器建立了TCP连接。
客户端 1 的TCP连接已关闭。
客户端 5 已传递了消息。
客户端 7 已与服务器建立了TCP连接。
客户端 2 已接收到消息。
客户端 2 的消息是：23333
客户端 7 已传递了消息。
客户端 2 的TCP连接已关闭。
客户端 3 已接收到消息。
Version Control
运行
TODO
问题
终端
Python Packages
Python 控制台

完整代码

- 服务端

```python
from concurrent.futures import ThreadPoolExecutor
from socket import *

serverPort = 12349

def Worker(connSocket, addr):
    sentence = connSocket.recv(1024).decode()
    print("服务器已接受来自地址 " + str(addr) + " 的消息")
    print("来自地址 " + str(addr) + " 的消息是：" + sentence)
    sentence = sentence.lower()
    connSocket.send(sentence.encode())
    print("服务器已将消息传递给地址 " + str(addr))
    connSocket.close()
    print("地址 " + str(addr) + " 的TCP连接已关闭。\n")

serverSocket = socket(AF_INET, SOCK_STREAM)
serverSocket.bind(('', serverPort))
serverSocket.listen(128)
print("服务器套接字已准备好接收连接。")

pool = ThreadPoolExecutor(max_workers=10)

while True:
    connectionSocket, addr = serverSocket.accept()
    pool.submit(Worker, connectionSocket, addr)
```

- 客户端

```python
from socket import *
import threading
import random

serverName = '127.0.0.1'
serverPort = 12349
sentencePool = ["YDY", "Hello, world", "23333", "...."]

def Worker(number):
    clientSocket = socket(AF_INET, SOCK_STREAM)
```

```
    clientSocket.connect((serverName, serverPort))
    print("客户端 " + str(number) + " 已与服务器建立了TCP连接。")
    sentence = sentencePool[random.randint(0, 3)]
    clientSocket.send(sentence.encode())
    print("客户端 " + str(number) + " 已传递了消息。")
    modifiedSentence = clientSocket.recv(1024).decode()
    print("客户端 " + str(number) + " 已接收到消息。")
    print("客户端 " + str(number) + " 的消息是：" + modifiedSentence)
    clientSocket.close()
    print("客户端 " + str(number) + " 的TCP连接已关闭。\n")

for number in range(10):
    thread = threading.Thread(target=Worker, args=(number,))
    thread.start()

```

写一个简单的chat程序，并能互传文件，编程语言不限

服务端

1. 引入必要的库：
   - `socketserver`：用于创建TCP服务器。
   - `os`：用于文件和目录操作。
   - `json`：用于将目录结构信息转化为JSON格式。
2. 定义服务器的主机名和端口号：
   - `serverName`：服务器主机名，设置为'localhost'，表示本地服务器。
   - `serverPort`：服务器端口号，设置为12101。
3. 创建服务器实例并启动：
   - 创建一个TCP服务器实例，使用`socketserver.ThreadingTCPServer`，并指定监听的主机名和端口号。
   - 通过`instance.serve_forever()`启动服务器，使其一直运行，监听来自客户端的连接请求。
4. 定义自定义服务器处理类`MyServer`，继承自`socketserver.BaseRequestHandler`：

- 在 `MyServer` 类中，定义了一个 `handle` 方法，该方法会在客户端连接到服务器时被调用，处理客户端的请求。

5. 处理客户端请求：
   - 通过 `conn = self.request` 获取与客户端连接的套接字连接对象。
6. 根据客户端发送的指令 (`op`) 执行不同的操作：
   - 如果 `op` 为 "u"，表示客户端请求上传文件，服务器会执行文件上传操作。
   - 如果 `op` 为 "d"，表示客户端请求下载文件，服务器会执行文件下载操作。
   - 如果 `op` 为 "c"，表示客户端发送信息给服务器，服务器会确认接收信息。
   - 如果 `op` 为 "ls"，表示客户端请求获取当前服务器目录的文件和目录结构。
7. 文件上传操作 ("u")：
   - 接收客户端发送的文件名和文件大小信息。
   - 检查服务器上是否已存在同名文件，如果存在，则重命名文件。
   - 告知客户端文件接收状态。
   - 接收文件内容并将其写入服务器上的文件。
   - 向客户端发送上传成功的消息。
8. 文件下载操作 ("d")：
   - 接收客户端请求下载的文件名。
   - 检查服务器上是否存在请求的文件，如果存在，发送 "yes" 给客户端，否则发送 "no"。
   - 发送文件名和文件大小给客户端。
   - 逐步发送文件内容给客户端。
9. 客户端信息确认操作 ("c")：
   - 接收客户端发送的信息并确认收到。
10. 获取目录结构操作 ("ls")：
    - 使用 `os.walk` 遍历服务器指定目录下的文件和子目录信息。
    - 将目录信息序列化为 JSON 格式并发送给客户端。

