1. 需求分析：
   * 1. LCX的使用场景描述：
2. 角色分析：

Local Server：相当于现实生活中的肉机；

Local Slave：和Remote建立隧道，间接地和Remote Client进行消息收发， 直接地和Local Server进行消息收发；

Remote Listen：和Local Server建立隧道，间接地和Local Server进行消息收 发，直接地和Remote Client进行消息收发；

Remote Client：相当于控制机，通过Remote Listen和Local Slave之间建立的 隧道，对Local Server进行消息收发；

1. 场景分析：
   * 1. Local Slave和Remote Listen之间的隧道建立：
        1. Remote Listen不断监听8000端口；
        2. Local Slave就位，尝试连接Remote Listen所属IP的8000端口
        3. Remote Listen监听到连接请求，并向请求方发送一个包含随机数的chap认证包；
        4. Local Slave接收到chap认证包，通过MD5算法对自己用户对应的密码和接受到的随机数进行加密，并将自己的用户名、密文继续向Remote Listen发送；
        5. Remote Listen接收到密文包，首先根据包中的用户名信息，在自己的信息字典里查询其对应的密码，并将改密码和之前发送的随机数进行MD5加密，然后，将自己加密的密文和接收到的密文进行对比，实现chap认证，最后，向Local Slave发送认证结果；
        6. Local Slave接收到认证结果，若结果显示认证成功，则chap认证结束，Local Slave和Remote Listen成功认证，且Local Slave向Remote Listen发送绑定请求包，其中包含着，Local Slave想要在Remote Listen绑定的监听端口号；否则认证失败，Local Server可以选择重新认证；
        7. 若Remote Listen接收到绑定请求包，其会对包中请求绑定的端口号进行监听，并向Local Slave发送绑定结果信息；
        8. Local Slave接收到绑定结果信息，若绑定成功，则Local Slave和Remote Listen之间成功建立隧道；否则，绑定失败，Local Slave可以选择重新发送绑定请求；
     2. Remote Client尝试连接：
        1. Remote Client向8001（Remote Listen绑定的监听端口）发送连接请求；
        2. Remote Listen接收到连接请求包，开始向Local Slave发起连接请求包，其中包含请求连接的Remote Client的标识；
        3. Local Slave接收到连接请求包，记录请求连接的Remote Client标识，并向Local Server发起连接请求；
        4. 当成功连接到Local Server后，Local Slave将Remote Client和Local Server的标识绑定，并向Remote Listen发送包含Local Server标识的连接响应包；
        5. Remote Listen接收到Local Slave发来的连接响应包，并将其中的Local Server标识和请求连接的Remote Client标识进行绑定，此时，Local Server和Remote Client之间成功建立了信息通路；
     3. 双向数据传递：
        1. Remote Client（已经建立双向连接）向Remote Listen发送数据信息；
        2. Remote Listen接收到Remote Client发来的数据信息后，将该数据信息和Remote Client的标识打包发送Local Slave；
        3. Local Slave接收到Remote Listen发送来的数据信息包，根据其中的Remote Client标识，找到对应的Local Server，并向其发送数据包中的数据；
        4. 对应的Local Server成功接收到Remote Client发送来的数据；
        5. Local Server（已经建立双向连接）向Local Slave发送数据信息；
        6. Local Slave接收到Local Server发来的数据信息后，将该数据信息和Local Server的标识打包发送到Remote Listen；
        7. Remote Listen接收到Local Slave发送的数据信息包，根据其中的Local Server标识，找到对应的Remote Client，并向其发送数据包中的数据；
        8. 对应的Remote Client成功接收到Local Server发来的数据；
     4. 一方TCP关闭连接：
        1. Remote Client关闭TCP连接；
        2. Remote Listen检测到Remote Client关闭了TCP连接，其将Remote Client的标识打包，向Local Slave发送断开一条连接的消息；
        3. Local Slave收到连接断开消息，其根据包中的Remote Client标识找到对应的Local Server，断开和其建立的连接；
        4. Local Server关闭TCP连接；
        5. Local Slave检测到Local Server关闭了TCP连接，其将Local Server的标识打包，向Remote Listen发送断开一条连接的消息；
        6. Remote Listen接收到连接断开消息，其根据包中的Local Server标识找到对应的Remote Client，断开和其建立的连接；
2. 方案设计：
   * 1. asyncio中的事件循环event loop和协程coroutine：

event loop是程序开启的一个无限循环，该事件循环可以通过：loop=asyncio.get\_event\_loop()进行创建，event loop对象包括两部分：event 和loop，其中，event负责I/O时间通知，而loop负责循环处理I/O通知并在就绪时调用回调。event loop内部维护着两个容易：\_ready和\_scheduled，\_ready表示可以执行，\_scheduled表示计划执行。然后，可以通过：loop.run\_until\_complete()将一些函数注册到该事件循环上并且该事件循环会执行到被注册的事件结束，此外，还可以通过：loop.create\_task()注册事件，如果通过：loop.run\_forever（）来运行事件循环的话，则只有在所有的事件均被完成（回调）后，该循环才结束，或者使用loop.stop（）结束循环，被注册的函数会被包装成future，future的初始状态为pending，当调用cancel（）方法时会立即进入cancelled状态并schedule callbacks，当被调用set\_result（）时，会进入finished状态，并schedule callbacks，二种情况传入callback的参数是不同的；

