# 一、tcp

## 1.socket编程原理

|  |
| --- |
|  |

## 2.流程说明

|  |
| --- |
|  |

## 3.最简单实例，分为服务器和客户端

需要先运行服务器端因为它需要监听，然后启动客户端，客户端需要先发送一条消息，然后服务器回复信息，通话结束

|  |  |
| --- | --- |
| # server.py  import socket  recv\_size =1024 # 创建服务器 tcp\_svr = socket.socket(socket.AF\_INET,socket.SOCK\_STREAM) # 绑定端口 tcp\_svr.bind(('127.0.0.1', 8000)) # 监听端口 tcp\_svr.listen(1000) # 接受用户连接,返回连接和对方地址 con,addr = tcp\_svr.accept() # 接收消息 data = con.recv(recv\_size) print(data.decode('utf-8')) msg = input("请输入发送信息: ") # 发送信息 con.send(msg.encode('utf-8')) print("done!!") con.close() tcp\_svr.close() | # Client.py  import socket  recv\_size = 1024 # 创建客户端 client = socket.socket(socket.AF\_INET,socket.SOCK\_STREAM) # 连接服务器 client.connect(('127.0.0.1',8000)) # 发送信息 msg = input("请输入发送给服务器的信息:") client.send(msg.encode("utf-8")) # 接收服务器信息 data = client.recv(recv\_size) print(data.decode('utf-8')) print("done!") client.close() |

## 4.基于tcp/ip的循环chat程序

|  |  |
| --- | --- |
| # tcp\_server\_loop.py  import socket  recv\_size =1024 # 创建服务器 tcp\_svr = socket.socket(socket.AF\_INET,socket.SOCK\_STREAM) # 绑定端口 tcp\_svr.bind(('127.0.0.1', 8000)) # 监听端口 tcp\_svr.listen(1000) while True: # 保证服务器可以在一个客户端关闭后仍然可以和别的客户端通讯  print("开始新会话。。。")  # 接受用户连接,返回连接和对方地址  con,addr = tcp\_svr.accept()  while True:  try:  # 接收消息  data = con.recv(recv\_size)  if not data: # 这个在Linux系统下面需要写，在windows系统下面可以不写  break  if data.decode('utf-8') == 'q':  con.send('q'.encode('utf-8'))  break  print(data.decode('utf-8'))  msg = input("请输入发送信息: ")  # 发送信息  con.send(msg.encode('utf-8'))  except Exception as e:  print('程序异常：',e)   print("done!!")  con.close()  tcp\_svr.close() | #tcp\_client\_loop.py  import socket  recv\_size = 1024 # 创建客户端 client = socket.socket(socket.AF\_INET,socket.SOCK\_STREAM) # 连接服务器 client.connect(('127.0.0.1',8000)) while True:  # 发送信息  msg = input("请输入发送给服务器的信息:")  client.send(msg.encode("utf-8"))  # 接收服务器信息  data = client.recv(recv\_size)  if data.decode('utf-8') == 'q':  client.send('q'.encode('utf-8'))  break  print(data.decode('utf-8'))  print("done!") client.close() |

## 5.基于tcp的远程命令调用

|  |  |
| --- | --- |
| # rpc\_server.py  import socket import subprocess import struct  recv\_size =1024 # 创建服务器 tcp\_svr = socket.socket(socket.AF\_INET,socket.SOCK\_STREAM) tcp\_svr.setsockopt(socket.SOL\_SOCKET,socket.SO\_REUSEADDR,1) # #加入这条socket配置，重用IP和端口，还可以在操作系统上配置ip和端口重用 # 绑定端口 tcp\_svr.bind(('127.0.0.1', 8000)) # 监听端口 tcp\_svr.listen(1000) while True: # 保证服务器可以在一个客户端关闭后仍然可以和别的客户端通讯  print("开始新会话。。。")  # 接受用户连接,返回连接和对方地址  con,addr = tcp\_svr.accept()  while True:  try:  # 接收消息  data\_cmd = con.recv(recv\_size)  if not data\_cmd: # 这个在Linux系统下面需要写，在windows系统下面可以不写  break  # print(data\_cmd.decode('utf-8'))  # 执行命令，shell=True,表示指令为字符串，False表示指令为序列  # subprocess.PIPE返回一个文件展示给用户看  result = subprocess.Popen(data\_cmd.decode('utf-8'),shell=True,  stderr=subprocess.PIPE,  stdout=subprocess.PIPE,  stdin=subprocess.PIPE)  err = result.stderr.read() # 通过res.stderr.read()读取错误消息  if err:  res\_data = err  else:  res\_data = result.stdout.read() # 通过res.stdout.read()读取标准输出消息  if not res\_data: # 有些系统命令是没有返回值的，需要处理一下  res\_data = '执行成功'.encode('gbk') # 防止操作系统执行命令后，返回值为空时，为客户端返回此消息,这里的编码是gbk因为cmd窗口就是gbk编码的  \_len = len(res\_data)  \_len\_str = struct.pack('i',\_len)  con.send(\_len\_str)  con.send(res\_data)  except Exception as e:  print('程序异常：',e)   print("done!!")  con.close()  tcp\_svr.close() | # rpc\_client.py  import socket import struct  recv\_size = 1024 # 创建客户端 client = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) # 连接服务器 client.connect(('127.0.0.1', 8000)) while True:  # 发送信息  cmd = input("请输入命令:")  if not cmd:  continue # 如果输入为空，跳过此次发送  if 'exit' == cmd:  break # 输入exit断开连接  # 发送命令给服务器  client.send(cmd.encode("utf-8"))  # 接收服务器信息  len\_data = client.recv(4)  header\_data = struct.unpack('i', len\_data)[0]  # 获取调用返回数据  res\_len = 0  res\_msg = b''  while res\_len < header\_data:  res\_msg += client.recv(recv\_size)  res\_len = len(res\_msg)  print(res\_len)  print("执行结果：" + res\_msg.decode('gbk')) # cmd窗口返回的内容是gbk编码的 print("done!") client.close() |

## 6.模拟远程文件下载服务器

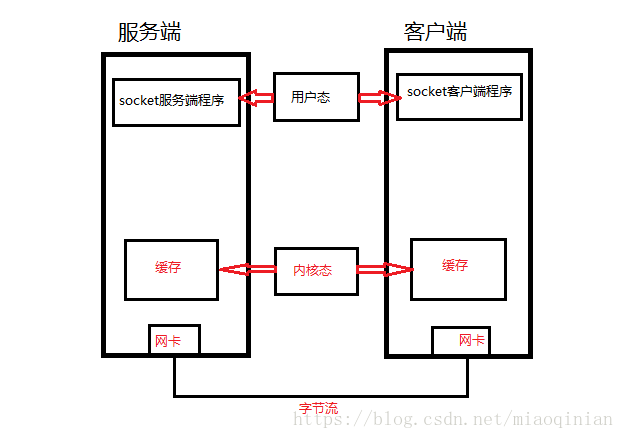
先启动服务端，再启动客户端，然后用客户端往服务端发送文件名，如果文件存在，服务器就读取文件内容并且发送给客户端，先发送数据的长度，再发送数据内容。

