[python之socket编程](https://www.cnblogs.com/charles1ee/p/6768326.html)

**客户端/服务器架构**

C/S架构（B/S架构 (浏览器/服务器)）

C/S架构与socket的关系：

学习socket就是为了完成C/S架构的开发

**OSI七层**

一个完整的计算机系统是由硬件，操作系统，应用软件三者组成，具备了这三个条件，一台计算机可以自己玩那么没啥问题。

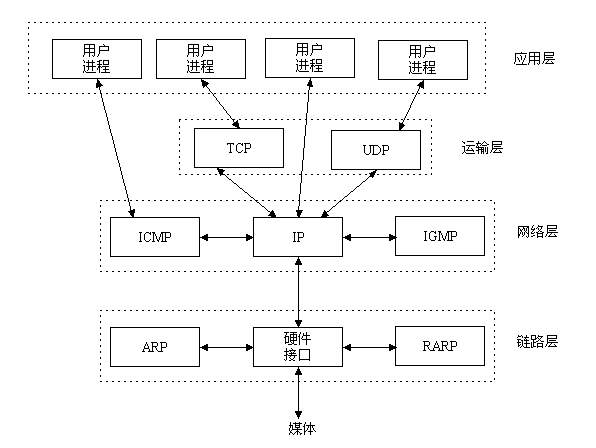
如果要跟别人一起玩，那么就需要上网了。互联网核心就是一堆协议组成，协议就是标准。全世界人通信的标准是英语，如果把计算机比喻成人，那么互联网协议就是计算机界的英语了。所有的计算机都学会了互联网协议，那所有的计算机都就可以按照统一的标准去收发信息从而完成通信了

详情：[点击这里](http://www.cnblogs.com/charles1ee/p/6595926.html" \t "_blank)

看如下图：

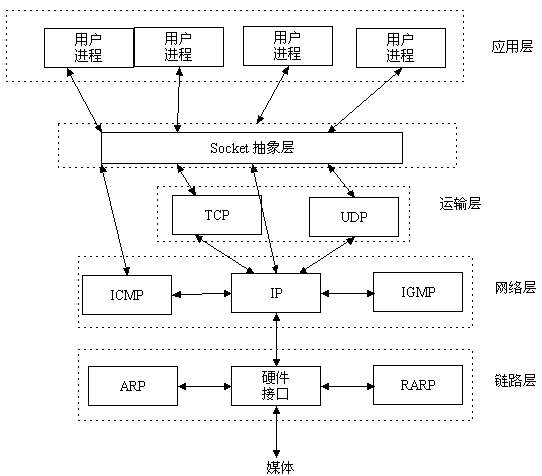
首先我们都属于应用程序开发工程师

应用层需要遵循TCP/UDP标准，那么我们在写应用软件的时候，需要先把TCP/UDP标准研究明白了才能写。但是TCP/UDP协议第一，太过于古老。第二，太过于庞大。对于这种已经很成熟的技术，我们没有必要在花费很长的数据去进行研究，找一个简单点的协议就好了



　　　　　　　　　　　　　　　　　　　　　图一

**socket层**



　　　　　　　　　　　　　　　　　　　图二

简单一点的就是在TCP/UDP层上面又封装了一个socket层

socket层的作用就是将TCP/UDP封装到socket接口后面。

所以，我们无需深入了解TCP/UDP协议，socket已经帮我们封装好了。我们只要遵循socket的规定去编程，写出的程序自然而然就遵循了TCP/UDP标准

**套接字发展史及分类**

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

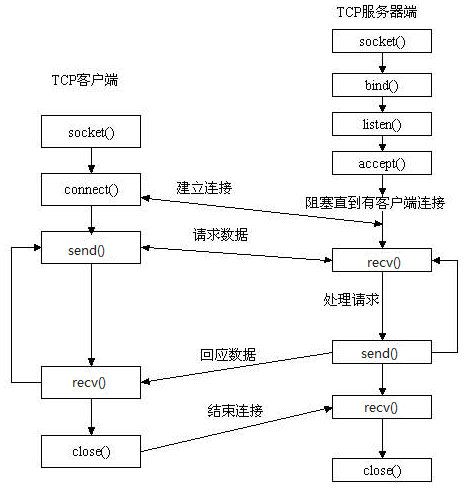
***基于网络类型的套接字家族***

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，所以大部分时候我么只使用AF\_INET)