客户端

1. 引入必要的库：
   - `socket`：用于创建套接字连接，实现客户端与服务器的通信。
   - `os`：用于文件和目录操作。

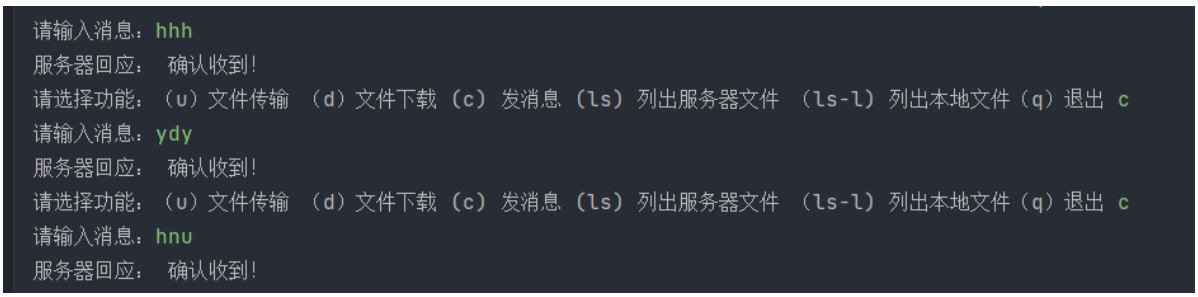
- `json`: 用于将目录结构信息转化为 JSON 格式。

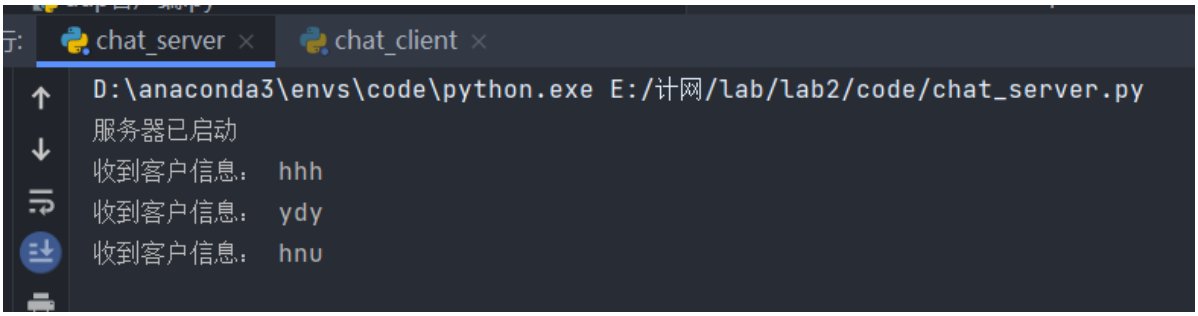
2. 定义服务器的主机名和端口号：
   - `serverName`: 服务器主机名，设置为'localhost'，表示本地服务器。
   - `serverPort`: 服务器端口号，设置为12101。
3. 创建套接字连接：
   - 使用 `socket.socket(socket.AF_INET, socket.SOCK_STREAM)` 创建一个套接字对象，指定使用IPv4和TCP协议。
   - 使用 `sk.connect(ip_port)` 连接到指定的服务器地址和端口。
4. 定义客户端本地文件存储路径 `base_path`。
5. 通过一个无限循环等待用户输入来选择不同的功能：
   - "u": 上传文件到服务器。
   - "d": 从服务器下载文件。
   - "c": 向服务器发送消息。
   - "ls": 列出服务器上的文件和目录结构。
   - "ls-l": 列出本地客户端的文件和目录结构。
   - "q": 退出客户端。
6. 文件上传操作 ("u")：
   - 获取用户输入的文件名和路径。
   - 检查本地文件是否存在，如果不存在则提示用户并返回循环。
   - 发送 "u" 给服务器表示文件上传操作。
   - 获取文件大小并发送文件名和文件大小给服务器。
   - 使用循环逐步读取文件内容并发送给服务器。
   - 接收服务器的上传成功消息并显示。
7. 文件下载操作 ("d")：
   - 发送 "d" 给服务器表示文件下载操作。
   - 获取用户输入的文件名并发送给服务器。
   - 接收服务器的文件存在与否的响应。
   - 如果文件存在，接收文件信息包括文件名和文件大小。
   - 如果本地存在同名文件，重命名文件。
   - 使用循环逐步接收文件内容并写入本地文件。
   - 接收服务器的下载成功消息并显示。
8. 列出服务器文件和目录结构 ("ls")：
   - 发送 "ls" 给服务器。
   - 接收服务器发送的JSON格式的目录结构信息，分为根目录、子目录和文件。
   - 显示目录结构信息。