客户端会判断数据的长度如果长度为零说明文件不存在，跳过，如果不为0，说明有数据，需要接收并且保持到本地

|  |  |
| --- | --- |
| #file\_server.py  import socket import os import struct  recv\_size = 1024 # 创建服务器 tcp\_svr = socket.socket(socket.AF\_INET,socket.SOCK\_STREAM) tcp\_svr.setsockopt(socket.SOL\_SOCKET,socket.SO\_REUSEADDR,1) # #加入这条socket配置，重用IP和端口，还可以在操作系统上配置ip和端口重用 # 绑定端口 tcp\_svr.bind(('127.0.0.1', 8000)) # 监听端口 tcp\_svr.listen(1000) while True: # 保证服务器可以在一个客户端关闭后仍然可以和别的客户端通讯  print("开始新会话。。。")  # 接受用户连接,返回连接和对方地址  con,addr = tcp\_svr.accept()  print(addr)  while True:  try:  # 接收消息  file\_name = con.recv(recv\_size)  print(file\_name.decode('utf-8'))  if not file\_name: # 这个在Linux系统下面需要写，在windows系统下面可以不写  break  file\_data = b''  if not os.path.exists(file\_name):  file\_data = ''  else:  with open(file\_name,'rb') as f:  file\_data = f.read()  \_len = len(file\_data)  \_len\_str = struct.pack('i',\_len)  con.send(\_len\_str)  con.send(file\_data)  except Exception as e:  print('程序异常：',e)   print("done!!")  con.close()  tcp\_svr.close() | # file\_client.py  import os import socket import struct  recv\_size = 1024 # 创建客户端 client = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) # 连接服务器 client.connect(('127.0.0.1', 8000)) while True:  # 发送信息  cmd = input("文件名:").strip()  if not cmd:  continue # 如果输入为空，跳过此次发送  if 'exit' == cmd:  break # 输入exit断开连接  # 发送命令给服务器  client.send(cmd.encode("utf-8"))  # 接收服务器信息  len\_data = client.recv(4)  header\_data = struct.unpack('i', len\_data)[0]  if header\_data == 0: # 返回空字符串上面文件不存在，跳过本次循环  print("文件不存在。。。")  continue  else:  # 获取调用返回数据  res\_len = 0  res\_msg = b''  with open(cmd,'wb') as f:  while res\_len < header\_data:  res\_msg = client.recv(recv\_size)  res\_len += len(res\_msg)  # print(res\_len)  f.write(res\_msg)  print('file saved...')  print("done!") client.close() |

## 注意这里有一个粘包的问题需要解决：

只有TCP有粘包问题，而[UDP](https://so.csdn.net/so/search?q=UDP&spm=1001.2101.3001.7020)永远不会粘包。我们先掌握一个socket收发消息的原理：



    服务端可以1kb，1kb地发向客户端送数据，客户端的应用程序可以在缓存当中2kb,2kb地取走数据，当然也可以更多，或都更少。也就是说，应用程序看到的数据是来个整体。**或者说是一个流**。一条消息有多少字节对应用程序是不可见的，**TCP协议是面向流的协议，这就是它容易粘包的问题原因。**

    所谓粘包问题主要还是因为接收方不知道消息之间的界限，不知道一个消息要提取多少字节的数据所造成的。

    此外，发送方引起的粘粘包是由TCP协议本身造成的，TCP为提高传输效率，发送方往往收集到足够多的数据后才一个TCP段。若连续几次需要发送的数据都很少，通常TCP会根据（Nagle）优化算法,把这些数据合成一个TCP段后发出去，这样接收方就收到了粘包数据。

## 粘包情况一：

发送端要等缓冲区满才发送出去（即发送的数据量少且时间间隔短，会合到一起），造成粘包。

1、服务端

**from socket import \***

**server=socket(AF\_INET,SOCK\_STREAM)**

**server.bind(('127.0.0.1',8080))**

**server.listen(5)**

**conn,client\_addr=server.accept()**

**res1=conn.recv(1024)**

**print('第一次:',res1)**

**res2=conn.recv(1024)**

**print('第二次:',res2)**

**conn.close()**

**server.close()**

2、客户端

from socket import \*

client=socket(AF\_INET,SOCK\_STREAM)

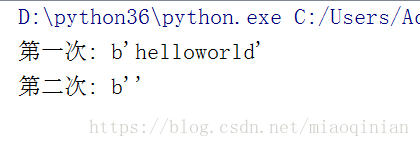
client.connect(('127.0.0.1',8080))

client.send(b'hello')

client.send(b'world')

client.close()

先启动服务端，后再启动客户端，服务端得到的结果为：



## 粘包情况二：

客户端发关了一段数据，服务端只收了一小部分，服务端下次再收的时候不是从缓冲区拿上次遗留的数据，产生粘包。

情况一的，客户端不变，服务端略作修改，如下：

**from socket import \***

**server=socket(AF\_INET,SOCK\_STREAM)**

**server.bind(('127.0.0.1',8080))**

**server.listen(5)**

**conn,client\_addr=server.accept()**

**res1=conn.recv(2)**

**print('第一次:',res1)**

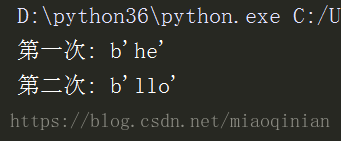
**res2=conn.recv(3)**

**print('第二次:',res2)**

**conn.close()**

**server.close()**

先启动服务端，后再启动客户端，服务端得到的结果为：



    现在我们来想一下如何处理粘包的方法。粘包的问题根源是接收端不知发送端将要传送的字节流，所以我们要让发送端在发送数据前，把要发送的字节流总大小让接收端知晓，然后接收端来个循环将其全部接收。这种方法比较低级，因为程序的运行速度运快于网络传输速度，所以在发送一段字节前，先发送该字节流的长度，这种方式会放大网络延迟带来的性能损耗。