**套接字工作流程**

       一个生活中的场景。你要打电话给一个朋友，先拨号，朋友听到电话铃声后提起电话，这时你和你的朋友就建立起了连接，就可以讲话了。等交流结束，挂断电话结束此次交谈。    生活中的场景就解释了这工作原理，也许TCP/IP协议族就是诞生于生活中，这也不一定



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端的连接就建立了。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

**基于TCP的socket编程**

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket

phone=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM) #获取手机

#AF\_INET:基于网络协议通信。

#SOCK\_STREAM:流。tcp协议。udp协议是SOCK\_DGRAM

phone.bind(('127.0.0.1',8080)) #绑定手机卡('IP',端口)要以元组的形式

phone.listen(5) #开机 listen(5)后面会说

#上面的操作就已经开机了。

conn,clinet\_addr\_port=phone.accept() #等待别人打电话。电话进来是一个元组的形式，有两个元素。我们把它拿出来

print('====>',conn,clinet\_addr\_port)

#conn:三次握手的连接，就是电话线

#clinet\_addr\_port:客户端的地址和端口

clinet\_data=conn.recv(1024) #基于拿到的线路进行发消息。也可以接消息

#1024代表收1024个字节

print('客户端发来的数据是:%s'%clinet\_data)

conn.send(clinet\_data.upper())

#收到消息后返回客户端大写消息

conn.close()

#通话结束后，断开电话线

phone.close()

#关闭手机

#拿到手机——> 绑卡——> 开机——> 等待别人给我打电话——> 收信息——>返回信息 ——>挂断电话——>关机

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket

phone=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)

#客户端也是一样，需要个手机

#但是这个例子有些不完美，不完美的地方就在于客户端不需要绑定手机卡

phone.connect(('127.0.0.1',8080)) #客户端直接连接

#拨通电话，需要以一个元组的形式。对应服务端的等待连接accept()

phone.send('hello'.encode('utf-8')) #发送信息到服务端

# phone.send(bytes('hello',encoding='utf-8'))#与上面是一样的

#基于网络传输，网络传输只能传输二进制

data=phone.recv(1024) #接受数据。

print(data)

phone.close()

#关机

#拿到手机——> 连接服务端——>发送信息——>接收信息——关机

#启动的时候有先启动服务器端，然后客户端才可以尽量三次握手进行连接

[复制代码](javascript:void(0);)

上面的是一个简易版的。一次通信流程。但是在真实情况中是一个循环的过程，所以请看下面代码。

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket

phone=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)

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

phone.listen(5)

#开机

conn,clinet\_addr\_port=phone.accept()

print('====>',conn,clinet\_addr\_port)

#等待

while True:

#通信循环

clinet\_data=conn.recv(1024)

print('客户端发来的数据是:%s'%clinet\_data)

conn.send(clinet\_data.upper())

#收发

conn.close()

phone.close()

#关闭

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket

phone=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)

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

#开机连接

while True:

#同样在客户端也是要进行通信循环，如果不这么写，那么客户端发完一次信息后服务端将进入死循环状态

msg=input('>>:')

phone.send(msg.encode('utf-8'))

data=phone.recv(1024)

print(data)

#收发信息

phone.close()

#关闭

[复制代码](javascript:void(0);)

虽然已经可以进行多次通话，但是还是有瑕疵。如果客户端发起非正常关闭操作。那么服务端就会报错。解决方法就是用异常处理。

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket

phone=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)

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

phone.listen(5)

