

网络编程之socket

网络编程之socket

网络编程之socket

看到本篇文章的题目是不是很疑惑，what is this?，不要着急，但是记住一说网络编程，你就想socket，socket是实现网络编程的工具，那么什么是socket，什么是网络编程，什么是网络，为什么要学习socket，都在下面有讲解，大家细细看来！

本节目录

- [一 为什么要学习socket](#)
- [二 客户端\服务端架构](#)
- [三 网络通信的整个流程](#)
- [四 网络通信协议\(互联网协议\)](#)
- [五 osi七层模型](#)
- [六 socket](#)
- [七 套接字socket的发展史及分类](#)
- [八 基于TCP和UDP两个协议下socket的通讯流程](#)
- [九 粘包现象](#)
- [十 粘包的解决方案](#)
- [十一 验证客户端的连接合法性](#)
- [十二 socketserver模块实现开发](#)
- [十三 网络编程的作业](#)
-

一 为什么要学习socket

首先我们python基础部分已经学完了，而socket是我们基础进阶的课程，也就是说，你自己现在完全可以写一些小程序了，但是前面的学习和练习，我们写的代码都是在自己的电脑上运行的，虽然我们学过了模块引入，文件引入import等等，我可以在程序中获取到另一个文件的内容，对吧，但是那么突然有一天，你的朋友和你说：“把你电脑上的一个文件通过你自己写的程序发送到我的电脑上”，这时候怎么办？你是不是会想，what？这怎么搞？就在此时，突然灵感来了，我可以通过qq、云盘、微信等发送给他啊，可是人家说了，让你用自己写的程序啊，嗯，这是个问题，此时又来一个灵感，我给他发送文件肯定是通过网络啊，这就产生了网络，对吧，那我怎么让我的程序能够通过网络来联系到我的朋友呢，并且把文件发送给他呢，那么查了一下，发现网络通信通过socket可以搞，但是怎么搞呢？首先，查询结果是对的，socket就是网络通信的工具，任何一门语言都有socket，他不是任何一个语言的专有名词，而是大家通过自己的程序与其他电脑进行网络通信的时候都用它。知道为什么要学习socket了吧~~朋友们~~而你使用自己的电脑和别人的电脑进行联系并发送消息或者文件等操作就叫做网络通信。

对于一个小白来讲，看到这一节标题的你，此刻的你内心是拒绝的，不明白在说些什么。我理解你的心情，不要惊慌、不要着急，且听我娓娓道来。

大家通过上面的内容大致的了解了一下什么是网络通信，那么在我们的日常生活中，哪里用到了网络通信呢，网络通信的整个流程又是什么样子的呢？我们要学的socket是怎么在网络中发挥作用的呢？让我们怀揣着这三个问题 来进行下面的学习。

二 客户端\服务端架构(哪里用到了网络通信)



我们使用qq、微信和别人聊天，通过浏览器来浏览页面、看京东的网站，通过优酷、快播(此处只是怀念一下)看片片啥的等等，通过无线打印机来打印一个word文档等，只要有无线、有网、有4G，我们就能好好的聊天，好好的看片片、好好

的购物什么的，对吧，那么这些操作都叫做网络通信，确切来说都需要使用网络通信，前提是你要有网(大家记着这个'网'，我下面会给大家详解)，原来生活中处处使用了网络通信，我们通过网络通信的不同形式：比如说qq是我们下载到电脑或者手机上的应用程序(qq应用程序就是人家腾讯开发的软件，放到你的电脑或者手机上供你使用的，大概明白应用程序意思就行，不用深究~~)，浏览器也是我们下载的应用程序，但是浏览器是通过页面来访问别人的网站的，而打印机我是通过我电脑上的word来操作使用的。根据这些不同的场景或者说不用的沟通方式，在业内划分了下面两个架构(架构：就是不同的组成结构)。在看下面的几个架构之前，我们需要知道什么是客户端，什么是服务端。客户端：安装在你电脑上的qq，浏览器(360浏览器、chrome浏览器、IE浏览器等)，当我们使用qq发送消息的时候，消息先发送到了腾讯，然后腾讯在转发到你朋友的qq上，此时你的qq就是客户端，腾讯就是服务端。当我们使用浏览器来看京东的网站的时候，我们电脑上的浏览器就叫做客户端，京东就叫做服务端。

客户端英文名称：Client(使用服务端的服务)，服务端英文名称：Server(一直运行着，等待服务别人，不能有一天访问百度，百度页面打不开，不行吧。)，下面所说的C\S架构就是说的Client\Server架构。

a.硬件C\S架构：打印机。

b.软件C\S架构：QQ、微信、优酷、暴风影音、浏览器(IE、火狐，360浏览器等)。其中浏览器又比较特殊，很多网站是基于浏览器来进行访问的，浏览器和各个网站服务端进行的通讯方式又常被称为B\S架构(浏览器英文名称：Browser)，web开发就是这个，后面大家知道有前端的课程对吧，前端就是浏览器上的知识，以后你会经常和浏览器打交道，学完前端就可以进行web开发全栈开发了。如果我把所有的东西都做成应用程序是不是很麻烦啊，要装很多的软件对吧，所有就开始有了B\S架构，只需要个浏览器就能使用很多的工具了，并且提供了一个统一入口，这也是为什么B\S架构火了起来。但是手机端的还是用的应用程序多一些，但是手机端B\S架构也是一个趋势，就像微信的小程序和公众号，为什么说是一个趋势呢，不仅仅是因为方便因为省钱，而是提供了一个统一的入口，其实微信早就实现了。统一入口是什么意思呢？就像我们公司经常用的一个公司内部管理系统，请假、打卡、报销、查客户等等，如果这些功能都需要打开一个网页或者app，是不是很难受啊，那么公司就做了这么一个系统，大家在这个系统上关于上班的一些你需要的功能就都能完成了，这就是统一入口。这也是一个开发思想，大程序分成几个小程序，开发速度也快，开发一个小功能就能上线，而不需要等着所有的功能全部开发完成才上线，解耦分治思想，公司做开发时这种思想很流行，迭代开发。说多了。。

不管哪个架构，他们都要进行网络通信，基本都要用socket，我们学习socket就是为了完成C\S架构项目的开发

三 网络通信的整个流程

还记得上面我说过的那个'网'吗，在这一节就给大家讲解，有些同学对网络是既熟悉又陌生，熟悉是因为我们都知道，我们安装一个路由器，拉一个网线，或者用无限路由器，连上网线或者连上wifi就能够上网购物、看片片、吃鸡了，但是这一系列的神操作到底是怎么让我们上网了呢？让我们起底揭秘！由于网络的内容非常的多，本篇博客主要是学socket网络编程，所以我把网络这方面的内容放到了我另外一篇博客上，这个博客很简单，不是什么深入研究类的博客，没有学过网络的或者说对网络不太熟悉的同学可以去看看，地址是[网络通信的整个流程](#)，有网络基础的同学，可以直接往下面学习，如果你自认上学时是个学渣，也可以过去大致溜一眼~~~将来你面向的是开发，所有网络这一块对你来讲就是大致知道就可以了，但是以后想在技术上有深造，那么就需要你深入的研究一下网络了，内容非常多，学海无涯~~

别忘了端口+IP能够确定一台电脑上的某一个应用程序~~

那么我们通过下面的代码简单看一下socket到底是个什么样子，大概如何使用：下面的程序就是一个应用程序，和qq啊、微信啊是一样的，都叫做应用程序。



```
import socket
#创建一个socket对象
server = socket.socket() #相当于创建了一部电话
ip_port = ('192.168.111.1',8001) #创建一个电话卡
server.bind(ip_port) #插上电话卡
server.listen(5) #监听着电话，我能监听5个，接到一个电话之后，后面还能有四个人给我打电话，但是后面这四个人都要排队等着，
print('11111')
#等着别人给我打电话，打来电话的时候，我就拿到了和对方的这个连线通道conn和对方的电话号码addr
conn,addr = server.accept() #阻塞住，一直等到有人连接我，连接之后得到一个元祖，里面是连线通道conn和对方的地址(ip+端口)
print('22222')
print(conn)
print('>>>>>>>>',addr)
while True:
    from_client_data = conn.recv(1024) #服务端必须通过两者之间的连接通道来收消息
    from_client_data = from_client_data.decode('utf-8')
```

```

print(from_client_data)
if from_client_data == 'bye':
    break
server_input = input('明威说>>>>: ')
conn.send(server_input.encode('utf-8'))
if server_input == 'bye':
    break
conn.close() #挂电话
server.close() #关手机

```



listen (3) , 这个3的意思是我连接着一个, 后面还可以有三个排队的, 也就是支持4个人的服务, 但是后面三个要排队。



```

# *_coding:utf-8_*
import socket
import time

client = socket.socket()
server_ip_port = ('192.168.111.1',8001)

client.connect(server_ip_port)

while True:
    client_input = input('小文说>>>>: ')
    client.send(client_input.encode('utf-8')) #给服务端发送消息
    if client_input == 'bye':
        break
    from_server_data = client.recv(1024)

    print('来自服务端的消息: ',from_server_data.decode('utf-8'))
    if from_server_data.decode('utf-8') == 'bye':
        break
client.close() #客户端挂电话

```



注意：先运行server，然后再运行client，然后你会发现client这个文件再输出台的地方让你输入内容，你输入一个内容然后回车，你会发现server那边的控制台就输出了以client发送的内容

今天的内容就到这里，今天学习的怎么样啊同学们，大家好好再重新过一遍，然后把练习题做一做。

=====

这里留两个小练习：

1.
 - server端：接收时间戳时间，转换成格式化时间
 - client端：每隔10秒中把时间戳发给server端，time.time()



```

import time

t = time.time() #获取时间戳
str_t = str(t)
print(str_t)

```

```
stru_t = time.localtime(float(str_t)) #将时间戳转换为结构化时间

s2 = time.strftime('%Y-%m-%d %H:%M:%S',stru_t) #将结构化时间转为格式化时间
print(s2,type(s2))
```



2. 一直对话的程序
 - server收一个发一个
 - client发一个收一个

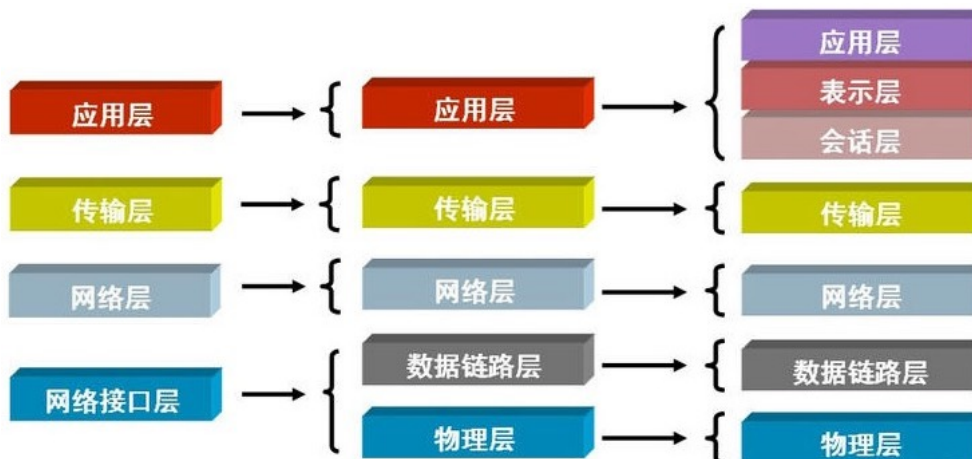
四 网络通信协议(互联网协议)