## 目前比较合理的处理方法是：

**为字节流加上一个报头，将这个报告做成字典，字典里包含将要发送的真实数据详细信息。将这个字典JSON序列化，然后用struck（不了解struck可看最后补充）将序列化后的数据长度打包成4个字节（4个字节完全够用）**

**发送时：**先发报头长度，再编码报头内容后发送，最后发真实数据。

**接收时：**用struct取出报头长度，然后根据长度取出报头，解码，反序列化。最后从反序列化的结果中取出待取数据的详细信息，然后取真实的数据内容。

来看实现的代码：

### 1、服务端

**from socket import \***

**import subprocess**

**import struct**

**import json**

**server=socket(AF\_INET,SOCK\_STREAM)**

**server.bind(('127.0.0.1',8080))**

**server.listen(5)**

**while True:**

**conn,client\_addr=server.accept()**

**print(conn,client\_addr)#(连接对象，客户端的ip和端口)**

**while True:**

**try:**

**cmd=conn.recv(1024)**

**obj=subprocess.Popen(**

**cmd.decode('utf-8'),**

**shell=True,**

**stdout=subprocess.PIPE,**

**stderr=subprocess.PIPE**

**)**

**stdout=obj.stdout.read()**

**stderr=obj.stderr.read()**

**#1、制作报头**

**header\_dic={**

**'total\_size':len(stdout)+len(stderr),**

**'md5':'dgdsfsdfdsdfsfewrewge',**

**'file\_name':'a.txt'**

**}**

**header\_json=json.dumps(header\_dic)**

**header\_bytes=header\_json.encode('utf-8')**

**#2、先发送报头的长度**

**header\_size=len(header\_bytes)**

**conn.send(struct.pack('i',header\_size))**

**#3、发送报头**

**conn.send(header\_bytes)**

**#4、发送真实的数据**

**conn.send(stdout)**

**conn.send(stderr)**

**except ConnectionResetError:**

**break**

**conn.close()**

**server.close()**

### 2、客户端

**from socket import \***

**import json**

**import struct**

**client=socket(AF\_INET,SOCK\_STREAM)**

**client.connect(('127.0.0.1',8080))**

**while True:**

**cmd=input(">>:").strip()**

**if not cmd:continue**

**client.send(cmd.encode('utf-8'))**

**#1、接收报文头的长度**

**header\_size=struct.unpack('i',client.recv(4))[0]**

**#2、接收报文**

**header\_bytes=client.recv(header\_size)**

**#3、解析报文**

**header\_json=header\_bytes.decode('utf-8')**

**header\_dic=json.loads(header\_json)**

**print(header\_dic)**

**#4、获取真实数据的长度**

**totol\_size=header\_dic['total\_size']**

**#5、获取数据**

**recv\_size=0**

**res=b''**

**while recv\_size<totol\_size:**

**recv\_date=client.recv(1024)**

**res+=recv\_date**

**recv\_size+=len(recv\_date)**

**print(res.decode('gbk'))**

**client.close()**

### 补充struct模块：

**import struct**

**#struct是用来将整型的数字转成固定长度的bytes.**

**import json**

**header\_dic={**

**'total\_size':32322,**

**'md5':'gdssfsfsdfsf',**

**'filename':'a.txt'**

**}**

**#1、将报头字典序列化。**

**header\_json=json.dumps(header\_dic)**

**#2、将序列后的字典转成字节**

**header\_bytes=header\_json.encode('utf-8')**

**#3、获取序列的字字典转成字节的个数**

**header\_size=len(header\_bytes)**

**print(header\_size)**

**#4、将这个个数转成固字长度的字节表示**

**obj=struct.pack('i',header\_size)**

**print(obj,len(obj))**

**#、这个固定长度的字节经过反转后是一个元组。**

**res=struct.unpack('i',obj)**

**#、通过按索取值就可等到报头字典长度。**

**header\_size=res[0]**

## 剖析和解决Python中网络粘包的正确姿势

## 1、粘包及其成因

### 1.1、粘包产生

先来看一个案例，单进程启动一个tcp socket通信，从服务端发送两次数据到客户端。

* 服务端tcp\_socket\_server1.py

**import socket**

**sk = socket.socket()**

**sk.bind(("127.0.0.1", 8888))**

**sk.listen()**

**conn, address = sk.accept()**

**conn.send(b'ab')**

**conn.send(b'cd')**

上面的服务端代码创建了一个socket对象，监听的ip地址是本地的127.0.0.1，端口是6666，并且在程序启动并正常和客户端连接进行阻塞后，分别发送了ab和cd两次数据。

* 客户端tcp\_socket\_client1.py

**import socket**

**import time**

**sk = socket.socket()**

**sk.connect(("127.0.0.1", 8888))**

**time.sleep(5)**

**bs1 = sk.recv(1024)**

**bs2 = sk.recv(1024)**

**print(bs1)**

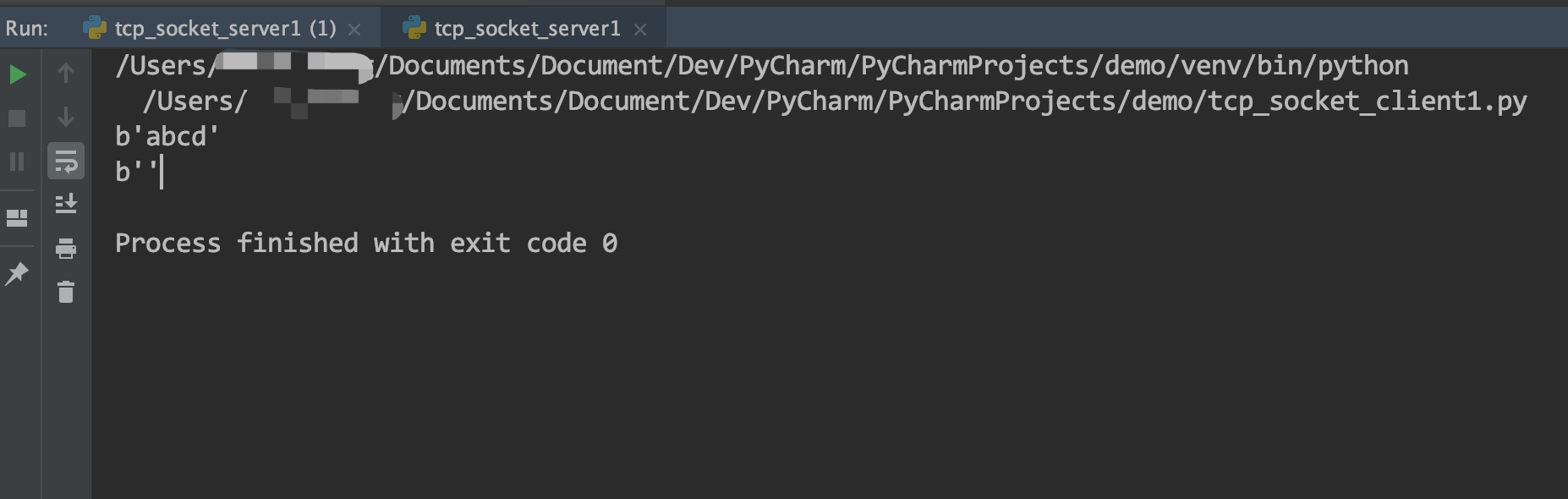
**print(bs2)**

上面的客户端代码创建了一个socket对象，连接的ip地址是127.0.0.1，端口是6666，并且在程序启动并正常和客户端连接后，让程序暂停了5秒钟，然后分别接收了bs1和bs2两次数据，每次接收的数据大小是1024个字节。

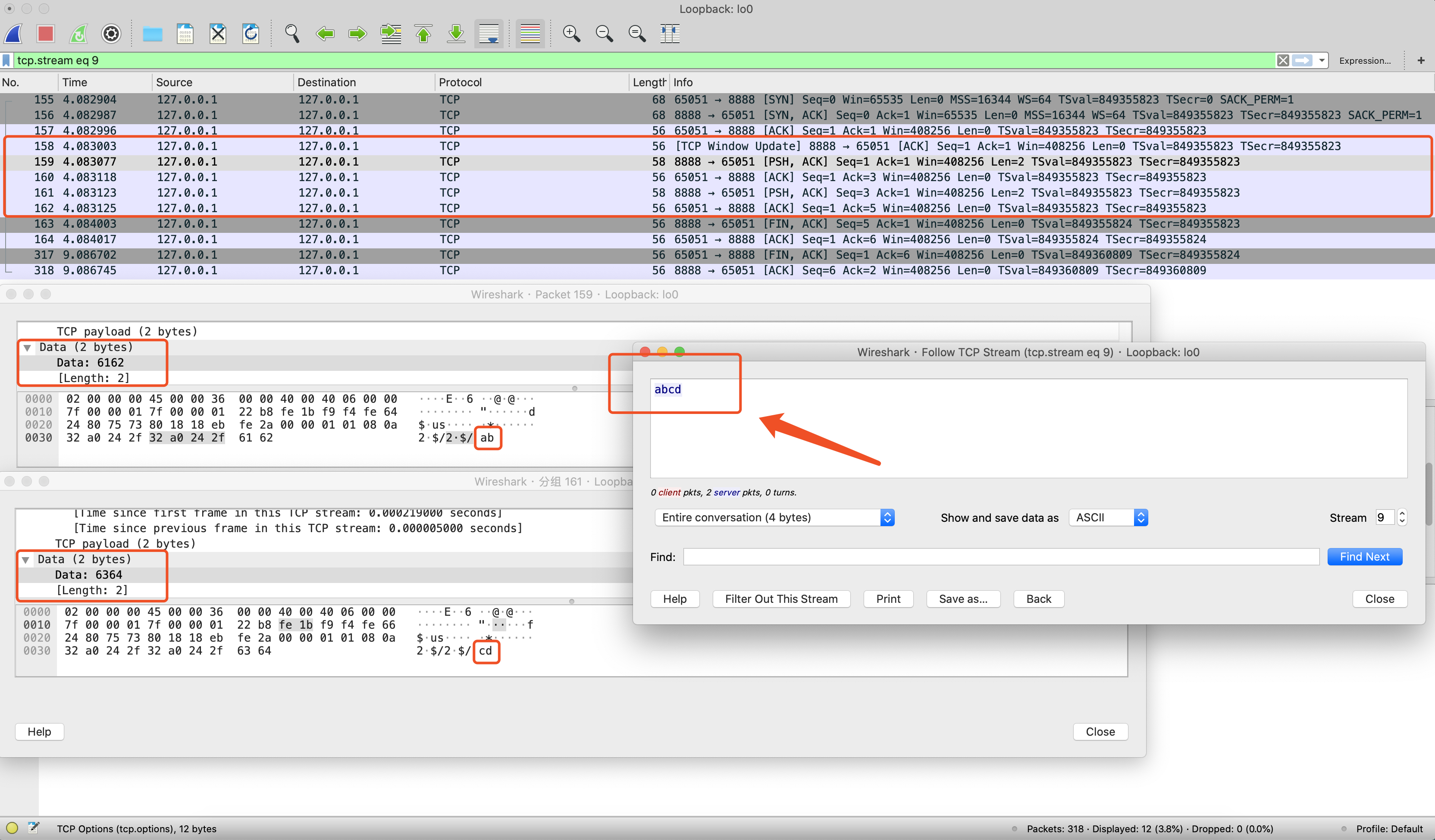
启动server端的程序，在终端查看本地端口情况，server端启动正常并且持续保持着监听