#开机

conn,clinet\_addr\_port=phone.accept()

print('====>',conn,clinet\_addr\_port)

#等待

while True:

try:

#通信循环

clinet\_data=conn.recv(1024)

print('客户端发来的数据是:%s'%clinet\_data)

conn.send(clinet\_data.upper())

#收发

except ConnectionResetError:

break

conn.close()

phone.close()

#关闭

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket

phone=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)

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

phone.listen(5)

#开机

conn,clinet\_addr\_port=phone.accept()

print('====>',conn,clinet\_addr\_port)

#等待

while True:

try:

#通信循环

clinet\_data=conn.recv(1024)

if not clinet\_data: break # 客户端正常关闭后防止服务端进入死循环

print('客户端发来的数据是:%s'%clinet\_data)

conn.send(clinet\_data.upper())

#收发

except ConnectionResetError:

break

conn.close()

phone.close()

#关闭

[复制代码](javascript:void(0);)

上面的通信属于单客户端通信。就是说客户端通信结束后，服务端也跟着结束。如果是多用户呢？

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket

phone=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)

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

phone.listen(5)

#开机

while True:

#连接循环

conn,clinet\_addr\_port=phone.accept()

print('====>',conn,clinet\_addr\_port)

#等待

while True:

try:

#通信循环

clinet\_data=conn.recv(1024)

if not clinet\_data: break # 客户端正常关闭后防止服务端进入死循环

print('客户端发来的数据是:%s'%clinet\_data)

conn.send(clinet\_data.upper())

#收发

except ConnectionResetError:

break

conn.close()

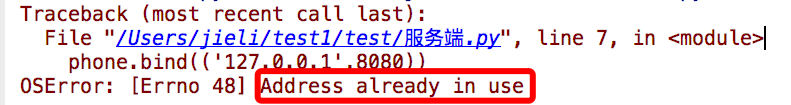
phone.close()

#关闭

[复制代码](javascript:void(0);)

客户端的代码都是相同的。所以copy一个客户端代码为客户端2.通过测试可以发现，服务端启动后，再启动两个客户端可以可以发现服务端一次只能处理一个连接，另外一个客户端属于等待状态。只有第一个客户端通信结束后，服务端才会接着处理下一个请求。所以并没有实现并发的效果，后面我会说到

 可能有很多人会遇到这种问题



这个是由于你的服务端仍然存在四次挥手的time\_wait状态在占用地址（如果不懂，请深入研究1.tcp三次握手，四次挥手 2.syn洪水攻击 3.服务器高并发情况下会有大量的time\_wait状态的优化方法）

两种解决方法

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

#加入一条socket配置，重用ip和端口

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

phone.setsockopt(SOL\_SOCKET,SO\_REUSEADDR,1) #就是它，在bind前加

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

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

发现系统存在大量TIME\_WAIT状态的连接，通过调整linux内核参数解决，

vi /etc/sysctl.conf

编辑文件，加入以下内容：

net.ipv4.tcp\_syncookies = 1

net.ipv4.tcp\_tw\_reuse = 1

net.ipv4.tcp\_tw\_recycle = 1

net.ipv4.tcp\_fin\_timeout = 30

然后执行 /sbin/sysctl -p 让参数生效。

net.ipv4.tcp\_syncookies = 1 表示开启SYN Cookies。当出现SYN等待队列溢出时，启用cookies来处理，可防范少量SYN攻击，默认为0，表示关闭；

net.ipv4.tcp\_tw\_reuse = 1 表示开启重用。允许将TIME-WAIT sockets重新用于新的TCP连接，默认为0，表示关闭；

net.ipv4.tcp\_tw\_recycle = 1 表示开启TCP连接中TIME-WAIT sockets的快速回收，默认为0，表示关闭。

net.ipv4.tcp\_fin\_timeout 修改系統默认的 TIMEOUT 时间

方法二

[复制代码](javascript:void(0);)