第二天再讲这里，大家第二天再看这里把~~~

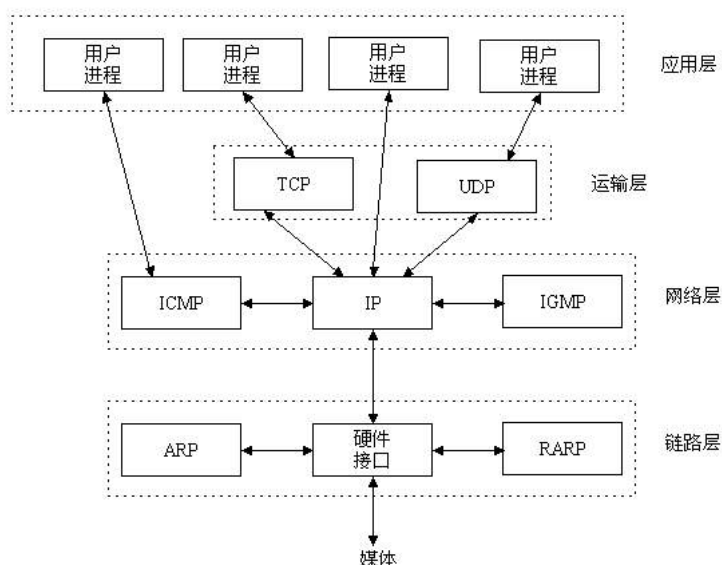
网络通信协议是网络传输的灵魂，非常重要，协议即准则，准则是传输消息的格式要求，那么我们从电脑上发出一个消息，到底是以什么样的消息格式发到了对方的手上呢，来看一看这里>>>，[网络通信协议](#)

五 osi七层模型

互联网的核心就是由一堆协议组成，协议就是标准，标准就是大家都认可的，所有人都按照这个来，这样大家都能够互相了解，互相深入了~~~比如全世界人通信的标准是英语



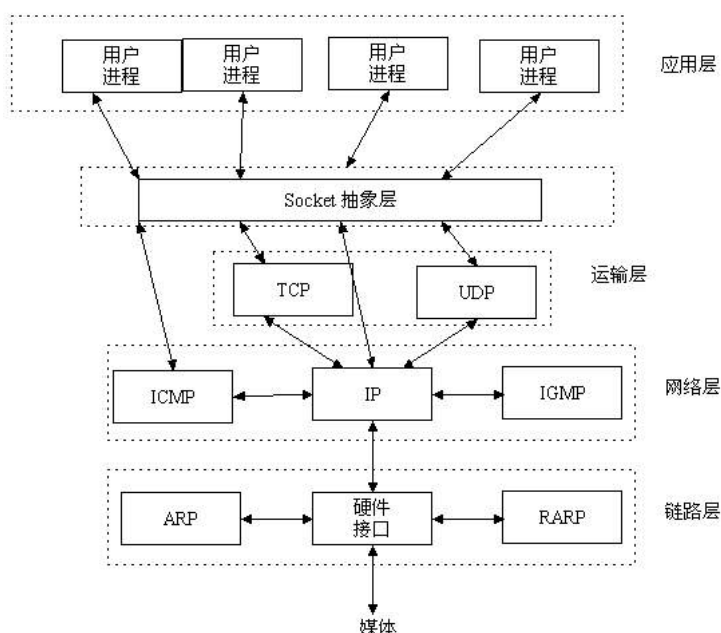
五层通信流程：



六 socket

结合上图来看，socket在哪一层呢，我们继续看下图

socket在内的五层通讯流程：



Socket又称为套接字，它是应用层与TCP/IP协议族通信的中间软件抽象层，它是一组接口。在设计模式中，Socket其实就是一个门面模式，它把复杂的TCP/IP协议族隐藏在Socket接口后面，对用户来说，一组简单的接口就是全部，让Socket去组织数据，以符合指定的协议。当我们使用不同的协议进行通信时就得使用不同的接口，还得处理不同协议的各种细节，这就增加了开发的难度，软件也不易于扩展(就像我们开发一套公司管理系统一样，报账、会议预定、请假等功能不需要单独写系统，而是一个系统上多个功能接口，不需要知道每个功能如何去实现的)。于是UNIX BSD就发明了socket这种东西，socket屏蔽了各个协议的通信细节，使得程序员无需关注协议本身，直接使用socket提供的接口来进行互联的不同主机间的进程的通信。这就好比操作系统给我们提供了使用底层硬件功能的系统调用，通过系统调用我们可以方便的使用磁盘（文件操作），使用内存，而无需自己去进行磁盘读写，内存管理。socket其实也是一样的东西，就是提供了tcp/ip协议的抽象，对外提供了一套接口，同过这个接口就可以统一、方便的使用tcp/ip协议的功能了。

其实站在你的角度上看，socket就是一个模块。我们通过调用模块中已经实现的方法建立两个进程之间的连接和通信。也有人将socket说成ip+port，因为ip是用来标识互联网中的一台主机的位置，而port是用来标识这台机器上的一个应用程序。所以我们只要确立了ip和port就能找到一个应用程序，并且使用socket模块来与之通信。

七 套接字socket的发展史及分类

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

基于文件类型的套接字家族

套接字家族的名字：AF_UNIX

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

基于网络类型的套接字家族

套接字家族的名字：AF_INET

(还有AF_INET6被用于ipv6，还有一些其他的地址家族，不过，他们要么是只用于某个平台，要么就是已经被废弃，或者是很少被使用，或者是根本没有实现，所有地址家族中，AF_INET是使用最广泛的一个，python支持很多种地址家族，但是由于我们只关心网络编程，所以大部分时候我们只使用AF_INET)

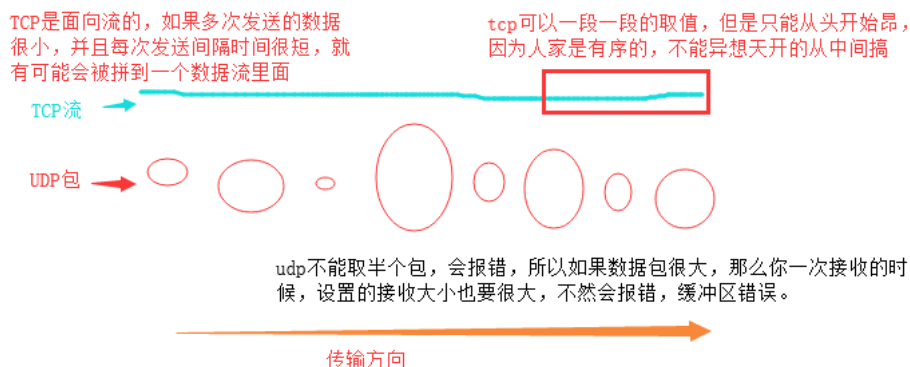
八 基于TCP和UDP两个协议下socket的通讯流程

1.TCP和UDP对比

TCP (Transmission Control Protocol) 可靠的、面向连接的协议 (eg:打电话)、传输效率低全双工通信 (发送缓存&接收缓存)、面向字节流。使用TCP的应用: Web浏览器; 文件传输程序。

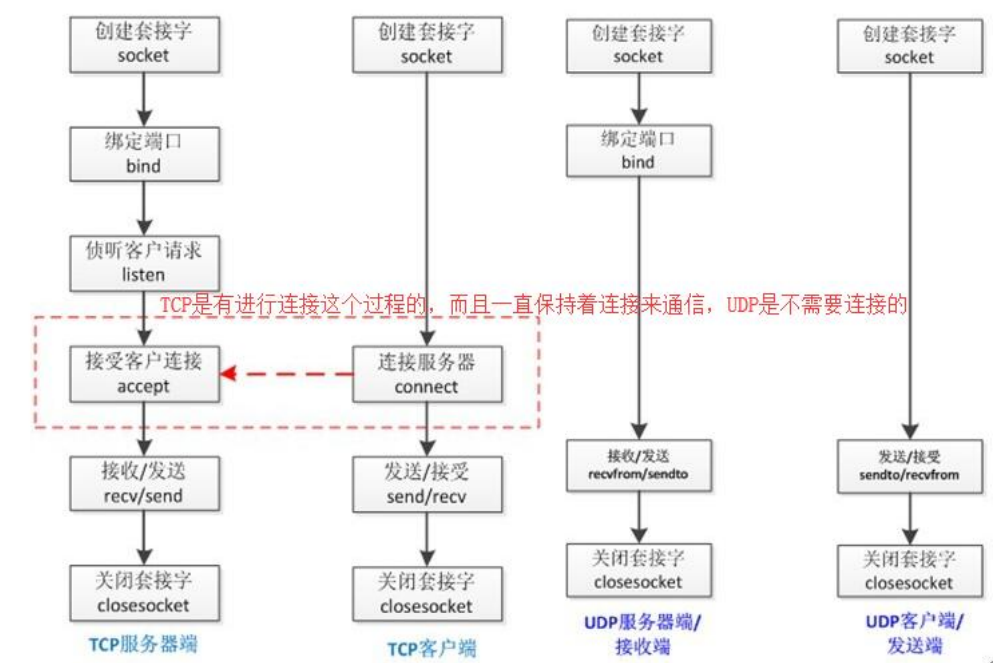
UDP (User Datagram Protocol) 不可靠的、无连接的服务, 传输效率高 (发送前时延小), 一对一、一对多、多对一、多对多、面向报文(数据包), 尽最大努力服务, 无拥塞控制。使用UDP的应用: 域名系统 (DNS); 视频流; IP语音 (VoIP)。

直接看图对比其中差异



继续往下看

TCP和UDP下socket差异对比图:

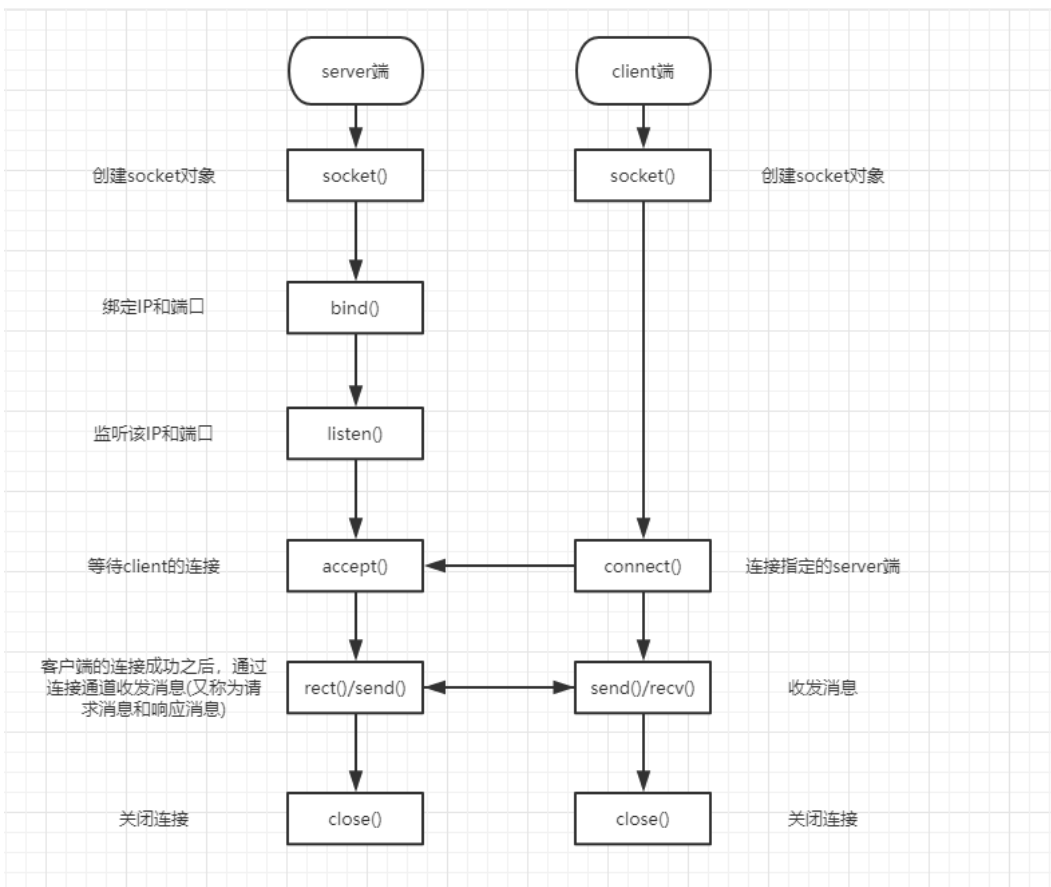


上面的图只是让大家感受一下TCP和UDP协议下, socket工作流程的不同, 两者之间的差异是tcp需要连接, udp不需要, 有些同学是不是有些迷糊, 老师, 这里面的bind、listen啥的都是什么东西啊, 我感觉人生是迷茫的! calm down! 下面我们就分开两者, 细细学习!