启动client端程序，查看终端的输出，检查server端发送过来的数据



可以发现，在client的终端界面打印出来的两个数据包，第二个数据为空，而第一个打印的是两个数据合在一起的效果，很奇妙，明明我们发了两次，接收的也是两次。  
为了证明事故现场的真实性，我调取了相关“监控”，没错，抓包，如下所示



通过上图可以发现，在一次tcp通信中包含了正常的三次握手(前3个数据包)和四次挥手(后4个数据包)，其中第158号数据包的作用简单来说就是用于连接上对等双方之间的流控制。具体可参考[此文章](http://stackoverflow.com/questions/1466307/what-is-a-tcp-window-update)，第159和160号数据包是第一次双方数据通信，第161和162号数据包是第二次双方数据通信。两次数据包中分别包含了ab和cd两个data。但是通过追踪数据流可以发现，两次的数据ab和cd被合并到了一起，因此接收方第一次就一次性收到了所有的数据，这与上面程序控制台的输出情况是吻合的。

### 1.2、粘包产生的原因

1.发送端需要等缓冲区满才发送出去，造成粘包(发送数据时间间隔很短，数据很小，会合到一起，产生粘包)，这样接收端，就难于分辨出来了，必须提供科学的拆包机制。 即面向流的通信是无消息保护边界的。

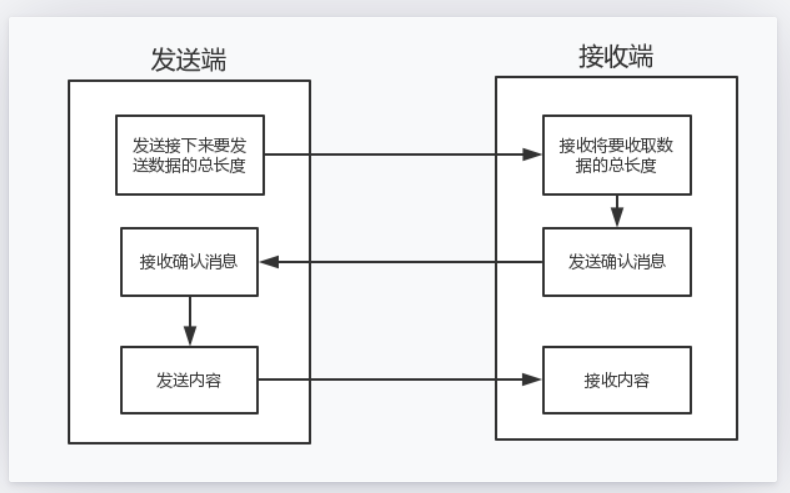
2.接收方不及时接收缓冲区的包，造成多个包接收（客户端发送了一段数据，服务端只收了一小部分，服务端下次再收的时候还是从缓冲区拿上次遗留的数据，产生粘包）

粘包问题主要还是因为接收方不知道消息之间的界限，不知道一次性提取多少字节的数据所造成的。

3.udp通信不会产生粘包现象。原因如下：

* UDP（user datagram protocol，用户数据报协议）是无连接的，面向消息的，提供高效率服务。  
  不会使用块的合并优化算法， 由于UDP支持的是一对多的模式，所以接收端的skbuff(套接字缓冲区）采用了链式结构来记录每一个到达的UDP包，在每个UDP包中就有了消息头（消息来源地址，端口等信息），这样，对于接收端来说，就容易进行区分处理了。 即面向消息的通信是有消息保护边界的。
* 对于空消息：tcp是基于数据流的，于是收发的消息不能为空，这就需要在客户端和服务端都添加空消息的处理机制，防止程序卡住，而udp是基于数据报的，即便是你输入的是空内容（直接回车），也可以被发送，udp协议会帮你封装上消息头发送过去。
* 不可靠不黏包的udp协议：udp的recvfrom是阻塞的，一个recvfrom(x)必须对唯一一个sendinto(y),收完了x个字节的数据就算完成,若是y;x数据就丢失，这意味着udp根本不会粘包，但是会丢数据，不可靠。

## 2、尝试解决粘包

粘包问题的现象及成因如上所示，如何解决上面的黏包问题？其实也很简单，之所以出现黏包就是因为数据没有边界，直接把两个包混合成了一个包，接收端不知道发送端将要传送的字节流的长度。所以解决粘包问题最先能想到的就是在发送正式的数据之前先发送一个包，这个包的内容就是告诉客户端将要发送的正式数据包的长度是多少（这个数据包有多大）， 客户端接收数据的时候，先读取该数据包的大小，然后再按照这个大小读取一定长度的数据，最终就不会产生黏包了。  
 

### 2.1、指定数据包的长度

按照上述思路，对文章开始给出的案例进行改造，在发送数据包之前先发一个包，包的内容是声明正式数据包（下一个数据包）的长度。

* 服务端tcp\_socket\_server2.py

**import socket**

**sk = socket.socket()**

**sk.bind(("127.0.0.1", 8888))**

**sk.listen()**

**conn, address = sk.accept()**

**conn.send(b'2') # 预先告诉对方下一个数据包的长度**

**conn.send(b'ab')**

**conn.send(b'cd')**

上面的服务端代码在发送数据包的开始先发送了一个数据包，内容是2，占1个字节，目的是预先告诉对方下一个数据包的长度。然后再分别发送两次数据包ab和cd。

* 客户端tcp\_socket\_client2.py

**import socket**

**import time**

**sk = socket.socket()**

**sk.connect(("127.0.0.1", 8888))**

**time.sleep(5)**

**bs = sk.recv(1) # 先接收到下一个数据包的长度**

**len = int(bs.decode("utf-8"))**

**bs1 = sk.recv(len) # 接受对应长度的字节**

**bs2 = sk.recv(1024)**

**print(bs1)**

**print(bs2)**

上面的客户端代码在正式接收数据包之前，先接收了大小为1个字节的数据包，表示接收服务端发来的第一个字节的数据包，此数据包对应的数据正是表示的服务端将要发送的正式数据包的长度，由于发送过来的是bytes类型，需要进行解码以及转换为int类型，转换后的数值就是下一个数据包的长度，然后分别接收了bs1和bs2两次数据，第一次接收的数据包长度是前面收到的长度，第二次接收的数据大小是1024个字节。

按照先后启动server端程序和client端程序，检查server端发送过来的数据，可以发现，本次收到的服务端发送过来的两次数据正常接收，没有了粘包现象。

### 2.2、固定数据包的长度

上一步提到的指定数据包长度的做法，解决了粘包问题。但是，不难想象，如果我们发送的数据包有多次，那么每发送一次数据包之前都需要发送一个声明着下一数据包长度的包，才能让客户端正常接收。有没有可以改进的方案呢，当然是有的。数据包的长度是可变的，很容易想到的一个解决方案就是我把表示正式数据包（下一个数据包）长度的这个数据包的长度固定下来。那么对服务端来说：首先发送固定长度的数据包来表示下一个数据包的长度，然后下一个数据包发送数据；对客户端来说，首先接收的第一个包是固定长度的，然后接收的第二个包就是正式的数据包。