不知道各位有没有遇到这种问题，就是在客户端那边输入回车的时候，发现卡顿了

为什么会卡住呢？是卡在客户端还是卡在服务端呢？

基于我们上边的服务端例子，不难理解有几处阻塞的地方，一处是等待别人来连接。如果没有人来连接，服务端会一直卡住。但是跟上边的问题没有关系。只要客户端连接上，服务端就会收到。客户端服务端建立连接进入通信循环

客户端输入回车后，服务端还处于通信循环中，客户端也处于通信循环中。那么到底是哪里卡住呢？请看分析

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket

phone=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)

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

#开机连接

while True:

msg=input('>>:') #输入回车

phone.send(msg.encode('utf-8')) #客户端发送

print('客户端已发送') #打印测试是否因为上面代码卡住

data=phone.recv(1024)

print('客户端以接收') #打印测试客户端是否收到服务端回的信息

print(data)

#收发信息

phone.close()

#关闭

#代码执行结果

>>:as

客户端已发送 #上面测试代码通信正常

客户端以接收

b'AS'

>>:

客户端已发送 #输入回车发现发送无法接收(只有收不到东西就会卡)。那么就查看服务端

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket

phone=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)

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

phone.listen(5)

#开机

while True:

#连接循环(包括通信循环)

conn,clinet\_addr\_port=phone.accept()

print('====>',conn,clinet\_addr\_port)

#等待

while True:

try:

#通信循环

clinet\_data=conn.recv(1024)

print('服务端接收客户端信息') #打印测试服务端是否收到客户端发来的信息

if not clinet\_data: break

print('客户端发来的数据是:%s'%clinet\_data)

conn.send(clinet\_data.upper())

#收发

except ConnectionResetError:

break

conn.close()

phone.close()

#关闭

#服务端打印，发现服务端没有收到，所以根本没法回

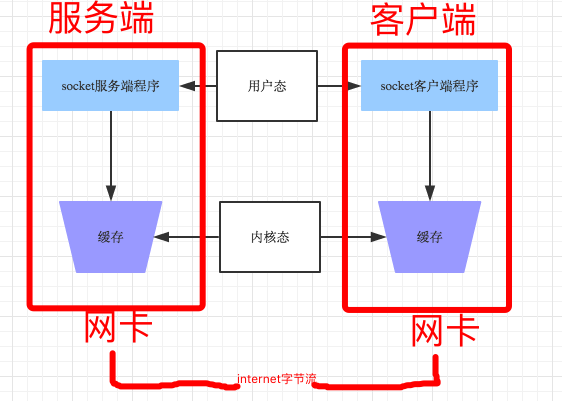
====> <socket.socket fd=224, family=AddressFamily.AF\_INET, type=SocketKind.SOCK\_STREAM, proto=0, laddr=('127.0.0.1', 8080), raddr=('127.0.0.1', 15271)> ('127.0.0.1', 15271)

[复制代码](javascript:void(0);)

由于我们是应用开发程序员，我们写的软件一定要运行中操作系统之上。软件要发包，就要要过硬件。所以发数据的时候就要调用底层的网卡。这时候就涉及到一个软件要使用硬件的操作了，但是软件无法使用硬件，所以对硬件的操作要转给操作系统。这个操作就是软件的用户态切换到内核态的过程。只要是软件就都工作在用户态。用户态程序不能够直接调用硬件。所以只有将我们的软件切成内核态，变成操作系统，然后调用硬件，把包发出去。

步骤:软件——>操作系统——>硬件

那么就明了了，软件往外发包实际上都发给了操作系统。send，并没有发给对端，而是发给了自己的操作系统。recv也不是上对端去取，而是向自己的操作系统去取



　　那么这个问题就解决了，客户端往自己的缓存中发了一个回车，相当于丢个操作系统一个空，操作系统在检查这个缓存的时候，发现缓存里面是空的，没有东西，所以操作系统也就不会往外发包了，服务端也就不会回包，客户端再向缓存取数据时，发现里面没有数据，拿不到所以就一直卡