2.TCP协议下的socket

来吧! 先上图!

基于TCP的socket通讯流程图片:



虽然上图将通讯流程中的大致描述了一下socket各个方法的作用，但是还是要总结一下通讯流程(下面一段内容)

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

上代码感受一下，需要创建两个文件，文件名称随便起，为了方便看，我的两个文件名称为tcp_server.py(服务端)和tcp_client.py(客户端)，将下面的server端的代码拷贝到tcp_server.py文件中，将下面client端的代码拷贝到tcp_client.py的文件中，然后先运行tcp_server.py文件中的代码，再运行tcp_client.py文件中的代码，然后在pycharm下面的输出窗口看一下效果。

server端代码示例(如果比喻成打电话)

□



```

import socket
sk = socket.socket()
sk.bind(('127.0.0.1',8898)) #把地址绑定到套接字
sk.listen() #监听链接
conn,addr = sk.accept() #接受客户端链接
ret = conn.recv(1024) #接收客户端信息
print(ret) #打印客户端信息
conn.send(b'hi') #向客户端发送信息
conn.close() #关闭客户端套接字
sk.close() #关闭服务器套接字(可选)

```



client端代码示例

□




```
import socket
sk = socket.socket()      # 创建客户套接字
sk.connect(('127.0.0.1',8898)) # 尝试连接服务器
sk.send(b'hello!')
ret = sk.recv(1024)      # 对话(发送/接收)
print(ret)
sk.close()               # 关闭客户套接字
```



socket绑定IP和端口时可能出现下面的问题：

```
/Library/Frameworks/Python.framework/Versions/3.5/bin
Traceback (most recent call last):
  File "/Users/apple/Desktop/develop/PycharmProjects/
    sk.bind(('127.0.0.1',8898)) #把地址绑定到套接字
OSError: [Errno 48] Address already in use
```

解决办法：



```
#加入一条socket配置，重用ip和端口
import socket
from socket import SOL_SOCKET,SO_REUSEADDR
sk = socket.socket()
sk.setsockopt(SOL_SOCKET,SO_REUSEADDR,1) #在bind前加，允许地址重用
sk.bind(('127.0.0.1',8898)) #把地址绑定到套接字
sk.listen()      #监听链接
conn,addr = sk.accept() #接受客户端链接
ret = conn.recv(1024) #接收客户端信息
print(ret)       #打印客户端信息
conn.send(b'hi')  #向客户端发送信息
conn.close()      #关闭客户端套接字
sk.close()        #关闭服务器套接字(可选)
```



但是如果你加上了上面的代码之后还是出现这个问题：OSError: [WinError 10013] 以一种访问权限不允许的方式做了一个访问套接字的尝试。那么只能换端口了，因为你的电脑不支持端口重用。

记住一点，用socket进行通信，必须是一收一发对应好。

关于setsockopt可以看这篇文章。[关于setsockopt的使用](#)

提一下：网络相关或者需要和电脑上其他程序通信的程序才需要开一个端口。

在看UDP协议下的socket之前，我们还需要加一些内容来讲：看代码

server端



```
import socket
from socket import SOL_SOCKET,SO_REUSEADDR
sk = socket.socket()
# sk.setsockopt(SOL_SOCKET,SO_REUSEADDR,1)
sk.bind(('127.0.0.1',8090))
sk.listen()
conn,addr = sk.accept() #在这阻塞，等待客户端过来连接
```



```

while True:
    ret = conn.recv(1024) #接收消息 在这还是要阻塞，等待收消息
    ret = ret.decode('utf-8') #字节类型转换为字符串中文
    print(ret)
    if ret == 'bye':      #如果接到的消息为bye，退出
        break
    msg = input('服务端>>') #服务端发消息
    conn.send(msg.encode('utf-8'))
    if msg == 'bye':
        break

conn.close()
sk.close()

```



client端



```

import socket
sk = socket.socket()
sk.connect(('127.0.0.1',8090)) #连接服务端

while True:
    msg = input('客户端>>>') #input阻塞，等待输入内容
    sk.send(msg.encode('utf-8'))
    if msg == 'bye':
        break
    ret = sk.recv(1024)
    ret = ret.decode('utf-8')
    print(ret)
    if ret == 'bye':
        break
sk.close()

```



你会发现，第一个连接的客户端可以和服务端收发消息，但是第二个连接的客户端发消息服务端是收不到的

原因解释：

tcp属于长连接，长连接就是一直占用着这个链接，这个连接的端口被占用了，第二个客户端过来连接的时候，他是可以连接的，但是处于一个占线的状态，就只能等着去跟服务端建立连接，除非一个客户端断开了(优雅的断开可以，如果是强制断开就会报错，因为服务端的程序还在第一个循环里面)，然后就可以进行和服务端的通信了。什么是优雅的断开呢？看代码。

server端代码：



```

import socket
from socket import SOL_SOCKET,SO_REUSEADDR
sk = socket.socket()
# sk.setsockopt(SOL_SOCKET,SO_REUSEADDR,1) #允许地址重用，这个东西都说能解决问题，我非常不建议大家这么做，容易出问题
sk.bind(('127.0.0.1',8090))
sk.listen()
# 第二步演示，再加一层while循环
while True: #下面的代码全部缩进进去，也就是循环建立连接，但是不管怎么聊，只能和一个聊，也就是另外一个优雅的断了之后才
    #它不能同时和很多人聊，还是长连接的原因，一直占用着这个端口的连接，udp是可以的，然后我们学习udp
    conn,addr = sk.accept() #在这阻塞，等待客户端过来连接
    while True:
        ret = conn.recv(1024) #接收消息 在这还是要阻塞，等待收消息

```

```

ret = ret.decode('utf-8') #字节类型转换为字符串中文
print(ret)
if ret == 'bye':      #如果接到的消息为bye, 退出
    break
msg = input('服务端>>') #服务端发消息
conn.send(msg.encode('utf-8'))
if msg == 'bye':
    break
conn.close()

```



client端代码



```

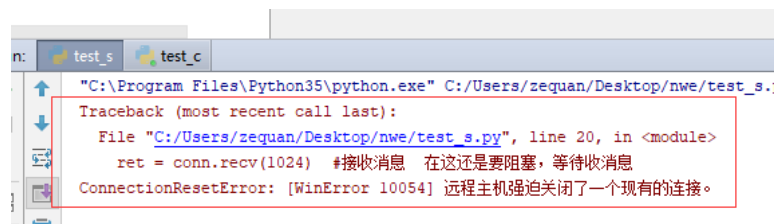
import socket
sk = socket.socket()
sk.connect(('127.0.0.1',8090)) #连接服务端

while True:
    msg = input('客户端>>>') #input阻塞, 等待输入内容
    sk.send(msg.encode('utf-8'))
    if msg == 'bye':
        break
    ret = sk.recv(1024)
    ret = ret.decode('utf-8')
    print(ret)
    if ret == 'bye':
        break
# sk.close()

```



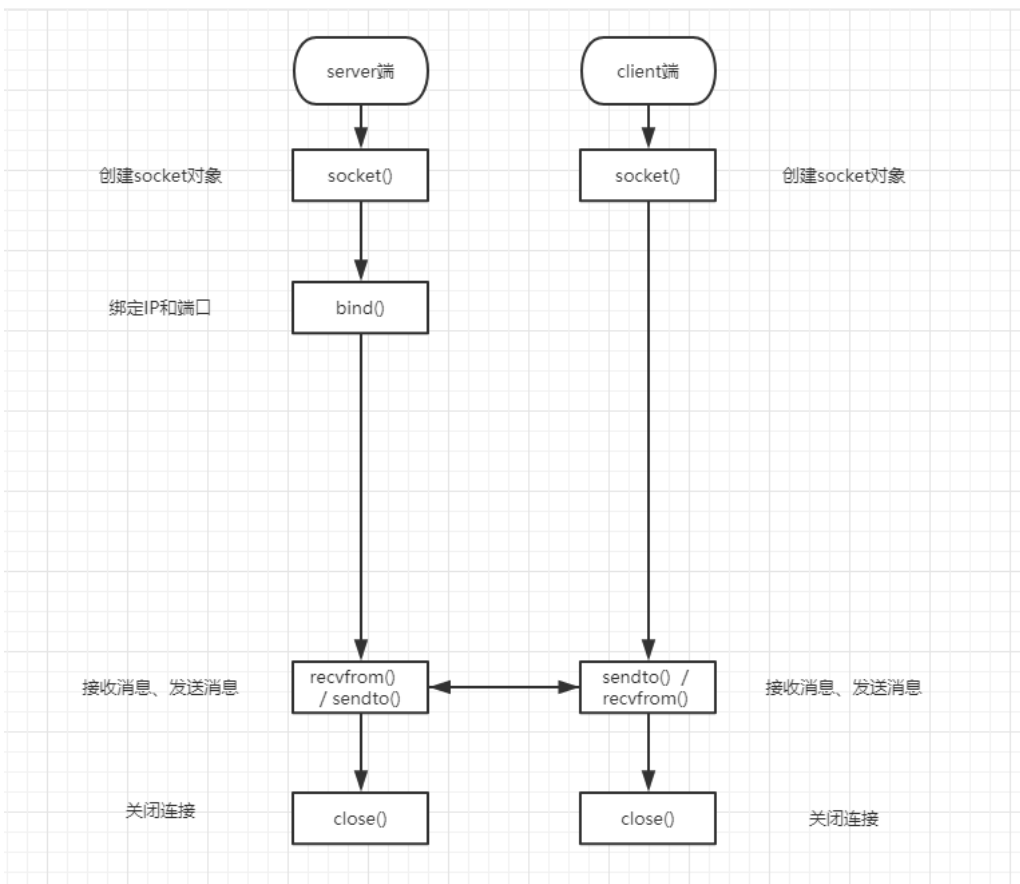
强制断开连接之后的报错信息:



3.UDP协议下的socket

老样子! 先上图!