* 服务端tcp\_socket\_server3.py

**import socket**

**sk = socket.socket()**

**sk.bind(("127.0.0.1", 8888))**

**sk.listen()**

**conn, address = sk.accept()**

**msg = input("请输入你要发送的数据>>>:")**

**bs = msg.encode('utf-8')**

**chang = len(bs)**

**chang\_str = format(chang, "04d")**

**conn.send(chang\_str.encode("utf-8"))**

**conn.send(bs)**

上面的服务端代码通过运行时接收用户输入来计算数据包的长度并且将表示数据包长度的这个数据包编码以及格式化成4个字节发送个客户端，然后再发送接收到的数据给客户端。

* 客户端tcp\_socket\_client3.py

**import socket**

**import time**

**sk = socket.socket()**

**sk.connect(("127.0.0.1", 8888))**

**time.sleep(5)**

**bs = sk.recv(4)**

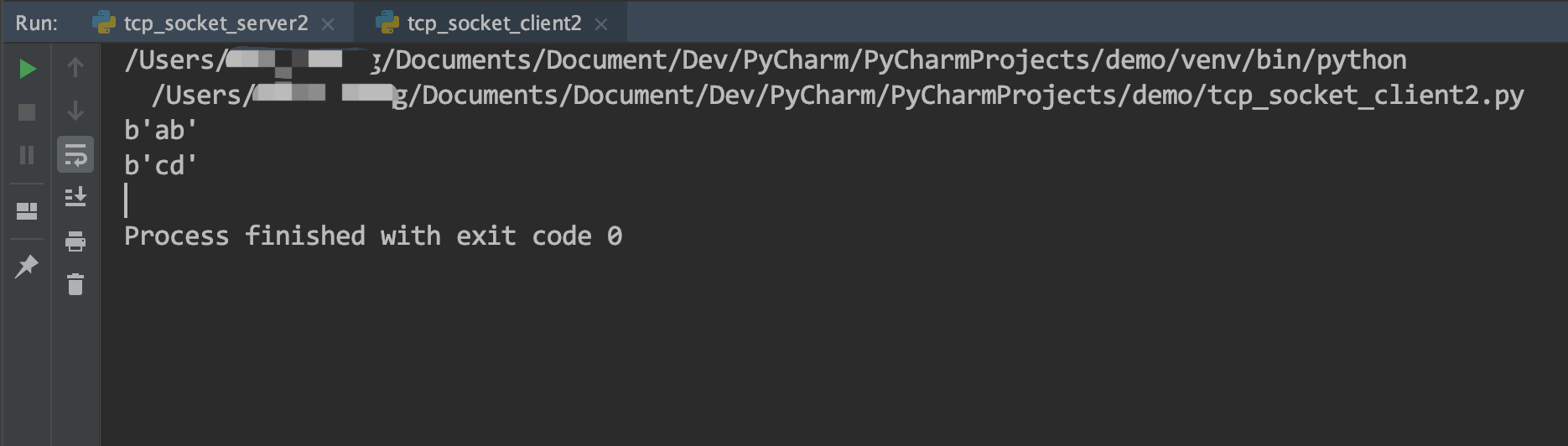
**chang = int(bs.decode("utf-8"))**

**msg = sk.recv(chang)**

**print(msg.decode("utf-8"))**

上面的客户端代码先接收了4个字节的数据包，然后通过对数据解码得到了其中的内容，这个内容就是表示正式数据包（下一个数据包）的长度，然后接收这个长度的数据包。

按照先后启动server端程序和client端程序，检查server端发送过来的数据，效果如下。



### 2.3、用函数实现多次调用发送数据

通过上面的小小改进，就可以在发送第一个数据包是固定长度，客户端接收的时候也是固定的长度，但是要想轻松的发送多次数据包，这种办法需要每次都执行一遍，更为友好的解决办法是可以将服务端发送数据包的代码以及客户端接收数据包的代码各自提取出一个函数，然后利用函数一次定义、多次调用的好处，每次发送数据，通过函数传参就可以了。此时不管发送多少次，代码只需要写一次就好了。

* 服务端tcp\_socket\_server4.py

**import socket**

**sk = socket.socket()**

**sk.bind(("127.0.0.1", 8888))**

**sk.listen()**

**conn, address = sk.accept()**

**def my\_send(msg):**

**bs = msg.encode('utf-8')**

**chang = len(bs)**

**chang\_str = format(chang, "04d")**

**conn.send(chang\_str.encode("utf-8"))**

**conn.send(bs)**

**my\_send(input(">>>"))**

**my\_send(input(">>>"))**

**my\_send(input(">>>"))**

**客户端tcp\_socket\_client4.py**

**import socket**

**import time**

**sk = socket.socket()**

**sk.connect(("127.0.0.1", 8888))**

**time.sleep(5)**

**def my\_recv():**

**bs = sk.recv(4)**

**chang = int(bs.decode("utf-8"))**

**msg = sk.recv(chang)**

**print(msg.decode("utf-8"))**

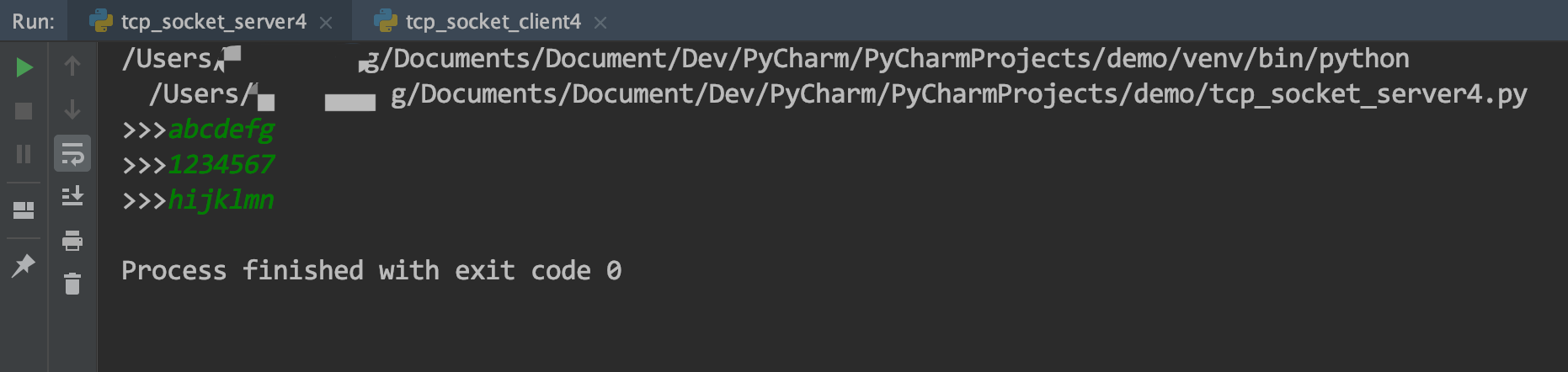
**my\_recv()**

**my\_recv()**

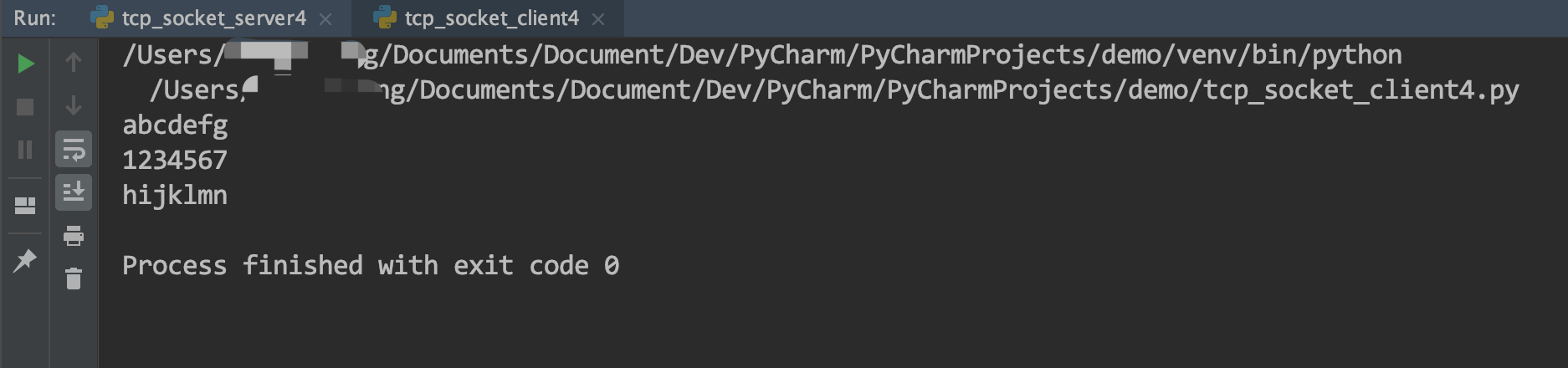
**my\_recv()**

按照先后启动server端程序和client端程序，检查server端发送过来的数据，效果如下。

服务端



客户端



## 3、解决粘包问题的正确姿势

按照第2章节中解决和升级解决粘包问题的方案，其实还是有不足的地方，虽然最终写成了函数，发送数据的时候调用就好了，但是程序的运行速度远快于网络传输速度，所以在发送一段字节前，先用send去发送该字节流长度，这种方式会放大网络延迟带来的性能损耗。