9. 列出客户端本地文件和目录结构 ("ls-l"):
    - 使用 `os.walk` 遍历本地客户端的文件和子目录信息。
    - 显示本地目录结构。
10. 与服务器通信操作 ("c"):
    - 发送 "c" 给服务器表示发送消息。
    - 获取用户输入的消息并发送给服务器。
    - 接收服务器的回应消息并显示。
11. 退出客户端操作 ("q"):
    - 断开与服务器的连接。
    - 退出循环，结束程序。

实验结果

- chat

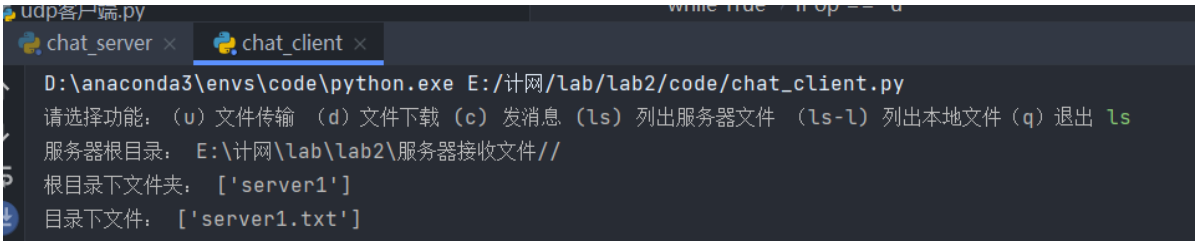




- 列出本地文件

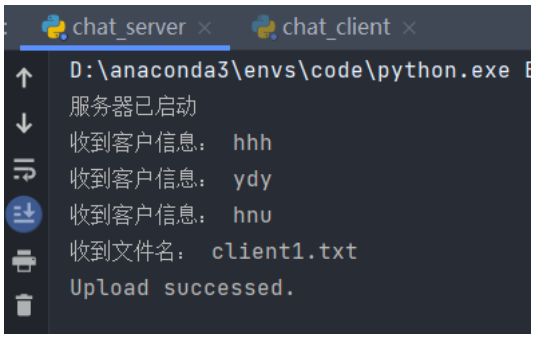
```
请选择功能：（u）文件传输 （d）文件下载 （c） 发消息 （ls） 列出服务器文件 （ls-l） 列出本地文件（q）退出 ls-l
E:\计网\lab\lab2\客户端发送文件/
['client1']
['client1.txt']

E:\计网\lab\lab2\客户端发送文件/client1
[]
['hello.txt', 'ydy.txt']
```

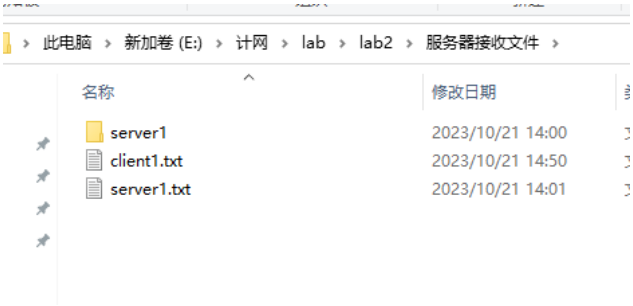
- 列出服务器文件



- 文件传输

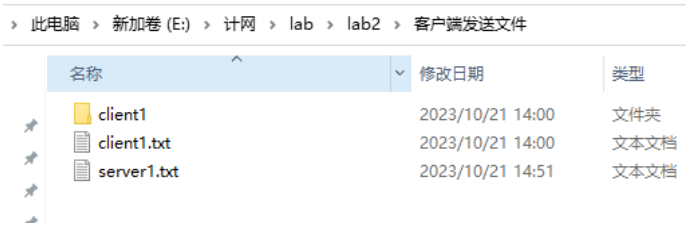
```
请选择功能：（u）文件传输 （d）文件下载 （c） 发消息 （ls） 列出服务器文件 （ls-l） 列出本地文件（q）退出 u
输入文件名(含后缀)：client1.txt
等待文件传输
Upload successed.
```