基于UDP的socket通讯流程:



总结一下UDP下的socket通讯流程

先从服务器端说起。服务器端先初始化Socket，然后与端口绑定(bind)，recvfrom接收消息，这个消息有两项，消息内容和对方客户端的地址，然后回复消息时也要带着你收到的这个客户端的地址，发送回去，最后关闭连接，一次交互结束

上代码感受一下，需要创建两个文件，文件名称随便起，为了方便看，我的两个文件名称为udp_server.py(服务端)和udp_client.py(客户端)，将下面的server端的代码拷贝到udp_server.py文件中，将下面client端的代码拷贝到udp_client.py的文件中，然后先运行udp_server.py文件中的代码，再运行udp_client.py文件中的代码，然后在pycharm下面的输出窗口看一下效果。

server端代码示例

□



```
import socket
udp_sk = socket.socket(type=socket.SOCK_DGRAM) #创建一个服务器的套接字
udp_sk.bind(('127.0.0.1',9000)) #绑定服务器套接字
msg,addr = udp_sk.recvfrom(1024)
print(msg)
udp_sk.sendto(b'hi',addr) # 对话(接收与发送)
udp_sk.close() # 关闭服务器套接字
```



client端代码示例

□

```
import socket
ip_port=('127.0.0.1',9000)
udp_sk=socket.socket(type=socket.SOCK_DGRAM)
udp_sk.sendto(b'hello',ip_port)
back_msg,addr=udp_sk.recvfrom(1024)
```

```
print(back_msg.decode('utf-8'),addr)
```

类似于qq聊天的代码示例:



```
#_*_coding:utf-8_*_  
import socket  
ip_port=('127.0.0.1',8081)  
udp_server_sock=socket.socket(socket.AF_INET,socket.SOCK_DGRAM) #DGRAM:datagram 数据报文的意思，象征着UDP协议  
udp_server_sock.bind(ip_port)#你对外提供服务的端口就是这一个，所有的客户端都是通过这个端口和你进行通信的  
  
while True:  
    qq_msg,addr=udp_server_sock.recvfrom(1024)# 阻塞状态，等待接收消息  
    print('来自[%s:%s]的一条消息:\033[1;44m%s\033[0m' %(addr[0],addr[1],qq_msg.decode('utf-8')))  
    back_msg=input('回复消息: ').strip()  
  
    udp_server_sock.sendto(back_msg.encode('utf-8'),addr)
```



```
#_*_coding:utf-8_*_  
import socket  
BUFSIZE=1024  
udp_client_socket=socket.socket(socket.AF_INET,socket.SOCK_DGRAM)  
  
qq_name_dic={  
    'taibai':('127.0.0.1',8081),  
    'Jedan':('127.0.0.1',8081),  
    'Jack':('127.0.0.1',8081),  
    'John':('127.0.0.1',8081),  
}  
  
while True:  
    qq_name=input('请选择聊天对象: ').strip()  
    while True:  
        msg=input('请输入消息,回车发送,输入q结束和他的聊天: ').strip()  
        if msg == 'q':break  
        if not msg or not qq_name or qq_name not in qq_name_dic:continue  
        udp_client_socket.sendto(msg.encode('utf-8'),qq_name_dic[qq_name])# 必须带着自己的地址，这就是UDP不一样的地方，  
  
        back_msg,addr=udp_client_socket.recvfrom(BUFSIZE)# 同样也是阻塞状态，等待接收消息  
        print('来自[%s:%s]的一条消息:\033[1;44m%s\033[0m' %(addr[0],addr[1],back_msg.decode('utf-8')))  
  
    udp_client_socket.close()
```



接下来，给大家说一个真实的例子，也就是实际当中应用的，那么这是个什么例子呢？就是我们电脑系统上的时间，windows系统的时间是和微软的时间服务器上的时间同步的，而mac本是和苹果服务商的时间服务器同步的，这是怎么做的呢，首先他们的时间服务器上的时间是和国家同步的，你们用我的系统，那么你们的时间只要和我时间服务器上的时间同步就行了，对吧，我时间服务器是不是提供服务的啊，相当于一个服务端，我们的电脑就相当于客户端，就是通过UDP来搞的。

我们自制一个时间服务器的代码示例：



```
from socket import *
from time import strftime
import time
ip_port = ('127.0.0.1', 9000)
bufsize = 1024

tcp_server = socket(AF_INET, SOCK_DGRAM)
tcp_server.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
tcp_server.bind(ip_port)

while True:
    msg, addr = tcp_server.recvfrom(bufsize)
    print('==>', msg)
    stru_time = time.localtime() #当前的结构化时间
    if not msg:
        time_fmt = '%Y-%m-%d %X'
    else:
        time_fmt = msg.decode('utf-8')
        back_msg = strftime(time_fmt, stru_time)
        print(back_msg, type(back_msg))
        tcp_server.sendto(back_msg.encode('utf-8'), addr)

tcp_server.close()
```



```
from socket import *
ip_port=('127.0.0.1',9000)
bufsize=1024

tcp_client=socket(AF_INET,SOCK_DGRAM)

while True:
    msg=input('请输入时间格式(例%Y %m %d)>>: ').strip()
    tcp_client.sendto(msg.encode('utf-8'),ip_port)

    data=tcp_client.recv(bufsize)
    print('当前日期: ',str(data,encoding='utf-8'))
```



UDP来个小练习吧：

练习的需求是这样的：1、服务端需要提供的服务有：接收消息(时间格式的字符串)、将我的本地的时间转换成接收到的消息的格式(也就是个时间格式的字符串)、发回给客户端。2、客户端自行想一下怎么写。

TCP协议和UDP协议下socket的基本使用ok了，那我们来深入分析一下socket。(这一块的内容初学者不要看，对socket有些了解的同学可以研究一下，切记看不懂很正常，不要深究，现阶段你们就是学习应用为主！)>>>>看这里

>>>>[socket原理剖析](#)，里面包含socket中各个方法的作用和方法中的参数。

这里我列出两个简易描述socket各个参数和方法的图，共大家参考：

socket类型：

socket类型	描述
socket.AF_UNIX	只能够用于单一的Unix系统进程间通信
socket.AF_INET	服务器之间网络通信，ipv4
socket.AF_INET6	IPv6
socket.SOCK_STREAM	流式socket，for TCP
socket.SOCK_DGRAM	数据报式socket，for UDP
socket.SOCK_RAW	原始套接字，普通的套接字无法处理ICMP、IGMP等网络报文，而SOCK_RAW可以；其次，SOCK_RAW也可以处理特殊的IPv4报文；此外，利用原始套接字，可以通过IP_HDRINCL套接字选项由用户构造IP头。
socket.SOCK_SEQPACKET	可靠的连续数据包服务
创建TCP Socket:	s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
创建UDP Socket:	s=socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

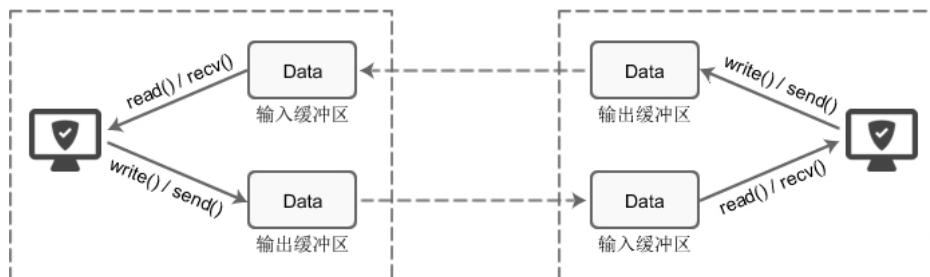
socket各个方法的解释：

socket函数	描述
服务端socket函数	
s.bind(address)	将套接字绑定到地址，在AF_INET下，以元组（host,port）的形式表示地址。
s.listen(backlog)	开始监听TCP传入连接。backlog指定在拒绝连接之前，操作系统可以挂起的最大连接数量。该值至少为1，大部分应用程序设为5就可以了。
s.accept()	接受TCP连接并返回（conn, address），其中conn是新的套接字对象，可以用来接收和发送数据。address是连接客户端的地址。
客户端socket函数	
s.connect(address)	连接到address处的套接字。一般address的格式为元组（hostname,port），如果连接出错，返回socket.error错误。
s.connect_ex(address)	功能与connect(address)相同，但是成功返回0，失败返回errno的值。
公共socket函数	
s.recv(bufsize[, flag])	接受TCP套接字的数据。数据以字符串形式返回，bufsize指定要接收的最大数据量。flag提供有关消息的其他信息，通常可以忽略。
s.send(string[, flag])	发送TCP数据。将string中的数据发送到连接的套接字。返回值是要发送的字节数量，该数量可能小于string的字节大小。
s.sendall(string[, flag])	完整发送TCP数据。将string中的数据发送到连接的套接字，但在返回之前会尝试发送所有数据。成功返回None，失败则抛出异常。
s.recvfrom(bufsize[, flag])	接受UDP套接字的数据。与recv()类似，但返回值是（data, address）。其中data是包含接收数据的字符串，address是发送数据的套接字地址。
s.sendto(string[, flag], address)	发送UDP数据。将数据发送到套接字，address是形式为（ipaddr, port）的元组，指定远程地址。返回值是发送的字节数。
s.close()	关闭套接字。
s.getpeername()	返回连接套接字的远程地址。返回值通常是元组（ipaddr, port）。
s.getsockname()	返回套接字自己的地址。通常是一个元组（ipaddr, port）
s.setsockopt(level, optname, value)	设置给定套接字选项的值。
s.getsockopt(level, optname[, buflen])	返回套接字选项的值。
s.settimeout(timeout)	设置套接字操作的超时期，timeout是一个浮点数，单位是秒。值为None表示没有超时期。一般，超时期应该在刚创建套接字时设置，因为它们可能用于连接的操作（如connect()）
s.gettimeout()	返回当前超时期的值，单位是秒，如果没有设置超时期，则返回None。
s.fileno()	返回套接字的文件描述符。
s.setblocking(flag)	如果flag为0，则将套接字设为非阻塞模式，否则将套接字设为阻塞模式（默认值）。非阻塞模式下，如果调用recv()没有发现任何数据，或send()调用无法立即发送数据，那么将引起socket.error异常。
s.makefile()	创建一个与该套接字相关连的文件

九 粘包现象

说粘包之前，我们先说两个内容，1.缓冲区、2.windows下cmd窗口调用系统指令

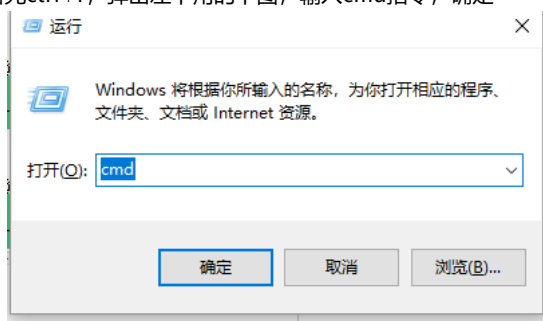
9.1 缓冲区（下面粘包现象的图里面还有关于缓冲区的解释）



socket缓冲区解释

9.2 windows下cmd窗口调用系统指令(linux下没有写出来，大家仿照windows的去摸索一下吧)

a.首先ctrl+r，弹出左下角的下图，输入cmd指令，确定



b.在打开的cmd窗口中输入dir（dir：查看当前文件夹下的所有文件和文件夹），你会看到下面的输出结果。

```
C:\WINDOWS\system32\cmd.exe
14 个文件      234,623 字节
4 个目录      21,843,202,048 可用字节

C:\Users\zequan\Desktop\rwe>dir
驱动器 C 中的卷是 Windows
卷的序列号是 1C88-905A

C:\Users\zequan\Desktop\rwe 的目录
2018/09/06 14:25 <DIR>      .
2018/09/06 14:25 <DIR>      ..
2018/09/06 14:26 <DIR>      .idea
2018/09/04 17:17          29,191 deep_socket_introduce.html
2018/09/03 10:51          19,233 internet_introduce.html
2018/09/03 14:15          27,787 internet_pro.html
2018/09/05 16:33          37,288 page_2.html
2018/08/27 11:09 <DIR>      Python开发
2018/08/27 14:26       104,895 Python开发.html
2018/09/06 11:28          979 tcp_nianbao_c.py
2018/09/06 11:28         2,000 tcp_nianbao_s.py
2018/09/06 10:53          782 test_c.py
2018/09/06 10:53         3,341 test_s.py
2018/09/05 11:08          788 udp_client.py
2018/09/06 11:02         1,503 udp_nianbao_c.py
2018/09/06 11:03          725 udp_nianbao_s.py
2018/09/05 11:10          696 udp_server.py
2018/09/04 20:52         6,831 关于socket的setsockopt的使用.html
14 个文件      236,039 字节
4 个目录      22,033,096,704 可用字节
```