事实上python提供了一个很好用的模块struct来帮我们解决这个问题，这个模块可以把要发送的数据长度转换成固定长度的字节。这样客户端每次接收消息之前只要先接受这个固定长度字节的内容看一看接下来要接收的信息大小，那么最终接受的数据只要达到这个值就停止，就能刚好不多不少的接收完整的数据了。

### 3.1、struct模块功能示例

struct模块用来解决bytes和其他二进制数据类型的转换。更多关于struct模块的说明，可以[参考此文章](https://www.liaoxuefeng.com/wiki/1016959663602400/1017685387246080)

* 把数字打包成字节

下面的程序先对字符串abcdefg进行了编码成为了字节bytes类型，然后对这个bytes类型的字符串计算了长度，并且按照struct模块把表示长度的数字打包转换成了固定长度的字节，终端输出字节的长度是4，这个4由打包时指定的参数i控制，i表示是int类型，而一个int类型占4个字节，每个字节占8个bit比特位，因此总共可以表示的数字大小是2^32，也就是4GB大小的长度，而装这个长度是根据发送的信息来计算的。因此，在打包参数为i的情况下，最多能够表示一次发送的数据是4GB，通常情况下，我们一次发送的数据包长度是不会超过4GB的。

import struct

**msg = "abcdefg"**

**bs = msg.encode('utf-8')**

**bs\_struct\_len = struct.pack("i", len(bs))**

**print(bs\_struct\_len) # 输出 b'\x07\x00\x00\x00'**

**print(len(bs\_struct\_len)) # 输出 4**

* 把字节还原成数字

下面的程序把上面通过struct进行打包得到的4个字节的bytes类型数据进行了解包，解包出来返回的值是一个元祖，通过取元祖的第0个元素，最终得到了上面最开始的字符串的长度7。

**bs = b'\x07\x00\x00\x00'**

**tuple = struct.unpack("i", bs)**

**print(tuple) # 输出 (7,)**

**num = struct.unpack("i", bs)[0]**

**print(num) # 输出 7**

**print(type(num)) # 输出 int**

通过struct可以把数字转换成定长的字节类型数据，也可以把这个转换后的定长字节类型数据还原成数字。对于粘包问题来说，就可以利用这个功能，来表示数据包的长度，服务端先将表示数据包长度的这个数据包转换成字节类型，发送给客户端，客户端再通过解包还原成数字。

### 3.2、struct优雅的解决粘包问题

struct模块的功能如上所示，因此可以将前面发送固定描述数据包长度的那个数据包用struct来代替。

* 服务端struct\_server1.py

**import socket**

**import struct**

**sk = socket.socket()**

**sk.bind(("127.0.0.1", 8999))**

**sk.listen()**

**conn, address = sk.accept()**

**msg = input(">>>:")**

**bs = msg.encode("utf-8")**

**msg\_len\_bs = struct.pack("i", len(bs))**

**conn.send(msg\_len\_bs)**

**conn.send(bs)**

**msg = input(">>>:")**

**bs = msg.encode("utf-8")**

**msg\_len\_bs = struct.pack("i", len(bs))**

**conn.send(msg\_len\_bs)**

**conn.send(bs)**

上面的服务端代码，计算得到用户输入内容的长度，然后把这个数据包通过struct打包发送给了客户端，最后发送数据包内容，程序连续执行了两次。

* 客户端struct\_client1.py

**import socket**

**import struct**

**import time**

**sk = socket.socket()**

**sk.connect(("127.0.0.1", 8999))**

**time.sleep(5)**

**bs\_msg\_len = sk.recv(4)**

**bs\_len = struct.unpack("i", bs\_msg\_len)[0]**

**bs = sk.recv(bs\_len)**

**print(bs.decode("utf-8"))**

**bs\_msg\_len = sk.recv(4)**

**bs\_len = struct.unpack("i", bs\_msg\_len)[0]**

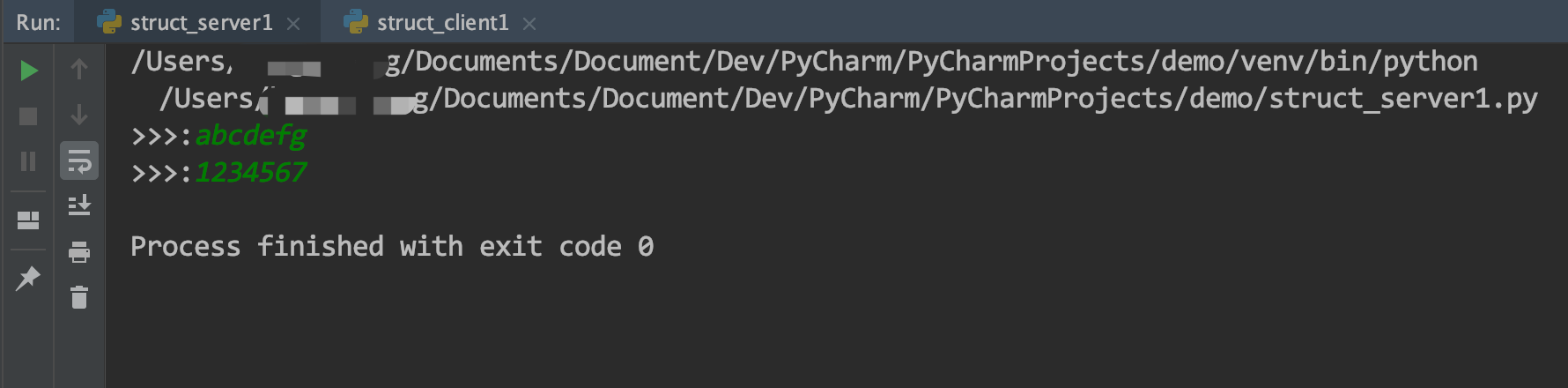
**bs = sk.recv(bs\_len)**

**print(bs.decode("utf-8"))**

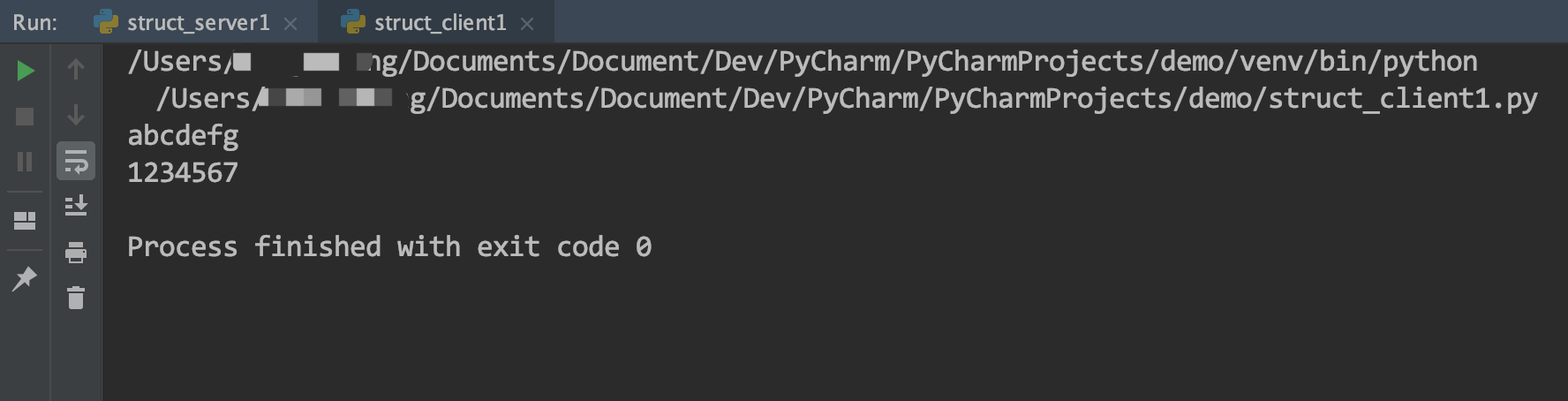
上面的客户端代码，先接收了长度的数据包，然后将这个数据包通过struct解包，然后再接收前面解包得到的数字大小的数据包，程序连续执行了两次。