coroutine是一个协程对象，通过async关键字可以将函数封装成一个协程对象，调用这个协程对象并不会启动它的代码，返回的其实是一个生成器对象。在函数代码中需要在阻塞操作前加上关键字await，这样，程序运行到这里就不会阻塞，而是继续运行，直到阻塞操作完成时，才从之前位置继续执行；

coroutine对象需要被注册到event loop中，事件循环执行过程中，如果遇到协程对象阻塞，那么当前协程会被挂起，事件循环会继续执行，当阻塞操作完成后，事件循环会暂停并将当前的上下文和寄存器信息保存在堆栈中，然后执行协程阻塞位置之后的代码，当I/O运行结束后，切换回之前的状态，这样就可以实现I/O的异步操作。

例如：

import asyncio

@asyncio.coroutine

def compute(x, y):

print("Compute %s + %s ..." % (x, y))

yield from asyncio.sleep(1.0)

return x + y

@asyncio.coroutine

def print\_sum(x, y):

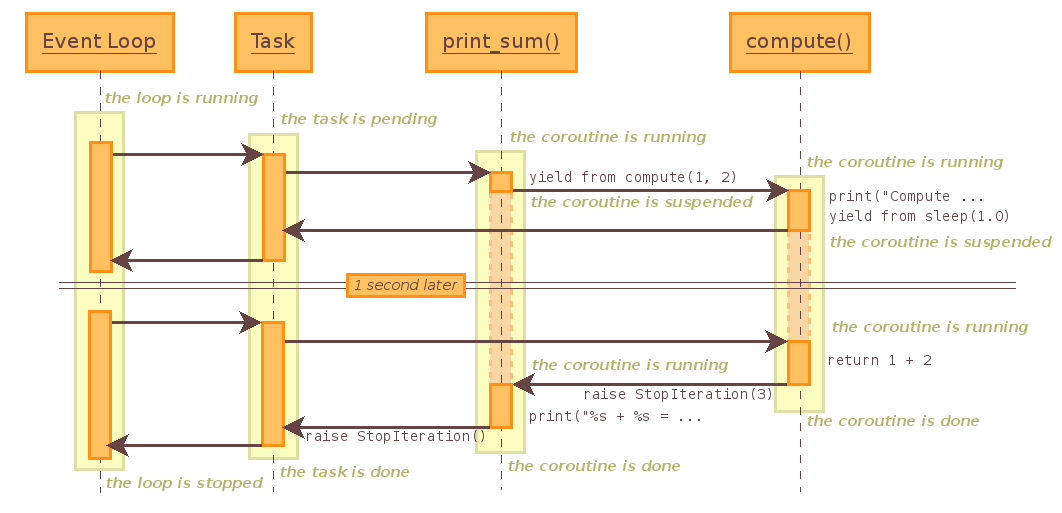
result = yield from compute(x, y)

print("%s + %s = %s" % (x, y, result))

loop = asyncio.get\_event\_loop()

loop.run\_until\_complete(print\_sum(1, 2))

loop.close()



例子摘自：https://docs.python.org/3.4/library/asyncio-task.html#coroutines

* + 1. callback based API：

1. 分析：

这种模式需要把协议封装成一个类，并将该协议类的实例添加进event loop。对于client来说每创建一个连接，都需要创建一个EchoClientProtocol实例，该实例中包含收发数据包以及断开连接等成员函数；对于server来说，每当有一个client对其发起连接，都需要创建一个EchoServerClientProtocol实例，其中包含接收数据包和处理连接等成员函数；

1. 优点：

当一次调用进行的异步操作较少时，由于不需要创建完整的堆栈，因此性能会更好；

1. 缺点：

当一次调用的异步操作很多时，由于闭包回调的每一次调用都必须创建闭包函数，因此开销会比较大，性能会下降；

* + 1. coroutine based API：

1. 分析：

和基于回调的API不同，这种模式不需要创建协议类。对于client来说，需要一个tcp\_echo\_client函数，并将其扔到event loop中，用来实现向服务器发起连接和收发数据包等功能；对于server来说，需要一个handle\_echo函数，并将其扔到event loop中，用来实现处理client的连接以及收发数据包等功能。但是，这种模式需要为协程创建一个完全的堆栈。

1. 优点：

当一次调用中包含大量异步操作时，由于只需要创建一次协程，只需要一个完全的堆栈，因此性能会比较好；

1. 缺点：

当一次调用中的异步操作比较少时，由于需要为协程创建一个完全的堆栈，因此开销会比较大，性能会下降；

* + 1. 我的LCX项目采用coroutine based API：

理由：无论是Remote Listen端还是Local Slave端，每次调用都需要进行大量的异步操作（收发数据包等），因此，使用该模式只需要创建一次协程，并为其创建完全堆栈，性能会比较好；

1. 代码实现：
   * 1. Remote Listen：

**import** asyncio  
**import** sys  
**import** random,os  
**import** hashlib  
**import** struct  
HOSTLEN=40  
TIMEOUT=300  
MAXSIZE=20480  
CONNECTNUM=5  
header\_len = 6  
client=[]  
mark=0  
flag=0  
markclient=0  
**async def** receive\_packet(reader):  
 header = **await** reader.read(header\_len)  
 **if** header == **''**:  
 **raise** RuntimeError(**"socket connection broken"**)  
 (message\_len,code,data\_len) = struct.unpack(**'!HHH'**, header)  
 print(message\_len,code,data\_len)  
 packet = header  
  