- 文件下载

```
请选择功能：（u）文件传输 （d）文件下载 （c） 发消息 （ls） 列出服务器文件 （ls-l） 列出本地文件（q）退出 d
输入文件名(含后缀)：server1.txt
Download successed.
```



完整代码

- 服务端

```
import socketserver
import os
import json
serverName = 'localhost'
serverPort = 12101
ip_port = (serverName, serverPort)
print("服务器已启动")
class MyServer(socketserver.BaseRequestHandler):
    def handle(self):
        base_path = 'E:\计网\lab\lab2\服务器接收文件/'
        conn = self.request
        while True:
            op = conn.recv(1024).decode()
            if(op=="u"):
                file_inf = conn.recv(1024).decode()

                # 获取请求方法、文件名、文件大小
                file_name, file_size = file_inf.split('|') #分割 文件名和大小

                id=1
                while (os.path.exists(base_path + '/' + file_name)):
                    file_name='1'+file_name
                    id+=1
                if (id!=1):
                    mess="文件重复，已经重命名为"+file_name
                    print('收到文件名：命名重复，新名称为：', file_name)
                else:
                    mess="等待文件传输"
                    print('收到文件名：', file_name)
```

```
                conn.send(mess.encode())
                # 已经接收文件的大小
                recv_size = 0
                # 上传文件路径拼接
                file_dir = os.path.join(base_path,
file_name)
                f = open(file_dir, 'wb')
                Flag = True
                while Flag:
                    # 未上传完毕,
                    if int(file_size) > recv_size:
                        # 最多接收1024，可能接收的小于1024
                        data = conn.recv(1024)
                        recv_size += len(data)
                        # 写入文件
                        f.write(data)
                    # 上传完毕，则退出循环
                    else:
                        recv_size = 0
                        Flag = False
                msg = "Upload successed."
                print(msg)
                conn.sendall(msg.encode())
                f.close()
            elif op =="d":#请求下载
                pre_data = conn.recv(1024).decode()

                if (os.path.exists(base_path + '/' +
pre_data)):
                    conn.send("yes".encode())
                else:
                    conn.send("no".encode())
                    print("请求文件不存在")
                    continue
                path = base_path+ '/' + pre_data
                file_size = os.path.getsize(path)
                # 发送文件名 和 文件大小u
                Informf = (str(pre_data) + '|' +
str(file_size))

                conn.sendall(Informf.encode())

```

```
                send_size = 0
                f = open(path, 'rb')
                Flag = True
                while Flag:
                    if send_size + 1024 > file_size:
                        data = f.read(file_size -
send_size)
                        Flag = False
                    else:
                        data = f.read(1024)
                        send_size += 1024
                    conn.send(data)
                conn.recv(1024)
                f.close()
            elif op =='c':
                pre_data= conn.recv(1024)
                print("收到客户信息：",pre_data.decode())
                conn.send('确认收到！'.encode())
            elif op=="ls":
                for root, dirs, files in
os.walk(base_path + '/' ):
                    # 当前目录路径
                    root_inf = json.dumps(root)
                    conn.send(root_inf.encode())
                    conn.recv(1024)
                    # 当前路径下所有子目录
                    dirs_inf = json.dumps(dirs)
                    conn.send(dirs_inf.encode())
                    conn.recv(1024)
                    files_inf = json.dumps(files)
                    conn.send(files_inf.encode())
                    conn.recv(1024)

if __name__ == '__main__':
    instance = socketserver.ThreadingTCPServer(ip_port,
MyServer)
    instance.serve_forever()
```