按照先后启动server端程序和client端程序，检查server端发送过来的数据，效果如下。

服务端



客户端



### 3.3、struct模块功能函数化

通过上面的struct功能分别连续发送和接收了两次数据，可以将struct打包后发送以及解包后接收的功能写成函数来实现，以后程序发送数据包时只需要调用函数名并传入一个参数（数据包内容）即可。

函数体struct\_func.py

**import struct**

**def my\_send(sk, msg):**

**msg\_bs = msg.encode("utf-8")**

**msg\_struct\_len = struct.pack("i", len(msg\_bs))**

**sk.send(msg\_struct\_len)**

**sk.send(msg\_bs)**

**def my\_recv(sk):**

**msg\_struct\_len = sk.recv(4)**

**msg\_len = struct.unpack("i", msg\_struct\_len)[0]**

**data = sk.recv(msg\_len)**

**return data.decode("utf-8")**

服务端发送数据struct\_server2.py

**import socket**

**import struct\_func as msu**

**sk = socket.socket()**

**sk.bind(("127.0.0.1", 8123))**

**sk.listen()**

**conn, addr = sk.accept()**

**msu.my\_send(conn, "abcdefg")**

**msu.my\_send(conn, "1234567")**

客户端接收数据struct\_client2.py

**import socket**

**import struct\_func as msu**

**import time**

**sk = socket.socket()**

**sk.connect(("127.0.0.1", 8123))**

**time.sleep(5)**

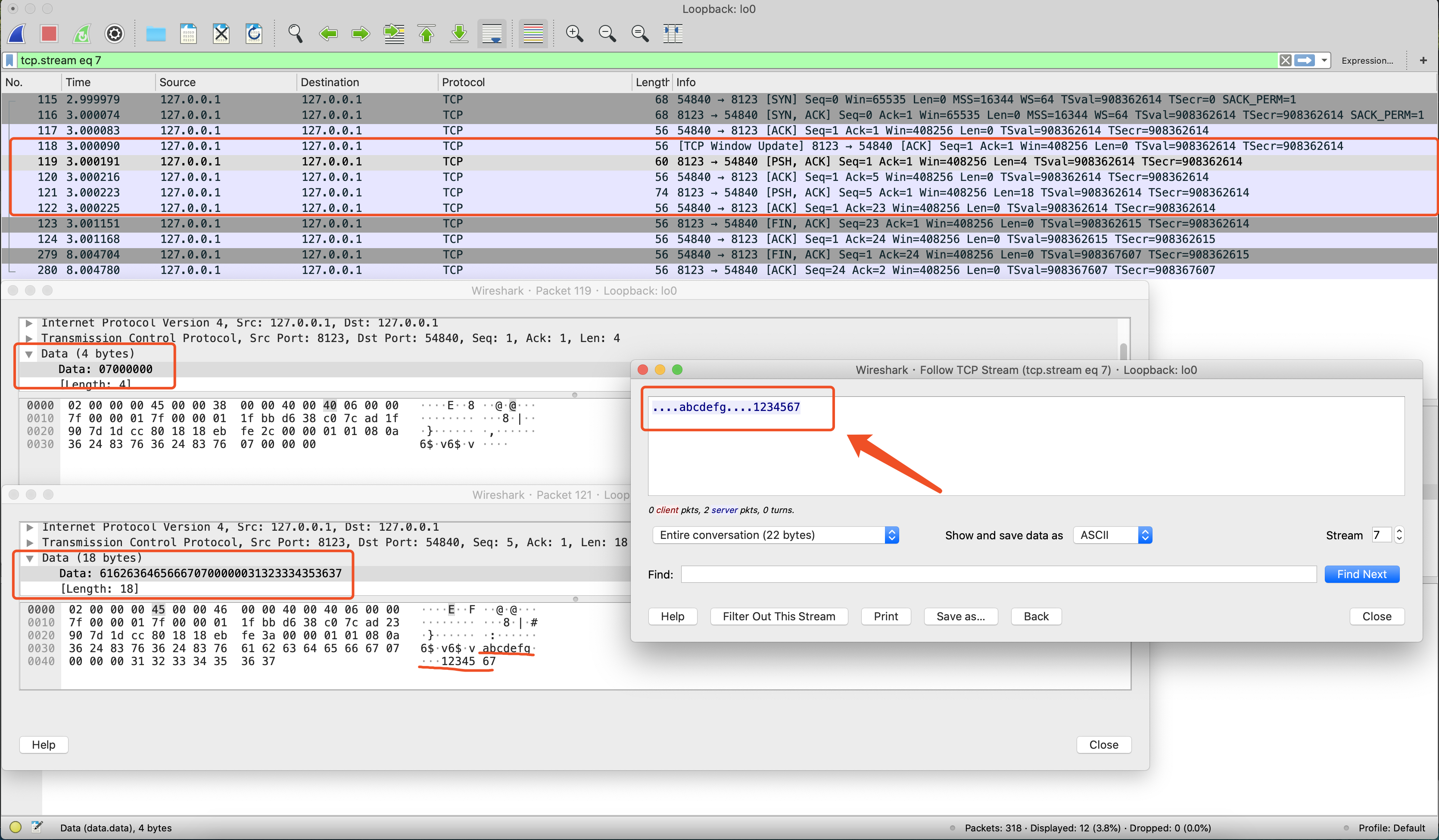
**print(msu.my\_recv(sk))**

**print(msu.my\_recv(sk))**

有了公共的函数体，每次发送和接收数据时只需要调用函数进行传参即可，终端输入输出效果同上。

### 3.4、证实粘包问题被解决

最后，我们启动上面通过struct模块，并调用函数实现发数据和接收数据的方式的程序，再来抓一次包，彻底证明上面的粘包问题得到了解决。



通过上图可以发现，在一次tcp通信中包含了正常的三次握手(前3个数据包)和四次挥手(后4个数据包)，第119和120号数据包是第一次双方数据通信，第121和122号数据包是第二次双方数据通信。两次数据包中分别包含了一个占4字节的数据和一个占18字节的数据。第一次的四字节大小的数据正式通过struct打包的表示长度的数据包，第二次的18字节组成依次是：7个字节的adcdefg数据、4个字节的表示长度的数据、7个字节的1234567数据。加起来正好18个字节，整个数据流依次是：长度数据、真实数据、长度数据、真实数据。

至此，一步步分析和解决python粘包问题的过程就完成啦✌️✌️✌️

本文中涉及到的代码文件以及抓取的数据包地址：  
https://github.com/Hargeek/python-nianbao-struct  
部分描述参考来源：  
https://www.cnblogs.com/Eva-J/articles/8244551.html

# 二、udp

## 1.udp chat

|  |  |
| --- | --- |
| #udp\_server.py  import socket from time import ctime  host = '' port = 8888 recv\_size = 1024 upd\_server = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM) upd\_server.bind((host,port)) while True:  data, addr = upd\_server.recvfrom(1024)  upd\_server.sendto(bytes('[%s]%s'%(ctime(),data.decode('utf-8')),'utf-8'),addr)  print("response to :",addr)  udp\_server.close() | #udp\_client.py  import socket  host = '127.0.0.1' port = 8888 recv\_size = 1024 addr = (host,port) upd\_client = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM) while True:  msg = input("请输入信息>>:")  if not msg:  break  upd\_client.sendto(bytes(msg,'utf-8'),addr)  data,addr = upd\_client.recvfrom(recv\_size)  if not data:  break  print(data.decode('utf-8')) upd\_client.close() |