 **while** len(packet) < message\_len:  
 chunk = **await** reader.read(message\_len - len(packet))  
 **if** chunk == **''**:  
 **raise** RuntimeError(**"socket connection broken"**)  
 packet = packet + chunk  
  
 (message\_len,code,data\_len,data) = struct.unpack(**'!HHH'** + str(data\_len) + **'s'**, packet)  
 **return** {**'message\_len'**:message\_len,  
 **'code'**: code,  
 **'data\_len'**: data\_len,  
 **'data'**: data}  
**async def** transmit1(reader,writer):  
 **global** markclient  
 markclient=markclient+1  
 mclient=markclient  
 recvierData = **b""  
 global** flag  
 **global** mark  
 mid=[reader,writer]  
 client.append(mid)  
 CoNid = 0  
 print(**'Accept Connection'**)  
 code = 7  
 data\_len = 4  
 pack\_format = **'!HHHHH'** message\_len = header\_len + data\_len  
 packet = struct.pack(pack\_format, message\_len, code, data\_len, request\_id, listen\_port)  
 print(**"Send a co nnect request!"**)  
 remote[1].write(packet)  
 **while** 1:  
 **try**:  
 data = **await** reader.read(20480)  
 **await** asyncio.sleep(1)  
 print(**'有数据从client发过来了'**)  
 print(data)  
 print(mark)  
 recvierData += data *# 要给8000发送的数据* **for** k, v **in** conn\_sock.items():  
 **if** v == [reader,writer]:  
 CoNid = k  
 print(CoNid)  
 print(**'查到了！！！'**)  
 **break  
 if** len(data)==0:  
 print(**'One of the clients are break!'**)  
 code = 10  
 data\_len = 2  
 connectid = CoNid  
 message\_len = header\_len + data\_len  
 pack\_format = **'!HHHH'** packet = struct.pack(pack\_format, message\_len, code, data\_len, connectid)  
 remote[1].write(packet)  
 print(**'已发送disconnect '**)  
 **for** k, v **in** conn\_sock.items():  
 **if** v == [reader,writer]:  
 conn\_sock.pop(k)  
 **break** writer.close()  
 **break  
 if** len(recvierData) != 0:  
 bytes = len(recvierData)  
 pack\_format = **'!HHHH'** + str(bytes) + **'s'** code = 9  
 data\_len = bytes + 2  
 message\_len = data\_len + header\_len  
 connect\_id=CoNid  
 print(**'data connectid is %d'**%(connect\_id))  
 packet = struct.pack(pack\_format, message\_len, code, data\_len, connect\_id, recvierData)  
 remote[1].write(packet)  
 print(**'数据已成功发送！'**)  
 recvierData = **b""  
 except**:  
 **break  
async def** transmit2():  
 senderData = **b""** Connectid = 0  
 **global** mark  
 **while** 1:  
 packet = **await** receive\_packet(remote[0])  
 print(**'New coming is :%d'** % (packet[**'code'**]))  
 **if** packet[**'code'**] == 8:  
 data = packet[**'data'**]  
 (Requestid, Result, Connect\_id) = struct.unpack(**'!HHH'**, data)  
 print(**'connect id is %d'** % (Connect\_id))  
 print(**"Get a request ack"**)  
 mid = {Connect\_id: client[mark]}  
 mark+=1  
 print(mid)  
 conn\_sock.update(mid)  
 mid.clear()  
 **elif** packet[**'code'**] == 10:  
 print(**'TCP connect was broken!'**)  
 data = packet[**'data'**]  
 (connectID,) = struct.unpack(**'!H'**, data)  
 conn\_sock[connectID][1].close()  
 conn\_sock.pop(connectID)  
 packet.clear()  
 **elif** packet[**'code'**] == 9:  
 data = packet[**'data'**]  
 data\_len = packet[**'data\_len'**]  
 (Connectid, Data) = struct.unpack(**'!H'** + str(data\_len - 2) + **'s'**, data)  
 print(Data)  
 print(**'connect id is %d'** % (Connectid))  
 senderData += Data *# 准备发给client的数据* **if** len(senderData) != 0:  
 **if** Connectid **in** conn\_sock.keys():  
 conn\_sock.get(Connectid)[1].write(senderData)  
 **else**:  
 print(**'The connectID is not exist!'**)  
 senderData = **b""  
  