- 客户端

```
import socket
import os
```

```python
import json
serverName = 'localhost'
serverPort = 12101

ip_port = (serverName, serverPort)
sk = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sk.connect(ip_port)
base_path="E:\计网\lab\lab2\客户端发送文件/" #客户端文件位置
while True:
    op = input("请选择功能：（u）文件传输 （d）文件下载 (c) 发消息 (ls) 列出服务器文件 （ls-l) 列出本地文件（q）退出 ");
    if op == "u":
        file_name = input('输入文件名(含后缀）： ')
        path = base_path + file_name
        if (not os.path.exists(path)):
            print("本地文件不存在")
            continue
        sk.send(op.encode())

        # 客户端输入要上传文件的路径
        # 获取文件大小
        file_size = os.path.getsize(path)
        # 发送文件名 和 文件大小u
        Informf = (str(file_name) +'|'+ str(file_size))
        sk.send(Informf.encode())
        # 为了防止粘包，将文件名和大小发送过去之后，等待服务端收到，直到从服务端接受一个信号（说明服务端已经收到）
        print(sk.recv(1024).decode())
        #指定了缓冲区的大小为1024字节，因此需要对于文件循环读取，直到文件的数据填满了一个缓冲区，此时将缓冲区数据发送出去，
        # 继续读取下一部分文件；或是当缓冲区未填满，而文件读取完毕，此时应当将这个未满的缓冲区发送给服务器。
        send_size = 0
        f = open(path, 'rb')
        Flag = True
        while Flag:
            if send_size + 1024 > file_size:
                data = f.read(file_size - send_size)
                Flag = False
            else:
                data = f.read(1024)
                send_size += 1024
```

```
            sk.send(data)
        msg = sk.recv(1024)
        print(msg.decode())
        f.close()
        print()
    elif op == 'd':
        sk.send(op.encode())
        file_name = input('输入文件名(含后缀)：')
        sk.send(file_name.encode())

        path = base_path + file_name
        ans = sk.recv(1024).decode()
        if ( ans == "no"):
            print("服务器中找不到该文件")
            continue

        file_inf = sk.recv(1024).decode()
        # 获取请求方法、文件名、文件大小
        file_name, file_size = file_inf.split('|')  # 分割 文件名和大小

        id = 1
        while (os.path.exists(base_path + '/' + file_name)):
            file_name = '1' + file_name
            id += 1
        if (id != 1):
            print('收到文件名重复,已重命名为：', file_name)
        # 已经接收文件的大小
        recv_size = 0
        # 上传文件路径拼接
        file_dir = os.path.join(base_path, file_name)
        f = open(file_dir, 'wb')
        Flag = True
        while Flag:
            # 未上传完毕,
            if int(file_size) > recv_size:
                # 最多接收1024，可能接收的小于1024
                data = sk.recv(1024)
                recv_size += len(data)
                # 写入文件
                f.write(data)
```

```
            # 上传完毕，则退出循环
            else:
                recv_size = 0
                Flag = False
        msg = "Download successed."
        print(msg)
        sk.sendall(msg.encode())
        f.close()
        print()
    elif op=='ls':
        sk.sendall(op.encode())
        json_string = json.loads(sk.recv(1024).decode())
        print("服务器根目录：",json_string)
        sk.send("ok".encode())
        # 防止粘包，给客户端发送一个信号。
        # 当发送内容较大时，由于服务器端的recv（buffer_size）方法中的buffer_size较小，不能一次性完全接收全部内容，
        # 因此在下一次请求到达时，接收的内容依然是上一次没有完全接收完的内容，因此造成粘包现象。
        json_string = json.loads(sk.recv(1024).decode())
        print("根目录下文件夹：",json_string)
        sk.send("ok".encode())
        json_string = json.loads(sk.recv(1024).decode())
        print("目录下文件：", json_string)
        sk.send("ok".encode())
        print()
    elif op=="ls-l":
        for root, dirs, files in os.walk(base_path):
            # 当前目录路径
            print(root)
            # 当前路径下所有子目录
            print(dirs)
            # 当前路径下所有非目录子文件
            print(files)
            print()
    elif op=="c":
        sk.send(op.encode())
        messa=input("请输入消息：")
        sk.send(messa.encode())
        print('服务器回应：',sk.recv(1024).decode())

    elif op=='q':
```

```
            print("连接断开")
            break
        else:
            print('功能选择有误，请重输！')
            continue
sk.close()
```

实验总结

通过本次实验，对socket编程有了初步的了解，学习了如何使用套接字采用TCP进行数据的收发、用UDP进行数据的收发，socket是应用层与TCP/IP协议中间的抽象层，作为一组接口，其将复杂的TCP/IP协议隐藏在socket接口后面，减轻了程序员编程的负担；同时也对比了多线程、线程池两种不同的多线程通信方案的实现方式和差别，相比于多线程方式，线程池预先创建多个线程，在需要新线程的时候将提前创建的线程直接取出来使用，在效率上是比多线程实现方式更优的，但两种方式都有自己的应用场景；最后利用python语言尝试编写了互传文件的程序，客户端与客户端之间的通信(聊天/传文件)本质上是一种通过服务器转发的方式，正确处理它们之间的关系非常重要。