## 2.udp远程命令调用

|  |  |
| --- | --- |
| # udp\_rpc\_server.py  import socket import subprocess  host = '' port = 8888 recv\_size = 1024 upd\_server = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM) upd\_server.bind((host, port)) while True:  data, addr = upd\_server.recvfrom(1024)  cmd = data.decode('utf-8')  print('收到客户端命令', cmd)  res = subprocess.Popen(cmd, shell=True,  stderr=subprocess.PIPE,  stdout=subprocess.PIPE,  stdin=subprocess.PIPE)  err = res.stderr.read()  if err:  res\_data = err  else:  res\_data = res.stdout.read()  if not res\_data:  res\_data = "command executed successfully.....".encode('gbk')  upd\_server.sendto(res\_data,addr)  print("response to :", addr)  udp\_server.close() | # udp\_rpc\_client.py  import socket  host = '127.0.0.1' port = 8888 recv\_size = 1024 addr = (host,port) upd\_client = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM) while True:  msg = input("请输入命令>>:")  if not msg:  continue  if 'exit' == msg:  break  upd\_client.sendto(bytes(msg,'utf-8'),addr)  data,addr = upd\_client.recvfrom(recv\_size)  if not data:  break  print(data.decode('gbk')) |

# 三、socketserver

**Tcp通信使用socket只能一对一通信，不能并发，也就是一次只能有一个客户端访问服务器，这是不能满足要求的，socketserver就可以实现并发**

## 1.socketserver基本通信：

|  |  |
| --- | --- |
| #socketserver\_server.py  import socketserver  """ socketserver可以实现并发 利用socketserver编程需要自己定义一个类继承socketserver.BaseRequestHandler  """   class TcpSocket\_server(socketserver.BaseRequestHandler):  def handle(self) -> None:  print("connection:", self.request)  # print("address:", self.client\_address)   # 通信  while True:  try:  data = self.request.recv(1024)  if not data:  break  # 判断是否是退出信号,是就退出  if b'exit' == data:  self.request.sendall(data)  self.request.close()  break  print("接收到客户端%s:%s信息：" % self.client\_address, data.decode('utf-8'))  # 发送信息  self.request.sendall(data)  except Exception as e:  print("Exception", e)  # self.server.shutdown()   if \_\_name\_\_ == '\_\_main\_\_':  s = socketserver.ThreadingTCPServer(('127.0.0.1', 8888), TcpSocket\_server)  s.serve\_forever() | #client.py  import socket  recv\_size = 1024 # 创建客户端 client = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) # 连接服务器 client.connect(('127.0.0.1', 8888)) while True:  # 发送信息  msg = input("请输入发送给服务器的信息:")  client.send(msg.encode("utf-8"))  # 接收服务器信息  data = client.recv(recv\_size)  if data.decode('utf-8') == 'exit':  break  print(data.decode('utf-8'))  print("done!") client.close() |

## 2.socketserver实现远程执行命令并且返回结果

|  |  |
| --- | --- |
| #rpc\_server.py  import socketserver import subprocess import struct  """ socketserver可以实现并发 利用socketserver编程需要自己定义一个类继承socketserver.BaseRequestHandler  """   class TcpSocket\_server(socketserver.BaseRequestHandler):   def handle(self) -> None:  print("connection:", self.request)  # print("address:", self.client\_address)   # 通信  while True:  try:  data\_cmd = self.request.recv(1024)  if not data\_cmd:  break  # 判断是否是退出信号,是就退出  if b'exit' == data\_cmd:  self.request.sendall(data\_cmd)  self.request.close()  break  print("接收到客户端%s:%s的命令" % self.client\_address, data\_cmd.decode('utf-8'))  result = subprocess.Popen(data\_cmd.decode('utf-8'), shell=True,  stderr=subprocess.PIPE,  stdout=subprocess.PIPE,  stdin=subprocess.PIPE)  err = result.stderr.read() # 通过res.stderr.read()读取错误消息  if err:  res\_data = err  else:  res\_data = result.stdout.read() # 通过res.stdout.read()读取标准输出消息  if not res\_data: # 有些系统命令是没有返回值的，需要处理一下  res\_data = '执行成功'.encode('gbk') # 防止操作系统执行命令后，返回值为空时，为客户端返回此消息,这里的编码是gbk因为cmd窗口就是gbk编码的  \_len = len(res\_data)  \_len\_str = struct.pack('i', \_len)  print("响应客户端的命令。。。。。")  self.request.sendall(\_len\_str)  self.request.sendall(res\_data)  except Exception as e:  print("Exception", e)  # self.server.shutdown()   if \_\_name\_\_ == '\_\_main\_\_':  s = socketserver.ThreadingTCPServer(('127.0.0.1', 8888), TcpSocket\_server)  s.serve\_forever() | #rpc\_client.py  import socket import struct  recv\_size = 1024 # 创建客户端 client = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) # 连接服务器 client.connect(('127.0.0.1', 8888)) while True:  # 发送信息  cmd = input("请输入命令:")  if not cmd:  continue # 如果输入为空，跳过此次发送  if 'exit' == cmd:  break # 输入exit断开连接  # 发送命令给服务器  client.send(cmd.encode("utf-8"))  # 接收服务器信息  len\_data = client.recv(4)  header\_data = struct.unpack('i', len\_data)[0]  # 获取调用返回数据  res\_len = 0  res\_msg = b''  while res\_len < header\_data:  res\_msg += client.recv(recv\_size)  res\_len = len(res\_msg)  print(res\_len)  print("执行结果：" + res\_msg.decode('gbk')) # cmd窗口返回的内容是gbk编码的 print("done!") client.close() |

## 3.socketserver模拟文件下载服务功能

|  |  |
| --- | --- |
| #file\_sockerserver.py  import socketserver import struct import os """ socketserver可以实现并发 利用socketserver编程需要自己定义一个类继承socketserver.BaseRequestHandler  """   class TcpSocket\_server(socketserver.BaseRequestHandler):  def handle(self) -> None:  print("connection:", self.request)  # print("address:", self.client\_address)   # 通信  while True:  try:  file\_name = self.request.recv(1024)  if not file\_name:  break  # 判断是否是退出信号,是就退出  if b'exit' == file\_name:  self.request.sendall(file\_name)  self.request.close()  break  print("filename:", file\_name.decode('utf-8'))  # 发送文件  file\_data = b''  if not os.path.exists(file\_name):  file\_data = ''  else:  with open(file\_name, 'rb') as f:  file\_data = f.read()  \_len = len(file\_data)  \_len\_str = struct.pack('i', \_len)  self.request.sendall(\_len\_str)  self.request.sendall(file\_data)  except Exception as e:  print("Exception", e)  # self.server.shutdown()   if \_\_name\_\_ == '\_\_main\_\_':  s = socketserver.ThreadingTCPServer(('127.0.0.1', 8888), TcpSocket\_server)  s.serve\_forever() | # client.py  import os import socket import struct  recv\_size = 1024 # 创建客户端 client = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) # 连接服务器 client.connect(('127.0.0.1', 8888)) while True:  # 发送信息  cmd = input("文件名:").strip() # strip()方法去除空格  if not cmd:  continue # 如果输入为空，跳过此次发送  if 'exit' == cmd:  break # 输入exit断开连接  # 发送命令给服务器  client.send(cmd.encode("utf-8"))  # 接收服务器信息  len\_data = client.recv(4)  header\_data = struct.unpack('i', len\_data)[0]  if header\_data == 0: # 返回空字符串上面文件不存在，跳过本次循环  print("文件不存在。。。")  continue  else:  # 获取调用返回数据  res\_len = 0  res\_msg = b''  with open(cmd,'wb') as f:  while res\_len < header\_data:  res\_msg = client.recv(recv\_size)  res\_len += len(res\_msg)  # print(res\_len)  f.write(res\_msg)  print('file saved...')  print("done!") client.close() |