async def** listener(reader,writer):  
 remote.clear()  
 remote.append(reader)  
 remote.append(writer)  
 salt = str(os.urandom(8)).encode()  
 code = 1  
 salt\_len = len(salt)  
 message\_len = header\_len + salt\_len  
 pack\_format = **'!HHH'** + str(salt\_len) + **'s'** packet = struct.pack(pack\_format, message\_len, code, salt\_len, salt)  
 writer.write(packet)  
 **await** writer.drain()  
 **global** request\_id  
 request\_id= 0  
 **global** listen\_port  
 listen\_port= 0  
 connect\_id = 0  
 **global** senderData  
 **global** Connectid  
 **while** 1:  
 packet=**await** receive\_packet(reader)  
 print(**'New coming is %d'**%(packet[**'code'**]))  
 **if** packet[**'code'**]==2:  
 print(**'Hashcode is back!'**)  
 data=packet[**'data'**]  
 (hash\_len,user\_len,hashuser)=struct.unpack(**'!HH'**+str(packet[**'data\_len'**]-4)+**'s'**,data)  
 (hashcode,username)=struct.unpack(**'!'**+str(hash\_len)+**'s'**+str(user\_len)+**'s'**,hashuser)  
 password=str(user.get(username.decode()))  
 secrect=password+str(salt)  
 my\_hash=hashlib.md5(secrect.encode()).hexdigest()  
 **if** hashcode.decode()==my\_hash:  
 print(**"Comfirm succeed!"**)  
 code=3  
 result=1  
 **else**:  
 print(**"Confirm failed!"**)  
 code=4  
 result=0  
 data\_len=2  
 pack\_format=**'!HHHH'** message\_len=2+header\_len  
 packet=struct.pack(pack\_format,message\_len,code,data\_len,result)  
 writer.write(packet)  
 **await** writer.drain()  
 **elif** packet[**'code'**]==5:  
 data=packet[**'data'**]  
 (request\_id,result,listen\_port)=struct.unpack(**'!HHH'**,data)  
 **if** listen\_port==0:  
 port = random.randrange(1, 9999, 1)  
 listen\_port = int(port)  
 code=6  
 **try**:  
 result = 1  
 **except**:  
 result=0  
 data\_len=6  
 pack\_format=**'!HHHHHH'** message\_len=header\_len+data\_len  
 packet=struct.pack(pack\_format,message\_len,code,data\_len,request\_id,result,listen\_port)  
 writer.write(packet)  
 **await** writer.drain()  
 loop.create\_task(transmit2())  
 loop.create\_task(asyncio.start\_server(transmit1, **'0.0.0.0'**, listen\_port, loop=loop))*#监听8001* **break  
def** main():  
 RemotePort=0  
 **global** user  
 **global** remote  
 **global** conn\_sock  
 conn\_sock={}  
 **global** task\_list  
 task\_list={}  
 remote=[]  
 user={}  
 success=0  
 **global** loop  
 **global** verbose  
 **if** len(sys.argv)>2:  
 **if** sys.argv[1]==**"-p" and** len(sys.argv)>=3:  
 RemotePort=int(sys.argv[2])  
 **else**:  
 print(**"instruction error1!"**)  
 **if** sys.argv[3]==**"-u" and** len(sys.argv)>=5:  
 string=str(sys.argv[4]).split(**','**)  
 **for** s **in** string:  
 us=s.split(**':'**)  
 dict1={us[0]:us[1]}  
 user.update(dict1)  
 success=1  
 **else**:  
 print(**"instruction error2!"**)  
 **else**:  
 print(**"instruction error3!"**)  
 **if** success==1:  
 loop = asyncio.get\_event\_loop()  
 coro = asyncio.start\_server(listener, **'0.0.0.0'**, RemotePort, loop=loop)  
 server = loop.run\_until\_complete(coro)  
 print(**'Serving on{}'**.format(server.sockets[0].getsockname()))  
 **try**:  
 loop.run\_forever()  
 **except** KeyboardInterrupt:  
 **pass** server.close()  
 loop.run\_until\_complete(server.wait\_closed())  
 loop.close()  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 main()

* + 1. Local Slave：

**import** asyncio  
**import** sys  
**import** random  
**import** hashlib  
**import** struct  
HOSTLEN=40  
TIMEOUT=300  
MAXSIZE=20480  
CONNECTNUM=5  
header\_len = 6  
client=[]  
address1=**""**cid=1  
**async def** receive\_packet(reader):  
 header = **await** reader.read(header\_len)  
 **if** header == **''**:  
 **raise** RuntimeError(**"socket connection broken"**)  
 (message\_len,code,data\_len) = struct.unpack(**'!HHH'**, header)  
 print(message\_len,code,data\_len)  
 packet = header  
  
 **while** len(packet) < message\_len:  
 chunk = **await** reader.read(message\_len - len(packet))  
 **if** chunk == **''**:  
 **raise** RuntimeError(**"socket connection broken"**)  
 packet = packet + chunk  
  