另外还有ipconfig（查看当前电脑的网络信息），在windows没有ls这个指令(ls在linux下是查看当前文件夹下所有文件和文件夹的指令，和windows下的dir是类似的），那么没有这个指令就会报下面这个错误

```
C:\Users\zequan\Desktop\rwe>ls
ls 不是内部或外部命令，也不是可运行的程序
或批处理文件。
```



linux shell中一个运行多个命令，命令间用;隔开即可

windows的命令提示符中运行多条命令用的是：&&、||、&

aa && bb

就是执行aa，成功后再执行bb

aa || bb

先执行aa，若执行成功则不再执行bb，若失败则执行bb

a & b

表示执行a再执行b，无论a是否成功

“执行成功”的意思是返回的errorlevel=0

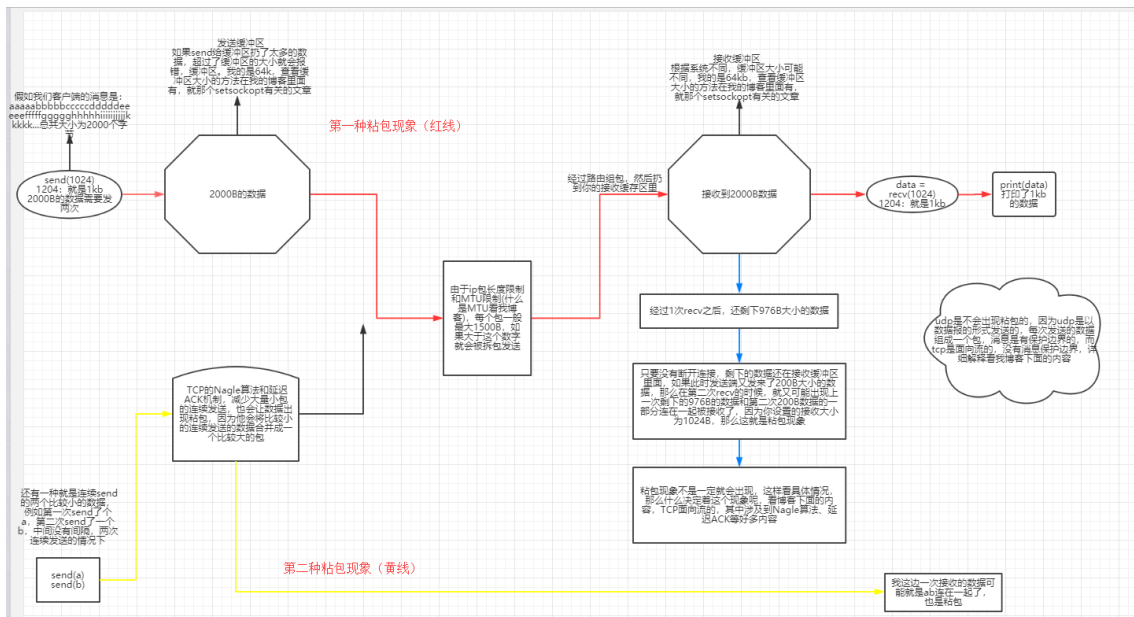


为什么要说这个系统指令呢，是希望借助系统指令和指令输出的结果来模拟一下粘包现象，那什么是粘包呢？

今天的内容就先到这里，明天我们认识粘包~~，大家好好理解练习一下把。

9.3 粘包现象 (两种)

先上图：（本图是我做出来为了让小白同学有个大致的了解用的，其中很多地方更加的复杂，那就需要将来大家有多余的精力的时候去做一些深入的研究了，这里我就不带大家搞啦）



关于MTU大家可以看看这篇文章 <https://yq.aliyun.com/articles/222535> 还有百度百科 [MTU百科](#)

MTU简单解释：

MTU是Maximum Transmission Unit的缩写。意思是网络上传送的最大数据包。MTU的单位是字节。大部分网络设备的MTU都是15

关于上图中提到的Nagle算法等建议大家去看一看Nagle算法、延迟ACK、linux下的TCP_NODELAY和TCP_CORK，这些内容等你们把python学好以后再去研究吧，网络的内容实在太多啦，也就是说大家需要努力的过程还很长，加油！

超出缓冲区大小会报下面的错误，或者udp协议的时候，你的一个数据包的大小超过了你一次recv能接受的大小，也会报下面的错误，tcp不会，但是超出缓存区大小的时候，肯定会报这个错误。

```

C:\Users\Administrator\AppData\Local\Programs\Python\Python36\python.exe C:/Users/Administrator/PycharmProjects/s9/day31/黏包问题/server.py
>>> ipconfig
Traceback (most recent call last):
  File "C:/Users/Administrator/PycharmProjects/s9/day31/黏包问题/server.py", line 10, in <module>
    msg, addr = sk.recvfrom(1024)
  OSError: [WinError 10040] 一个在数据报套接字上发送的消息大于内部消息缓冲区或其他一些网络限制，或该用户用于接收数据报的缓冲区比数据报小。

```

9.4 模拟一个粘包现象

在模拟粘包之前，我们先学习一个模块subprocess。

```

import subprocess
cmd = input('请输入指令>>>')
res = subprocess.Popen(
    cmd,                #字符串指令: 'dir','ipconfig',等等
    shell=True,         #使用shell, 就相当于使用cmd窗口
    stderr=subprocess.PIPE, #标准错误输出, 凡是输入错误指令, 错误指令输出的报错信息就会被它拿到
    stdout=subprocess.PIPE, #标准输出, 正确指令的输出结果被它拿到
)
print(res.stdout.read().decode('gbk'))
print(res.stderr.read().decode('gbk'))

```

注意:

如果是windows, 那么res.stdout.read()读出的就是GBK编码的, 在接收端需要用GBK解码

且只能从管道里读一次结果, PIPE称为管道。

下面是subprocess和windows上cmd下的指令的对应示意图: subprocess的stdout.read()和stderr.read(), 拿到的结果是bytes类型, 所以需要转换为字符串打印出来看。

The diagram illustrates the relationship between a Windows command prompt (CMD) and the subprocess module in Python. It consists of three main components: a Windows command prompt window, a PyCharm code editor, and a terminal window.

- Windows Command Prompt (CMD):** Shows the execution of the `dir` command. The output is displayed in a list of files and directories. A red box highlights the command `dir` and the output, with a red arrow pointing to the subprocess code.
- PyCharm Code Editor:** Shows the Python code for the subprocess module. The code is as follows:


```

import subprocess
cmd = input('请输入指令>>>')
res = subprocess.Popen(
    cmd,                #字符串指令: 'dir','ipconfig',等等
    shell=True,         #使用shell, 就相当于使用cmd窗口
    stderr=subprocess.PIPE, #标准错误输出, 凡是输入错误指令, 错误指令输出的报错信息就会被它拿到
    stdout=subprocess.PIPE, #标准输出, 正确指令的输出结果被它拿到
)
print(res.stdout.read().decode('gbk'))
print(res.stderr.read().decode('gbk'))

```

 Red boxes highlight the `cmd` variable, the `subprocess.Popen` function, and the `print` statements. Red arrows point from the CMD window to these elements.
- Terminal Window:** Shows the execution of the `dir` command. The output is displayed in a list of files and directories. A red box highlights the command `dir` and the output, with a red arrow pointing to the subprocess code.

Additional annotations include:

- A red arrow pointing from the `cmd` variable to the `cmd` parameter in the `subprocess.Popen` function.
- A red arrow pointing from the `stdout` parameter in the `subprocess.Popen` function to the `print(res.stdout.read().decode('gbk'))` statement.
- A red arrow pointing from the `stderr` parameter in the `subprocess.Popen` function to the `print(res.stderr.read().decode('gbk'))` statement.
- A red arrow pointing from the `print` statements to the terminal window output.

好，既然我们会使用subprocess了，那么我们就通过它来模拟一个粘包，终于到模拟粘包现象了，这一天真的是好累。

tcp粘包演示(一):

先从上面粘包现象中的第一种开始：**接收方没有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）**

server端代码示例：



```
cket import *
import subprocess

ip_port=('127.0.0.1',8080)
BUFSIZE=1024

tcp_socket_server=socket(AF_INET,SOCK_STREAM)
tcp_socket_server.setsockopt(SOL_SOCKET,SO_REUSEADDR,1)
tcp_socket_server.bind(ip_port)
tcp_socket_server.listen(5)

while True:
    conn,addr=tcp_socket_server.accept()
    print('客户端>>>',addr)

    while True:
        cmd=conn.recv(BUFSIZE)
        if len(cmd) == 0:break

        res=subprocess.Popen(cmd.decode('gbk'),shell=True,
                               stdout=subprocess.PIPE,
                               stdin=subprocess.PIPE,
                               stderr=subprocess.PIPE)

        stderr=res.stderr.read()
        stdout=res.stdout.read()
        conn.send(stderr)
        conn.send(stdout)
```



client端代码示例：



```
import socket
ip_port = ('127.0.0.1',8080)
size = 1024
tcp_sk = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
res = tcp_sk.connect(ip_port)
while True:
    msg=input('>>: ').strip()
    if len(msg) == 0:continue
    if msg == 'quit':break

    tcp_sk.send(msg.encode('utf-8'))
    act_res=tcp_sk.recv(size)
    print('接收的返回结果长度为>',len(act_res))
    print('std>>>',act_res.decode('gbk')) #windows返回的内容需要用gbk来解码，因为windows系统的默认编码为gbk
```



tcp粘包演示(二): 发送数据时间间隔很短, 数据也很小, 会合到一起, 产生粘包

server端代码示例: (如果两次发送有一定的时间间隔, 那么就不会出现这种粘包情况, 试着在两次发送的中间加一个time.sleep(1))



```
from socket import *
ip_port=('127.0.0.1',8080)

tcp_socket_server=socket(AF_INET,SOCK_STREAM)
tcp_socket_server.bind(ip_port)
tcp_socket_server.listen(5)
conn,addr=tcp_socket_server.accept()
data1=conn.recv(10)
data2=conn.recv(10)

print('----->',data1.decode('utf-8'))
print('----->',data2.decode('utf-8'))

conn.close()
```



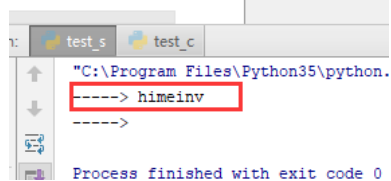
client端代码示例:



```
import socket
BUFSIZE=1024
ip_port=('127.0.0.1',8080)
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
# res=s.connect_ex(ip_port)
res=s.connect(ip_port)
s.send('hi'.encode('utf-8'))
s.send('meinv'.encode('utf-8'))
```



示例二的结果: 全部被第一个recv接收了



```
"C:\Program Files\Python35\python.
-----> hi meinv
----->
Process finished with exit code 0
```

udp粘包演示: 注意: udp是面向包的, 所以udp是不存在粘包的
server端代码示例:



```
import socket
from socket import SOL_SOCKET,SO_REUSEADDR,SO_SNDBUF,SO_RCVBUF
sk = socket.socket(type=socket.SOCK_DGRAM)
# sk.setsockopt(SOL_SOCKET,SO_RCVBUF,80*1024)
sk.bind(('127.0.0.1',8090))
msg,addr = sk.recvfrom(1024)
```

```

while True:
    cmd = input('>>>>')
    if cmd == 'q':
        break
    sk.sendto(cmd.encode('utf-8'),addr)
    msg,addr = sk.recvfrom(1032)
    # print('>>>>', sk.getsockopt(SOL_SOCKET, SO_SNDBUF))
    # print('>>>>', sk.getsockopt(SOL_SOCKET, SO_RCVBUF))
    print(len(msg))
    print(msg.decode('utf-8'))

sk.close()

```



client端代码示例:



```

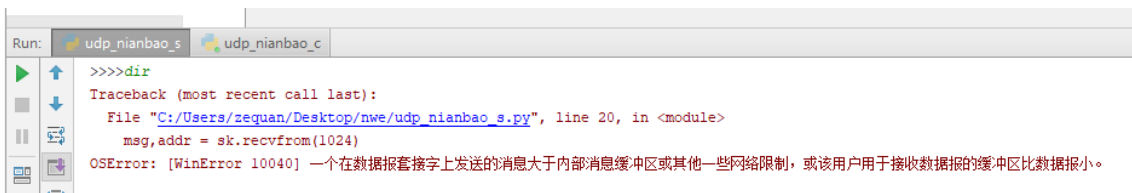
import socket
from socket import SOL_SOCKET,SO_REUSEADDR,SO_SNDBUF,SO_RCVBUF
sk = socket.socket(type=socket.SOCK_DGRAM)
# sk.setsockopt(SOL_SOCKET,SO_RCVBUF,80*1024)
sk.bind(('127.0.0.1',8090))
msg,addr = sk.recvfrom(1024)
while True:
    cmd = input('>>>>')
    if cmd == 'q':
        break
    sk.sendto(cmd.encode('utf-8'),addr)
    msg,addr = sk.recvfrom(1024)
    # msg,addr = sk.recvfrom(1218)
    # print('>>>>', sk.getsockopt(SOL_SOCKET, SO_SNDBUF))
    # print('>>>>', sk.getsockopt(SOL_SOCKET, SO_RCVBUF))
    print(len(msg))
    print(msg.decode('utf-8'))

sk.close()