 (message\_len,code,data\_len,data) = struct.unpack(**'!HHH'** + str(data\_len) + **'s'**, packet)  
 **return** {**'message\_len'**:message\_len,  
 **'code'**: code,  
 **'data\_len'**: data\_len,  
 **'data'**: data}  
**async def** transmit(message1,loop):  
 senderData = **b""  
 global** Connectid  
 **global** address1  
 **global** cid  
 Connectid = 0  
 **while** 1:  
 packet = **await** receive\_packet(message1[0])  
 print(**'New coming is %d'** % (packet[**'code'**]))  
 **if** packet[**'code'**] == 7:  
 data = packet[**'data'**]  
 (Requestid, Listenport) = struct.unpack(**'!HH'**, data)  
 print(**"The listenport is %d"** % (Listenport))  
 print(**"Start to connect slavePC %s:%d"** % (local))  
 reader, writer = **await** asyncio.open\_connection(local[0],local[1] , loop=loop)  
 midd=[reader,writer]  
 client.append(midd)  
 loop.create\_task(transmit2(message1,client[len(client)-1][0],client[len(client)-1][1]))  
 print(**"Connect slave succeed!"**)  
 code = 8  
 result = 1  
 ConnectID = cid  
 cid+=1  
 print(**'connect id is %d'**%(ConnectID))  
 slave=[reader, writer]  
 mid = {ConnectID: slave}  
 conn\_sock.update(mid)  
 data\_len = 6  
 message\_len = data\_len + header\_len  
 pack\_format = **'!HHHHHH'** packet = struct.pack(pack\_format, message\_len, code, data\_len, Requestid, result, ConnectID)  
 print(**"Send connect ack!"**)  
 message1[1].write(packet)  
 print(**"Connect ack send succeed!"**)  
 **elif** packet[**'code'**] == 10:  
 print(**'TCP connect was broken!'**)  
 data = packet[**'data'**]  
 (connect\_id,) = struct.unpack(**'!H'**, data)  
 print(**'Close connect\_id is :%d'** % connect\_id)  
 conn\_sock[connect\_id][1].close()  
 conn\_sock.pop(connect\_id)  
 **elif** packet[**'code'**] == 9:  
 data = packet[**'data'**]  
 data\_len = packet[**'data\_len'**]  
 (Connectid, Data) = struct.unpack(**'!H'** + str(data\_len - 2) + **'s'**, data)  
 print(Data)  
 print(**'the coming id is:%d'**%Connectid)  
 senderData += Data *# 准备发给client的数据* address1 = address  
 **if** len(senderData) != 0:  
 **if** Connectid **in** conn\_sock.keys():  
 conn\_sock.get(Connectid)[1].write(senderData)  
 **else**:  
 print(**'The connectID is not exist!'**)  
 senderData = **b""  
  
  
async def** transmit2(message1,reader,writer):  
 recvierData = **b""** verbose = **False  
 if '-v' in** sys.argv:  
 verbose = **True** connect\_id = 0  
 slave=[reader,writer]  
 **while** 1:  
 **try**:  
 data = **b""** data = **await** reader.read(20480)  
 recvierData += data *# 要给8000发送的数据* address1 = local  
 **for** k, v **in** conn\_sock.items():  
 **if** v == slave:  
 print(**'查到了！'**)  
 connect\_id = k  
 **break  
 if** len(data)==0:  
 print(**'One of the local\_slave are break!'**)  
 code = 10  
 data\_len = 2  
 connectid = 0  
 **for** k, v **in** conn\_sock.items():  
 **if** v == slave:  
 connectid = k  
 **break** conn\_sock[connectid][1].close()  
 print(**'The connect\_id is:%d'** % (connectid))  
 message\_len = header\_len + data\_len  
 pack\_format = **'!HHHH'** packet = struct.pack(pack\_format, message\_len, code, data\_len, connectid)  
 message1[1].write(packet)  
 print(**'已发送disconnect'**)  
 **break** bytes = len(data)  
 **if** verbose:  
 print(**"Recv From %s:%d"** % address1, **" %d bytes"** % bytes)  
 **if** len(recvierData) != 0:  
 bytes = len(recvierData)  
 **if** verbose:  
 print(**"Send to %s:%d"**, address, **" %d bytes"** % bytes)  
 pack\_format = **'!HHHH'** + str(bytes) + **'s'** code = 9  
 data\_len = bytes + 2  
 message\_len = data\_len + header\_len  
 packet = struct.pack(pack\_format, message\_len, code, data\_len, connect\_id, recvierData)  
 message1[1].write(packet) *# 向slave发送Data数据包* recvierData = **b""  
 except**:  
 **break  
  
  
  
  
async def** slaver(message,loop):  
 reader, writer = **await** asyncio.open\_connection(message[0], message[1], loop=loop)  
 request\_id=0  
 connect\_id=0  
 listen\_port=0  
 **while True**:  
 packet=**await** receive\_packet(reader)  
 print(**"New coming is:%d"**%(packet[**'code'**]))  
 **if** packet[**'code'**]==1:  
 print(**"Got a salt!"**)  
 salt=str(packet[**'data'**])  
 code=2  
 secrect=password+salt  
 hashcode=hashlib.md5(secrect.encode()).hexdigest()  
 hash\_len=len(hashcode.encode())  
 user\_len=len(user.encode())  
 data\_len=hash\_len+user\_len+4  
 message\_len=data\_len+header\_len  
 pack\_format=**'!HHHHH'**+str(hash\_len)+**'s'**+str(user\_len)+**'s'** packet=struct.pack(pack\_format,message\_len,code,data\_len,hash\_len,user\_len,hashcode.encode(),user.encode())  
 print(**"Send chap result!"**)  
 writer.write(packet)  
 **elif** packet[**'code'**]==3:  
 print(**"Confirm succeed!"**)  
 code=5  
 request\_id=random.randint(0,100)  
 result=1  
 listen\_port=int(Remotelisten)  
 data\_len=6  
 message\_len=data\_len+header\_len  
 pack\_format = **'!HHHHHH'** packet = struct.pack(pack\_format, message\_len, code, data\_len, request\_id,result,listen\_port)  
 writer.write(packet)  
 **elif** packet[**'code'**]==4:  
 print(**"Confirm failed!"**)  
 **break**;  
 **elif** packet[**'code'**]==6:  
 data=packet[**'data'**]  
 (Requestid,Result,Listenport)=struct.unpack(**'!HHH'**,data)  
 **if** Result==1:  
 print(**'Bind response is back!'**)  
 print(**'ListenPort is %d'**%(Listenport))  
 **else**:  
 print(**'Deploy listenport Failed!'**)  
 message1=[reader,writer]  
 **await** transmit(message1,loop)  
 **break  
def** main():  
 **global** verbose  
 **global** conn\_sock  
 conn\_sock = {}  
  
 **global** loop,address,Remotelisten,local,user,password  
 **if** len(sys.argv)>2:  
 **if** sys.argv[1] == **"-r" and** len(sys.argv)>=3:  
 s=str(sys.argv[2]).split(**':'**)  
 RemoteAddr=s[0]  
 RemotePort=int(s[1])  
 **else**:  
 print(**"instruction error!"**)  
 **return  
 if** sys.argv[3]==**"-u" and** len(sys.argv)>=5:  
 s=str(sys.argv[4]).split(**':'**)  
 user=s[0]  
 password=s[1]  
 **else**:  
 print(**"instruction error!"**)  
 **return  
 if** sys.argv[5]==**"-p" and** len(sys.argv)>=7:  
 Remotelisten=sys.argv[6]  
 **else**:  
 print(**"instruction error!"**)  
 **return  
 if** sys.argv[7]==**"-l" and** len(sys.argv)>=9:  
 s=str(sys.argv[8]).split(**':'**)  
 LocalAddr=s[0]  
 LocalPort=int(s[1])  
 success=1  
 **else**:  
 print(**"instruction error!"**)  
 **return  
 else**:  
 print(**"instruction error!"**)  
 **return  
 if** success==1:  
 local=(LocalAddr,int(LocalPort))  
 address=((RemoteAddr,int(RemotePort)))  
 print(**'Connect to %s:%d'** % (address))  
 message=[RemoteAddr,int(RemotePort)]  
 loop = asyncio.get\_event\_loop()  
 loop.run\_until\_complete(slaver(message,loop))  
 loop.close()  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 main()

1. 测试方案：
   * 1. 方案描述：

通过批处理文件先执行Remote Listen，再执行Local Slave，当dos窗口显示认证成功，隧道已经建立的标志后，执行老师给的b4lcxt.py，当出现“all test over”后，检查过程是否正确，如果每一步都正确，即每个Remote Client都接收到“1234567”，且每个Local Server都接收到“ABCDEFG”，则说明测试成功。

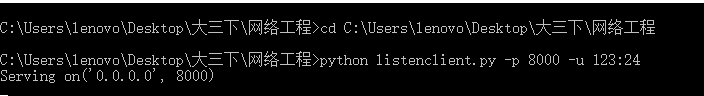
* + 1. 测试过程：

1. 先通过批处理文件执行Remote Listen部分代码：

批处理文件：

网络工程

批处理内容：



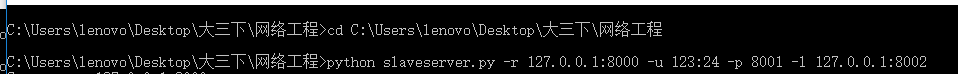
（其中，8000表示Remote Listen监听的端口，123表示Remote Listen本地保存的用户名，24为其对应的密码）

1. 再通过批处理文件执行Local Server部分代码：

批处理文件：

网络工程

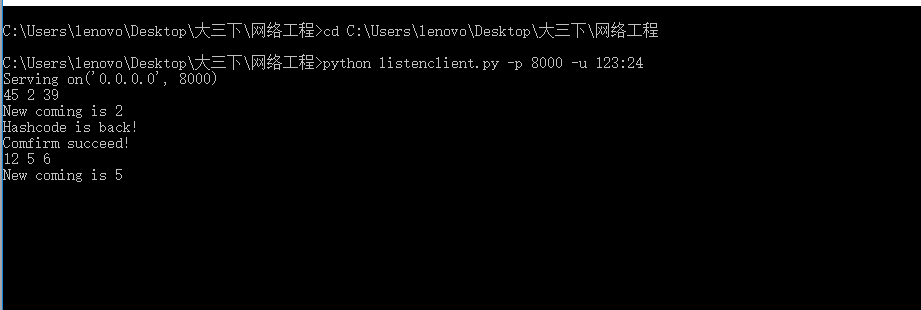
批处理内容：



（其中，127.0.0.1:8000表示Local Slave连接的IP地址以及端口号；123表示Local Slave对应的用户名，24表示其对应的密码；8001表示Local Slave想要Remote Listen绑定的监听端口号；127.0.0.1:8002表示Local Slave所要连接的Local Server对应的IP地址以及端口号）