```



在udp的代码中，我们在server端接收返回消息的时候，我们设置的recvfrom(1024)，那么当我输入的执行指令为'dir'的时候，dir在我当前文件夹下输出的内容大于1024，然后就报错了，报的错误也是下面这个：



解释原因：是因为udp是面向报文的，意思就是每个消息是一个包，你接收端设置接收大小的时候，必须要比你发的这个包要大，不然一次接收不了就会报这个错误，而tcp不会报错，这也是为什么ucp会丢包的原因之一，这个和我们上面缓冲区那个错误的报错原因是不一样的。

9.5 TCP会粘包、UDP永远不会粘包

看下面的解释原因：



发送端可以是一K—K地发送数据，而接收端的应用程序可以两K两K地提走数据，当然也有可能一次提走3K或6K数据，或者一次只提走例如基于tcp的套接字客户端往服务端上传文件，发送时文件内容是按照一段一段的字节流发送的，在接收方看了，根本不知道该文件的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，不知道一次性提取多少字节的数据所造成的。

此外，发送方引起的粘包是由TCP协议本身造成的，TCP为提高传输效率，发送方往往要收集到足够多的数据后才发送一个TCP段。若

- 1.TCP (transport control protocol，传输控制协议)是面向连接的，面向流的，提供高可靠性服务。收发两端（客户端和服务端）
- 2.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，，
- 3.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是基于消息的，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tcp的协议数据不会丢，没有收完包，下次接收，会继续上次继续接收，己端总是在收到ack时才会清除缓冲区内容。数据是可靠的，但



补充两个问题：



补充问题一：为何tcp是可靠传输，udp是不可靠传输

tcp在数据传输时，发送端先把数据发送到自己的缓存中，然后协议控制将缓存中的数据发往对端，对端返回一个ack=1，发送端则而udp发送数据，对端是不会返回确认信息的，因此不可靠

补充问题二：send(字节流)和sendall

send的字节流是先放入己端缓存，然后由协议控制将缓存内容发往对端，如果待发送的字节流大小大于缓存剩余空间，那么数据丢失

用UDP协议发送时，用sendto函数最大能发送数据的长度为： $65535 - \text{IP头}(20) - \text{UDP头}(8) = 65507$ 字节。用sendto函数发送数据时

用TCP协议发送时，由于TCP是数据流协议，因此不存在包大小的限制（暂不考虑缓冲区的大小），这是指在用send函数时，数据长度



粘包的原因：**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**

学到这里，我们留一个小作业（做不做是你的事情，我的事情是真心的教会你，希望你尊重自己的努力）：实现一个简单的网盘功能。

网盘

文件的 上传 下载

server端 和 client端

登录 客户端登录 将用户名和密码发给服务端 服务端确认信息之后

上传下载

选择上传 / 下载

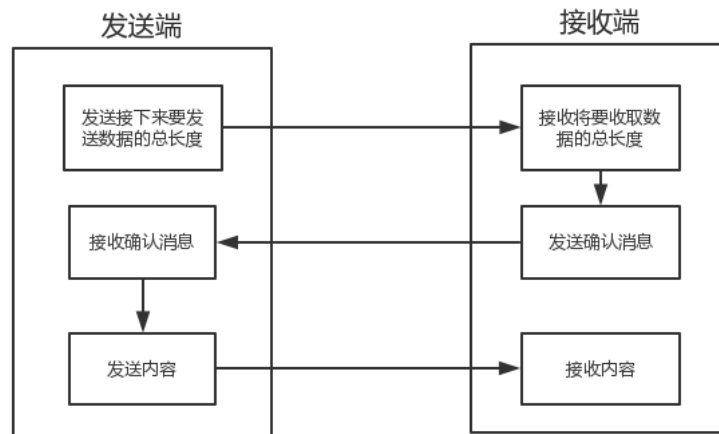
上传：选择要上传的文件路径，在server创建一个同名的空文件.cache

下载：选择要下载的文件路径，在client端创建一个同名的空文件

十 粘包的解决方案

解决方案（一）：

问题的根源在于，接收端不知道发送端将要传送的字节流的长度，所以解决粘包的方法就是围绕，如何让发送端在发送数据前，把自己将要发送的字节流总大小让接收端知晓，然后接收端发一个确认消息给发送端，然后发送端再发送过来后面的真实内容，接收端再来一个死循环接收完所有数据。



看代码示例：

server端代码

```
import socket, subprocess
ip_port=('127.0.0.1',8080)
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

s.bind(ip_port)
s.listen(5)

while True:
    conn,addr=s.accept()
    print("客户端",addr)
    while True:
        msg=conn.recv(1024)
        if not msg:break
        res=subprocess.Popen(msg.decode('utf-8'),shell=True,\
                               stdin=subprocess.PIPE,\
                               stderr=subprocess.PIPE,\
                               stdout=subprocess.PIPE)
```



```

err=res.stderr.read()
if err:
    ret=err
else:
    ret=res.stdout.read()
data_length=len(ret)
conn.send(str(data_length).encode('utf-8'))
data=conn.recv(1024).decode('utf-8')
if data == 'recv_ready':
    conn.sendall(ret)
conn.close()

```



client端代码示例



```

import socket,time
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
res=s.connect_ex(('127.0.0.1',8080))

while True:
    msg=input('>>: ').strip()
    if len(msg) == 0:continue
    if msg == 'quit':break

    s.send(msg.encode('utf-8'))
    length=int(s.recv(1024).decode('utf-8'))
    s.send('recv_ready'.encode('utf-8'))
    send_size=0
    rcv_size=0
    data=b''
    while rcv_size < length:
        data+=s.recv(1024)
        rcv_size+=len(data)

    print(data.decode('utf-8'))

```



解决方案（二）：

通过struct模块将需要发送的内容的长度进行打包，打包成一个4字节长度的数据发送到对端，对端只要取出前4个字节，然后对这四个字节的数据进行解包，拿到你要发送的内容的长度，然后通过这个长度来继续接收我们实际要发送的内容。不是很好理解是吧？哈哈，没关系，看下面的解释~~

为什么要说一下这个模块呢，因为解决方案（一）里面你发现，我每次要先发送一个我的内容的长度，需要接收端接收，并切需要接收端返回一个确认消息，我发送端才能发后面真实的内容，这样是为了保证数据可靠性，也就是接收双方能顺利沟通，但是多了一次发送接收的过程，为了减少这个过程，我们就要使struct来发送你需要发送的数据的长度，来解决上面我们所说的通过发送内容长度来解决粘包的问题。

关于struct的介绍：

了解C语言的人，一定会知道struct结构体在C语言中的作用，不了解C语言的同学也没关系，不影响，其实它就是定义了一种结构，里面包含不同类型的数据(int,char,bool等等)，方便对某一结构对象进行处理。而在网络通信当中，大多传递的数据是以二进制流（binary data）存在的。当传递字符串时，不必担心太多的问题，而当传递诸如int、char之类的基本数据的时候，就需要有一种机制将某些特定的结构体类型打包成二进制流的字符串然后再网络传输，而接收端也应该可以通过某种机制进行解包还原出原始的结构体数据。python中的struct模块就提供了这样的机制，该模块的主要作用就是对python基本类型值与用python字符串格式表示的C struct类型间的转化（This module performs conversions between Python values and C structs represented as Python strings.）。

struct模块的使用：struct模块中最重要的两个函数是pack()打包, unpack()解包。

Format	C Type	Python type	Standard size	Notes
x	pad byte	no value		
c	char	string of length 1	1	
b	signed char	integer	1	(3)
B	unsigned char	integer	1	(3)
?	_Bool	bool	1	(1)
h	short	integer	2	(3)
H	unsigned short	integer	2	(3)
i	int	integer	4	(3)
I	unsigned int	integer	4	(3)
l	long	integer	4	(3)
L	unsigned long	integer	4	(3)
q	long long	integer	8	(2), (3)
Q	unsigned long long	integer	8	(2), (3)
f	float	float	4	(4)
d	double	float	8	(4)
s	char[]	string		
p	char[]	string		
P	void *	integer		(5), (3)

pack(): #我在这里只介绍一下'i'这个int类型, 上面的图中列举除了可以打包的所有的数据类型, 并且struct除了pack和unpack两个方法之外还有好多别的方法和用法, 大家以后找时间可以去研究一下, 这里我就不做介绍啦, 网上的教程很多~~

```
import struct
a=12
# 将a变为二进制
bytes=struct.pack('i',a)
```

-----stru

pack方法图解:

?	_Bool	bool	1	(1)
h	short	integer	2	(3)
H	unsigned short	integer	2	(3)
i	int	integer	4	(3)
I	unsigned int	integer	4	(3)
l	long	integer	4	(3)
L	unsigned long	integer	4	(3)
q	long long	integer	8	(2), (3)
Q	unsigned long long	integer	8	(2), (3)
f	float	float	4	(4)
d	double	float	8	(4)
s	char[]	string		
p	char[]	string		
P	void *	integer		(5), (3)

pack():

```
import struct
a=12
# 将a变为二进制
bytes=struct.pack('i',a)
```

将int类型的数据12打包

一个4字节的bytes类型数据

'i' 表示后面的数据为int类型

unpack():

注意, unpack返回的是tuple !!

a,=struct.unpack('i',bytes) #将bytes类型的数据解包后, 拿到int类型数据

好, 到这里我们将struct这个模块将int类型的数据打包成四个字节的的方法了, 那么我们就来使用它解决粘包吧。

先看一段伪代码示例:



```

import json,struct
#假设通过客户端上传1T:1073741824000的文件a.txt

#为避免粘包,必须自定义报头
header={'file_size':1073741824000,'file_name':'/a/b/c/d/e/a.txt','md5':'8f6bf8347faa4924a76856701edb0f3'} #1T数据,

#为了该报头能传送,需要序列化并且转为bytes, 因为bytes只能将字符串类型的数据转换为bytes类型的, 所有需要先序列化一下这个字典
head_bytes=bytes(json.dumps(header),encoding='utf-8') #序列化并转成bytes,用于传输

#为了让客户端知道报头的长度,用struct将报头长度这个数字转成固定长度:4个字节
head_len_bytes=struct.pack('i',len(head_bytes)) #这4个字节里只包含了一个数字,该数字是报头的长度

#客户端开始发送
conn.send(head_len_bytes) #先发报头的长度,4个bytes
conn.send(head_bytes) #再发报头的字节格式
conn.sendall(文件内容) #然后发真实内容的字节格式

#服务端开始接收
head_len_bytes=s.recv(4) #先收报头4个bytes,得到报头长度的字节格式
x=struct.unpack('i',head_len_bytes)[0] #提取报头的长度

head_bytes=s.recv(x) #按照报头长度x,收取报头的bytes格式
header=json.loads(json.dumps(header)) #提取报头

#最后根据报头的内容提取真实的数据,比如
real_data_len=s.recv(header['file_size'])
s.recv(real_data_len)

```



下面看正式的代码：

server端代码示例：报头：就是消息的头部信息，我们要发送的真实内容为报头后面的内容。



```

import socket,struct,json
import subprocess
phone=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
phone.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1) #忘了这是干什么的了，地址重用？想起来了嘛~

phone.bind(('127.0.0.1',8080))
phone.listen(5)
while True:
    conn,addr=phone.accept()
    while True:
        cmd=conn.recv(1024)
        if not cmd:break
        print('cmd: %s' %cmd)
        res=subprocess.Popen(cmd.decode('utf-8'),
                               shell=True,
                               stdout=subprocess.PIPE,
                               stderr=subprocess.PIPE)
        err=res.stderr.read()
        if err:
            back_msg=err
        else:
            back_msg=res.stdout.read()
        conn.send(struct.pack('i',len(back_msg))) #先发back_msg的长度