1. chap验证以及Remote Listen和Local Client之间的隧道建立：

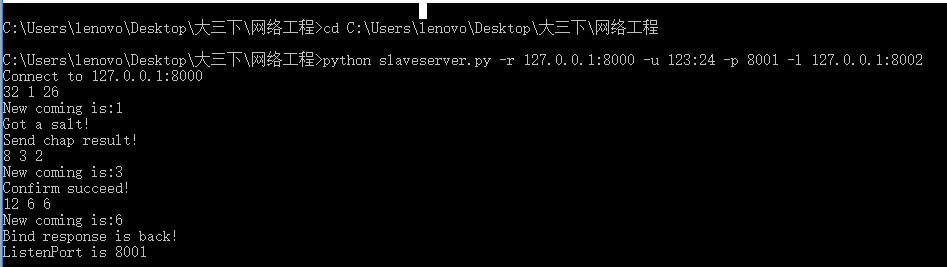
Remote Listen端：



其中：

1. Serving on（‘0.0.0.0’,8000）表示在本地的8000端口监听来自Local Slave的连接请求：
2. Hashcode is back表示随机数发送成功，且收到了Local Slave返回的 hash码；
3. 开始通过hash码对Local Slave进行chap认证，Comfirm succeed！表 示认证成功；
4. New coming is 5表示接收到来自Local Slave的code为5的数据包， 该数据包里含有请求绑定监听的端口号，Remote Listen开始对其进行监 听；

Local Slave端：



其中：

1. Connect to 127.0.0.1:8000表示开始连接Remote Listen；
2. Got a salt！表示连接Remote Listen成功，并接收到其发送来的随机数；
3. Send chap result！表示Local Slave成功将自己的用户名以及密码和随 机数的MD5密文发送给了Remote Listen；
4. Comfirm succeed！表示Local Slave的身份在Remote Listen那里认证 成功，Local Slave开始向Remote Listen发送监听绑定请求；
5. Bind response is back表示Remote Listen进行了对Local Slave的绑 定监听端口请求的回应；
6. ListenPort is 8001表示Remote Listen成功的对8001端口进行了监听， 至此，隧道建立成功；
7. 最后执行测试用例b4lcxt.py：