```

```
conn.sendall(back_msg) #在发真实的内容
```

#其实就是连续的将长度和内容一起发出去，那么整个内容的前4个字节就是我们打包的后面内容的长度，对吧

```
conn.close()
```



client端代码示例:



```
import socket,time,struct
s=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
res=s.connect_ex(('127.0.0.1',8080))
while True:
    msg=input('>>: ').strip()
    if len(msg) == 0:continue
    if msg == 'quit':break
    s.send(msg.encode('utf-8')) #发送给一个指令
    l=s.recv(4) #先接收4个字节的数据，因为我们将要发送过来的内容打包成了4个字节，所以先取出4个字节
    x=struct.unpack('i',l)[0] #解包，是一个元祖，第一个元素就是我们的内容的长度
    print(type(x),x)
    # print(struct.unpack('i',l))
    r_s=0
    data=b''
    while r_s < x: #根据内容的长度来继续接收4个字节后面的内容。
        r_d=s.recv(1024)
        data+=r_d
        r_s+=len(r_d)
    # print(data.decode('utf-8'))
    print(data.decode('gbk')) #windows默认gbk编码
```



复杂一些的代码示例

server端:



```
import socket,struct,json
import subprocess
phone=socket.socket(socket.AF_INET,socket.SOCK_STREAM)
phone.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)

phone.bind(('127.0.0.1',8080))
phone.listen(5)

while True:
    conn,addr=phone.accept()
    while True:
        cmd=conn.recv(1024)
        if not cmd:break
        print('cmd: %s' %cmd)

        res=subprocess.Popen(cmd.decode('utf-8'),
                               shell=True,
                               stdout=subprocess.PIPE,
                               stderr=subprocess.PIPE)
```

```

err=res.stderr.read()
print(err)
if err:
    back_msg=err
else:
    back_msg=res.stdout.read()

headers={'data_size':len(back_msg)}
head_json=json.dumps(headers)
head_json_bytes=bytes(head_json,encoding='utf-8')

conn.send(struct.pack('i',len(head_json_bytes))) #先发报头的长度
conn.send(head_json_bytes) #再发报头
conn.sendall(back_msg) #在发真实的内容

conn.close()

```



客户端：



```

from socket import *
import struct,json

ip_port=('127.0.0.1',8080)
client=socket(AF_INET,SOCK_STREAM)
client.connect(ip_port)

while True:
    cmd=input('>>: ')
    if not cmd:continue
    client.send(bytes(cmd,encoding='utf-8'))

    head=client.recv(4)
    head_json_len=struct.unpack('i',head)[0]
    head_json=json.loads(client.recv(head_json_len).decode('utf-8'))
    data_len=head_json['data_size']

    recv_size=0
    recv_data=b''
    while recv_size < data_len:
        recv_data+=client.recv(1024)
        recv_size+=len(recv_data)

    #print(recv_data.decode('utf-8'))
    print(recv_data.decode('gbk')) #windows默认gbk编码

```



其实上面复杂的代码做了个什么事情呢，就是自定义了报头：

```

head = {'filename': 'test', 'filesize': 409600, 'filetype': 'txt', 'filepath': r'\\user\\bin'}
# 报头的长度 # 先接收4个字节
# send(head) # 报头 # 根据这4个字节获取报头
# send(file) # 报文 # 从报头中获取filesize, 然后根据filesize接收文件

```

自定义报头，也可以称为我们自定义的协议

有同学问：老师，你为啥多次send啊，其实多次send和将数据拼接起来send一次是一样的，因为我们约定好了，你接收的时候先接收4个字节，然后再接收后面的内容。



整个流程的大致解释：我们可以把报头做成字典，字典里包含将要发送的真实数据的描述信息(大小啊之类的)，然后json序列化，然后我们在网络上传输的所有数据 都叫做数据包，数据包里的所有数据都叫做报文，报文里面不止有你的数据，还有ip地址、mac地址、端发送时：

先发报头长度

再编码报头内容然后发送

最后发真实内容

接收时：

先手报头长度，用struct取出来

根据取出的长度收取报头内容，然后解码，反序列化

从反序列化的结果中取出待取数据的描述信息，然后去取真实的数据内容



FTP上传下载文件的代码（简易版）



```
import socket
import struct
import json
sk = socket.socket()
# buffer = 4096 # 当双方的这个接收发送的大小比较大的时候，就像这个4096，就会丢数据，这个等我查一下再告诉大家，改小了#
buffer = 1024 #每次接收数据的大小
sk.bind(('127.0.0.1',8090))
sk.listen()

conn,addr = sk.accept()
#接收
head_len = conn.recv(4)
head_len = struct.unpack('i',head_len)[0] #解包
json_head = conn.recv(head_len).decode('utf-8') #反序列化
head = json.loads(json_head)
filesize = head['filesize']
with open(head['filename'],'wb') as f:
    while filesize:
        if filesize >= buffer: #>=是因为如果刚好等于的情况出现也是可以的。
            content = conn.recv(buffer)
            f.write(content)
            filesize -= buffer
        else:
            content = conn.recv(buffer)
            f.write(content)
            break

conn.close()
sk.close()
```



```
import os
import json
import socket
import struct
sk = socket.socket()
sk.connect(('127.0.0.1',8090))
```

```

buffer = 1024 #读取文件的时候，每次读取的大小
head = {
    'filepath':r'D:\打包程序', #需要下载的文件路径，也就是文件所在的文件夹
    'filename':'xxx.mp4', #改成上面filepath下的一个文件
    'filesize':None,
}

file_path = os.path.join(head['filepath'],head['filename'])
filesize = os.path.getsize(file_path)
head['filesize'] = filesize
# json_head = json.dumps(head,ensure_ascii=False) #字典转换成字符串
json_head = json.dumps(head) #字典转换成字符串
bytes_head = json_head.encode('utf-8') #字符串转换成bytes类型
print(json_head)
print(bytes_head)

#计算head的长度，因为接收端先接收我们自己定制的报头，对吧
head_len = len(bytes_head) #报头长度
pack_len = struct.pack('i',head_len)
print(head_len)
print(pack_len)
sk.send(pack_len) #先发送报头长度
sk.send(bytes_head) #再发送bytes类型的报头

#即便是视频文件，也是可以按行来读取的，也可以readline，也可以for循环，但是读取出来的数据大小就不固定了，影响效率，有可
#所有我们可以用read，设定一个一次读取内容的大小，一边读一边发，一边收一边写
with open(file_path,'rb') as f:
    while filesize:
        if filesize >= buffer: #>=是因为如果刚好等于的情况出现也是可以的。
            content = f.read(buffer) #每次读取出来的内容
            sk.send(content)
            filesize -= buffer #每次减去读取的大小
        else: #那么说明剩余的不够一次读取的大小了，那么只要把剩下的读取出来发送过去就行了
            content = f.read(filesize)
            sk.send(content)
            break

sk.close()

```



FTP上传下载文件的代码（升级版）（注：咱们学完网络编程就留FTP作业，这个代码可以参考，当你用函数的方式写完之后，再用面向对象进行改版却没有思路的时候再来看，别骗自己昂~~）



```

import socket
import struct
import json
import subprocess
import os

class MYTCPServer:
    address_family = socket.AF_INET

    socket_type = socket.SOCK_STREAM

    allow_reuse_address = False

    max_packet_size = 8192

```



```

coding='utf-8'

request_queue_size = 5

server_dir='file_upload'

def __init__(self, server_address, bind_and_activate=True):
    """Constructor. May be extended, do not override."""
    self.server_address=server_address
    self.socket = socket.socket(self.address_family,
                                self.socket_type)
    if bind_and_activate:
        try:
            self.server_bind()
            self.server_activate()
        except:
            self.server_close()
            raise

def server_bind(self):
    """Called by constructor to bind the socket.
    """
    if self.allow_reuse_address:
        self.socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    self.socket.bind(self.server_address)
    self.server_address = self.socket.getsockname()

def server_activate(self):
    """Called by constructor to activate the server.
    """
    self.socket.listen(self.request_queue_size)

def server_close(self):
    """Called to clean-up the server.
    """
    self.socket.close()

def get_request(self):
    """Get the request and client address from the socket.
    """
    return self.socket.accept()

def close_request(self, request):
    """Called to clean up an individual request."""
    request.close()

def run(self):
    while True:
        self.conn,self.client_addr=self.get_request()
        print('from client ',self.client_addr)
        while True:
            try:
                head_struct = self.conn.recv(4)
                if not head_struct:break

                head_len = struct.unpack('i', head_struct)[0]
                head_json = self.conn.recv(head_len).decode(self.coding)
                head_dic = json.loads(head_json)

                print(head_dic)
                #head_dic={'cmd': 'put', 'filename': 'a.txt', 'filesize': 123123}

```

```

        cmd=head_dic['cmd']
        if hasattr(self,cmd):
            func=getattr(self,cmd)
            func(head_dic)
    except Exception:
        break

def put(self,args):
    file_path=os.path.normpath(os.path.join(
        self.server_dir,
        args['filename']
    ))

    filesize=args['filesize']
    recv_size=0
    print('---->',file_path)
    with open(file_path,'wb') as f:
        while recv_size < filesize:
            recv_data=self.conn.recv(self.max_packet_size)
            f.write(recv_data)
            recv_size+=len(recv_data)
            print('recvsize:%s filesize:%s'%(recv_size,filesize))

tcpserver1=MYTCPServer(('127.0.0.1',8080))

tcpserver1.run()

```



```

import socket
import struct
import json
import os

class MYTCPClient:
    address_family = socket.AF_INET

    socket_type = socket.SOCK_STREAM

    allow_reuse_address = False

    max_packet_size = 8192

    coding='utf-8'

    request_queue_size = 5

    def __init__(self, server_address, connect=True):
        self.server_address=server_address
        self.socket = socket.socket(self.address_family,
                                    self.socket_type)

        if connect:
            try:
                self.client_connect()
            except:
                self.client_close()

```

```

        raise

def client_connect(self):
    self.socket.connect(self.server_address)

def client_close(self):
    self.socket.close()

def run(self):
    while True:
        inp=input(">>: ").strip()
        if not inp:continue
        l=inp.split()
        cmd=l[0]
        if hasattr(self,cmd):
            func=getattr(self,cmd)
            func(l)

def put(self,args):
    cmd=args[0]
    filename=args[1]
    if not os.path.isfile(filename):
        print('file:%s is not exists' %filename)
        return
    else:
        filesize=os.path.getsize(filename)

    head_dic={'cmd':cmd,'filename':os.path.basename(filename),'filesize':filesize}
    print(head_dic)
    head_json=json.dumps(head_dic)
    head_json_bytes=bytes(head_json,encoding=self.coding)

    head_struct=struct.pack('i',len(head_json_bytes))
    self.socket.send(head_struct)
    self.socket.send(head_json_bytes)
    send_size=0
    with open(filename,'rb') as f:
        for line in f:
            self.socket.send(line)
            send_size+=len(line)
            print(send_size)
        else:
            print('upload successful')

client=MYTCPClient(('127.0.0.1',8080))

client.run()

```



ok~今天的内容就到这里，大家别着急，稳扎稳打，把上面学习的这些内容在好好理解理解，写写代码练习练习~~~

=====

十一 验证客户端的连接合法性

首先，我们来探讨一下，什么叫验证合法性，举个例子：有一天，我开了一个socket服务端，只想让咱们这个班的同学使用，但是有一天，隔壁班的同学过来问了一下我开的这个服务端的ip和端口，然后他是不是就可以去连接我了啊，那怎么办，我是不是不想让他连接我啊，我需要验证一下你的身份，这就是验证连接的合法性，再举个例子，就像我们上面说的你的windows系统是不是连接微软的时间服务器来获取时间的啊，你的mac能到人家微软去获取时间吗，你愿

意，人家微软还不愿意呢，对吧，那这时候，你每次连接我来获取时间的时候，我是不是就要验证你的身份啊，也就是你要带着你的系统信息，我要判断你是不是我微软的windows，对吧，如果是mac，我是不是不让你连啊，这就是连接合法性。如果验证你的连接是合法的，那么如果我还要对你的身份进行验证的需求，也就是要验证用户名和密码，那么我们还需要进行身份认证。连接认证>>身份认证>>ok你可以玩了。

好大致描述相信大家基本理解了，如果这还没有理解，那么同学，我要哭晕在厕所了。

如果你想在分布式系统中实现一个简单的客户端链接认证功能，又不像SSL那么复杂，那么利用hmac+加盐的方式来实现，直接看代码！（SSL，我们都

□



```
from socket import *
import hmac,os

secret_key=b'Jedan has a big key!'
def conn_auth(conn):
    """
    认证客户端链接
    :param conn:
    :return:
    """
    print('开始验证新链接的合法性')
    msg=os.urandom(32)#生成一个32字节的随机字符串
    conn.sendall(msg)
    h=hmac.new(secret_key,msg)
    digest=h.digest()
    response=conn.recv(len(digest))
    return hmac.compare_digest(response,digest)

def data_handler(conn,bufsize=1024):
    if not conn_auth(conn):
        print('该链接不合法,关闭')
        conn.close()
        return
    print('链接合法,开始通信')
    while True:
        data=conn.recv(bufsize)
        if not data:break
        conn.sendall(data.upper())

def server_handler(ip_port,bufsize,backlog=5):
    """
    只处理链接
    :param ip_port:
    :return:
    """
    tcp_socket_server=socket(AF_INET,SOCK_STREAM)
    tcp_socket_server.bind(ip_port)
    tcp_socket_server.listen(backlog)
    while True:
        conn,addr=tcp_socket_server.accept()
        print('新连接[%s:%s]' %(addr[0],addr[1]))
        data_handler(conn,bufsize)

if __name__ == '__main__':
    ip_port=('127.0.0.1',9999)
    bufsize=1024
    server_handler(ip_port,bufsize)
```



```
from socket import *
import hmac,os

secret_key=b'Jedan has a big key! '
def conn_auth(conn):
    """
    验证客户端到服务器的链接
    :param conn:
    :return:
    """
    msg=conn.recv(32)
    h=hmac.new(secret_key,msg)
    digest=h.digest()
    conn.sendall(digest)

def client_handler(ip_port,bufsize=1024):
    tcp_socket_client=socket(AF_INET,SOCK_STREAM)
    tcp_socket_client.connect(ip_port)

    conn_auth(tcp_socket_client)

    while True:
        data=input('>>: ').strip()
        if not data:continue
        if data == 'quit':break

        tcp_socket_client.sendall(data.encode('utf-8'))
        response=tcp_socket_client.recv(bufsize)
        print(response.decode('utf-8'))
    tcp_socket_client.close()

if __name__ == '__main__':
    ip_port=('127.0.0.1',9999)
    bufsize=1024
    client_handler(ip_port,bufsize)
```



介绍代码中使用的两个方法：

1、os.urandom(n)

其中os.urandom(n) 是一种bytes类型的随机生成n个字节字符串的方法，而且每次生成的值都不相同。再加上md5等加密的处理，就能够生成内容不同长度相同的字符串了。



os.urandom(n)函数在python官方文档中做出了这样的解释

函数定位： Return a string of n random bytes suitable for cryptographic use.

意思就是，返回一个有n个byte那么长的一个string，然后很适用于加密。

然后这个函数，在文档中，被归结于os这个库的Miscellaneous Functions，意思是不同种类的函数（也可以说是混种函数）

原因是： This function returns random bytes from an OS-specific randomness source.（函数返回的随机字节是根据不同的操



使用方法：

```
import os
from hashlib import md5

for i in range(10):
    print md5(os.urandom(24)).hexdigest()
```

2、hmac：我们完全可以用hashlib来实现，但是学个新的吗，没什么不好的，这个操作更方便一些。

Python自带的hmac模块实现了标准的Hmac算法，我们首先需要准备待计算的原始消息message，随机key，哈希算法，这里采用MD5，使用hmac的代码如下：

```
import hmac
message = b'Hello world'
key = b'secret'
h = hmac.new(key,message,digestmod='MD5')
print(h.hexdigest())
```

比较两个密文是否相同，可以用hmac.compare_digest(密文、密文)，然会True或者False。

可见使用hmac和普通hash算法非常类似。hmac输出的长度和原始哈希算法的长度一致。**需要注意传入的key和message都是 bytes 类型，str 类型需要首先编码为 bytes。**



```
def hmac_md5(key, s):
    return hmac.new(key.encode('utf-8'), s.encode('utf-8'), 'MD5').hexdigest()

class User(object):
    def __init__(self, username, password):
        self.username = username
        self.key = ''.join([chr(random.randint(48, 122)) for i in range(20)])
        self.password = hmac_md5(self.key, password)
```



十二 socketserver模块实现并发

为什么要讲socketserver？我们之前写的tcp协议的socket是不是一次只能和一个客户端通信，如果用socketserver可以实现和多个客户端通信。它是在socket的基础上进行了一层封装，也就是说底层还是调用的socket，在py2.7里面叫做SocketServer也就是大写了两个S，在py3里面就小写了。后面我们要写的FTP作业，需要用它来实现并发，也就是同时可以和多个客户端进行通信，多个人可以同时进行上传下载等。

那么我们先看socketserver怎么用呢，然后在分析，先看下面的代码



```
import socketserver          #1、引入模块
class MyServer(socketserver.BaseRequestHandler): #2、自己写一个类，类名自己随便定义，然后继承socketserver这个模块里面

    def handle(self):         #3、写一个handle方法，必须叫这个名字
        #self.request          #6、self.request 相当于一个conn

        self.request.recv(1024)    #7、收消息
        msg = '亲，学会了吗'
        self.request.send(bytes(msg,encoding='utf-8')) #8、发消息

        self.request.close()       #9、关闭连接
        # 拿到了我们对每个客户端的管道，那么我们自己在这个方法里面的就写我们接收消息发送消息的逻辑就可以了
        pass
if __name__ == '__main__':
```

#thread 线程，现在只需要简单理解线程，别着急，后面很快就会讲到啦，看下面的图
server = socketserver.ThreadingTCPServer(('127.0.0.1',8090),MyServer)#4、使用socketserver的ThreadingTCPServer这个
server.serve_forever() #5、使用我们上面这个类的对象来执行serve_forever()方法，他的作用就是说，我的服务一

#注意:

#有socketserver 那么有socketclient的吗?

#当然不会有，我要作为客户去访问京东的时候，京东帮我也客户端了吗，客户端是不是在我们自己的电脑啊，并且socketserver对客户



ThreadingTCPServer，多线程，简单解释：看图

wuchao

socket_server_s.py
PyCharm

假如这个红色的框是我们的程序，那么有的程序里面可以有多个线程

里面的蓝线为线程
server #1、引入模块
class MyServer(socketserver.BaseRequestHandler):
 一个类，类名自己随便定义，然后继承socketserver这个模块里面的BaseRequestHandler
 def handle(self): #3、写一个handle方法，必须叫这个名字
 线程，现在只需要简单理解线程，别着急，后面很快就会讲到啦，简单下面
server = ThreadingTCPServer()

那么我们的一个程序里面有了多个线程，每个线程去帮你处理一个socket连接，也就是处理一个对应的客户端，那么我们的程序就可以同时和多个客户端进行沟通了，就简单理解到这里，好吗？

通过上面的代码，我们来分析socket的源码：（大家还记得面向对象的继承吗，来，实战的时候来啦）



在整个socketserver这个模块中，其实就干了两件事情：1、一个是循环建立链接的部分，每个客户链接都可以连接成功 2、一个通讯
还记得面向对象的继承吗？来，大家自己尝试着看看源码：

查找属性的顺序：ThreadingTCPServer->ThreadingMixIn->TCPServer->BaseServer

实例化得到server，先找ThreadMixIn中的__init__方法，发现没有init方法，然后找类ThreadingTCPServer的__init__，在TCPServer中找server下的serve_forever，在BaseServer中找到，进而执行self._handle_request_noblock()，该方法同样是在BaseServer中执行self._handle_request_noblock()进而执行request, client_address = self.get_request()（就是TCPServer中的self.socket.accept()）在ThreadingMixIn中找到process_request，开启多线程应对并发，进而执行process_request_thread，执行self.finish_request(request, client_address)上述四部分完成了链接循环，本部分开始进入处理通讯部分，在BaseServer中找到finish_request，触发我们自己定义的类的实例化，去源码分析总结：

基于tcp的socketserver我们自己定义的类中的

self.server即套接字对象
self.request即一个链接
self.client_address即客户端地址

基于udp的socketserver我们自己定义的类中的

self.request是一个元组（第一个元素是客户端发来的数据，第二部分是服务端的udp套接字对象），如(b'adsf', <socket.socket object at 0x...>)
self.client_address即客户端地址



一个完整的socketserver代码示例：

服务端代码示例：



```
import socketserver

class Myserver(socketserver.BaseRequestHandler):
    def handle(self):
        self.data = self.request.recv(1024).strip()
        print("{} wrote:".format(self.client_address[0]))
        print(self.data)
        self.request.sendall(self.data.upper())

if __name__ == "__main__":
    HOST, PORT = "127.0.0.1", 9999

    # 设置allow_reuse_address允许服务器重用地址
    socketserver.TCPServer.allow_reuse_address = True
    # 创建一个server, 将服务地址绑定到127.0.0.1:9999
    #server = socketserver.TCPServer((HOST, PORT), Myserver)
    server = socketserver.ThreadingTCPServer((HOST, PORT), Myserver)
    # 让server永远运行下去, 除非强制停止程序
    server.serve_forever()
```



客户端代码示例：



```
import socket

HOST, PORT = "127.0.0.1", 9999
data = "hello"

# 创建一个socket链接, SOCK_STREAM代表使用TCP协议
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as sock:
    sock.connect((HOST, PORT)) # 链接到客户端
    sock.sendall(bytes(data + "\n", "utf-8")) # 向服务端发送数据
    received = str(sock.recv(1024), "utf-8") # 从服务端接收数据

print("Sent: {}".format(data))
print("Received: {}".format(received))
```



十三 网络编程的作业

好了同学们，到了这儿，我们的网络编程socket就讲完了，大致就是这些内容，给大家留个作业：（你的努力的结果你自己是看的到的~！）

加粗的是必须要做的，倾斜的是比较有难度的，大家别放松呀。

1. **多用户同时登陆**
2. **用户登陆，加密认证**
3. **上传/下载文件，保证文件一致性**
4. **传输过程中现实进度条**
5. 不同用户家目录不同，且只能访问自己的家目录
6. 对用户进行磁盘配额、不同用户配额可不同
7. 用户登陆server后，可在家目录权限下切换子目录
8. 查看当前目录下文件，新建文件夹

9. 删除文件和空文件夹
10. 充分使用面向对象知识
11. 支持断点续传

简单分析一下实现方式：

1. 字符串操作以及打印 —— 实现上传下载的进度条功能



```
一、
import sys
import time
for i in range(50):
    sys.stdout.write('>')
    sys.stdout.flush()
    time.sleep(0.2)

二、
#总共接收到的大小和总文件大小的比值：
#all_size_len表示当前总共接受的多长的数据，是累计的
#file_size表示文件的总大小
per_cent = round(all_size_len/file_size,2) #将比值做成两位数的小数
#通过\r来实现同一行打印，每次打印都回到行首打印
print('\r'+ '%s%%'%(str(int(per_cent*100))) + '*'*(int(per_cent*100)),end='') #由于float类型的数据没法通过%s来进行字符
```



- 2.socketserver —— 实现ftp server端和client端的交互

- 3.struct模块 —— 自定义报头解决文件上传下载过程中的粘包问题

- 4.hashlib或者hmac模块 —— 实现文件的一致性校验和用户密文登录

- 5.os模块 —— 实现目录的切换及查看文件文件夹等功能

- 6.文件操作 —— 完成上传下载文件及断点续传等功能

看一下流程图：