命令行1：python b4lcxt.py -b 127.0.0.1:8001 -l 8002 -s c -t 10

命令行2：python b4lcxt.py -b 127.0.0.1:8001 -l 8002 -s s -t 10

解释：

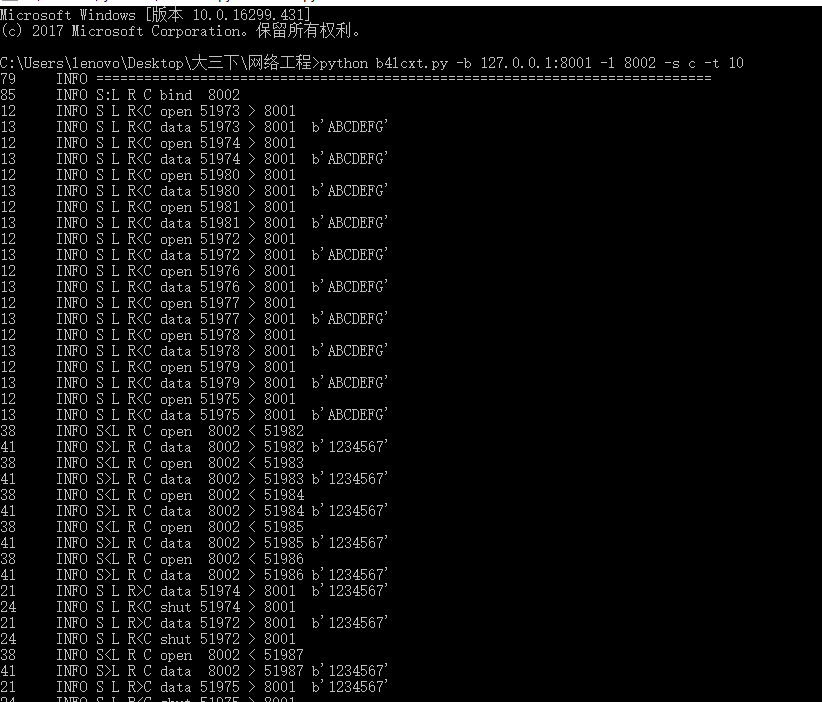
-b后表示Remote Client连接的IP地址和端口号；

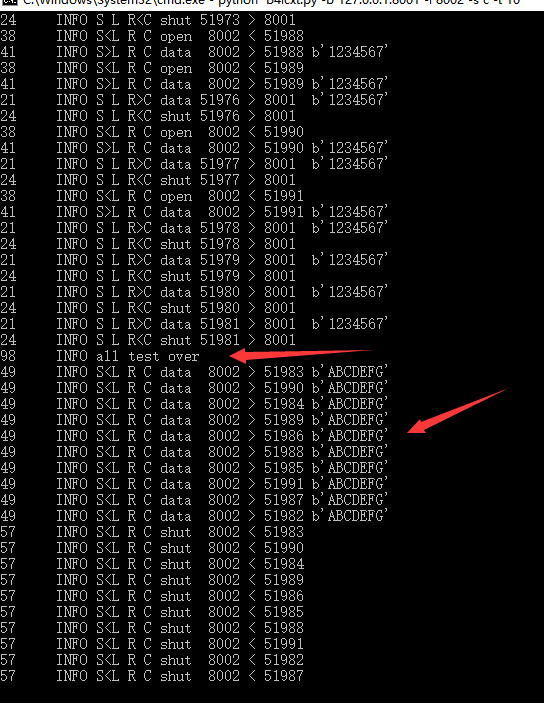
-l后表示Local Slave连接的端口号；

-s后表示关闭模式（c：关闭Remote Client；s：关闭Local Server）；

-t后表示同时进行连接的Remote Client个数；

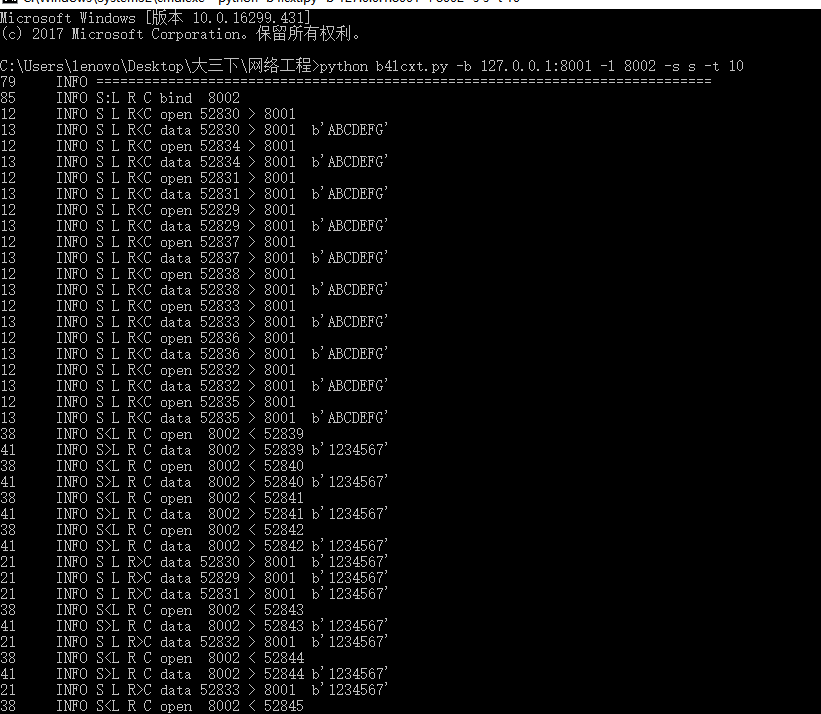
执行命令行1：

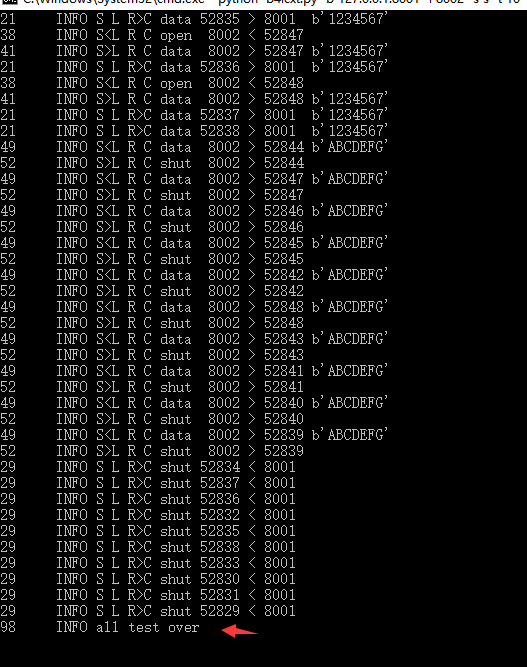




结果：成功出现“all test over”并且所有的Remote Client都接收到了来自 Local Server的“1234567”，并且所有的Local Server也都接收到了来自Remote Client的“ABCDEFG”；

执行命令行2（紧接着上一步，不用关掉Remote Listen和Local Slave）：





结果：成功出现“all test over”并且所有的Remote Client都接收到了来自 Local Server的“1234567”，并且所有的Local Server也都接收到了来自Remote Client的“ABCDEFG”；

综上所述：测试成功！

1. 总结：
   * 1. 需求分析阶段：

这次需求分析的关键在于准确对角色的交互过程的理解。在这一过程中，我加深了对连接建立以及chap验证的理解：验证过程是不可以传递明文的，因此需要对用户名和密码进行加解密，此外，为了确保验证消息的一对一传递，还需要服务器发送salt作为握手的凭据；

* + 1. 方案设计阶段：

该阶段的关键在于对python3中的asyncio库的准确理解与应用。在这一过程中，我首先仔细读了一遍有关asyncio的官方文档，然后又查询了一些博客，但是依然没有对asyncio完全理解。所以我根据coroutine based API里给出的TCP编程的例子，编写了几个测试代码，通过实践逐步完善对asyncio库的理解；

* + 1. 代码实现阶段：

该阶段的关键在于对数据包的处理以及asyncio的实现。数据包封装过程中，我首先在Word上把所有的数据包格式确定了下来，这为之后用“struct”打包解包做了不错的铺垫；对于异步I/O的实现，我以coroutine based API里给出的TCP编程的例子做基础并在GitHub上参考了用select实现的类似版本，然后逐步扩展，最终成型；

* + 1. 代码测试阶段：

该阶段的关键在于对b4lcxt.py的理解。我最初测试时，发现connectID和socket的关联总是建立失败，在仔细观察了老师给的用例后发现，多个Remote Client建立连接后，马上就发送了data，但是我的Remote Listen这个时候还在向Local Slave请求connectID，因此，接受到的数据会无法发送到对应的Local Server，也就是说我的代码存在结构问题，解决该问题后，测试就正常了。