**问题解决**

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket

phone=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)

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

#开机连接

while True:

msg=input('>>:')

if not msg:continue #进行if判断，如果输入为空则继续输入

phone.send(msg.encode('utf-8'))

data=phone.recv(1024)

print(data)

#收发信息

phone.close()

#关闭

[复制代码](javascript:void(0);)

**基于UDP的socket编程**

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket #导入模块

s=socket.socket(socket.AF\_INET,socket.SOCK\_DGRAM)#使用udp

s.bind(('127.0.0.1',8080))

# s.listen(5)

#对于udp来说，不需要listen().因为udp是无连接的

#s.accept()

#同样udp也不需要accept(),无连接状态，所以不需要等待连接

#所以直接收包发包即可

data=s.recvfrom(1024) #与tcp不同，upd接收请求是recvfrom

# data,client\_addr\_port=s.recvfrom(1024)

print(data)

s.close()

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket

c=socket.socket(socket.AF\_INET,socket.SOCK\_DGRAM)

c.sendto('hello'.encode('utf-8'),('127.0.0.1',8080)) #由于udp无连接，所以直接发包

#使用sendto应给两个参数一个是bytes'要发的数据',另一个是服务端地址(ip地址与端口)。以元组的形式

c.close()

[复制代码](javascript:void(0);)

完善一

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket #导入模块

s=socket.socket(socket.AF\_INET,socket.SOCK\_DGRAM)#使用udp

s.bind(('127.0.0.1',8080))

data,client\_addr\_port=s.recvfrom(1024)

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

#基于发送中文需要进行utf-8解码操作

s.sendto('你也好啊！'.encode('utf-8'),client\_addr\_port)#服务端回包的时候也是sendto

#两个元素，一个是回包数据，另一个是客户端的地址，客户端地址就是recvfrom接收的元组中第二个元素，也就是client\_addr\_port

s.close()

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket

c=socket.socket(socket.AF\_INET,socket.SOCK\_DGRAM)

c.sendto('你好啊'.encode('utf-8'),('127.0.0.1',8080)) #由于udp无连接，所以直接发包

#使用sendto应给两个参数一个是bytes'要发的数据',另一个是服务端地址(ip地址与端口)。以元组的形式

data,server\_addr\_port=c.recvfrom(1024)#同样，在客户端也是要接收两个元素，一个是数据一个是服务端的地址

print(data.decode('utf-8'))#解码打印

c.close()

[复制代码](javascript:void(0);)

可以发现上面就类似于聊天一样。由于udp是无连接的所以不需要有连接循环了，但是要有通信循环

完善二

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket #导入模块

s=socket.socket(socket.AF\_INET,socket.SOCK\_DGRAM)

s.bind(('127.0.0.1',8080))

#开机

while True:

#通信循环就是收发的过程，所以将recvfrom与sendto放到循环里

data,client\_addr\_port=s.recvfrom(1024)

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

#收信息

msg=input('>>:') #让用户输入

s.sendto(msg.encode('utf-8'),client\_addr\_port)

#发信息

s.close()

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket

c=socket.socket(socket.AF\_INET,socket.SOCK\_DGRAM)

# 开机

while True:

#与服务端相同

msg=input('>>:')#让用户输入

c.sendto(msg.encode('utf-8'),('127.0.0.1',8080))

# 发信息

data,server\_addr\_port=c.recvfrom(1024)

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

# 收信息

c.close()

[复制代码](javascript:void(0);)

UDP是否可以支持多客户端

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket #导入模块

s=socket.socket(socket.AF\_INET,socket.SOCK\_DGRAM)

s.bind(('127.0.0.1',8080))

#开机

while True:

data,client\_addr\_port=s.recvfrom(1024)

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

#收信息

# msg=input('>>:')

s.sendto(data,client\_addr\_port)

#发信息

s.close()

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket

c=socket.socket(socket.AF\_INET,socket.SOCK\_DGRAM)

# 开机

while True:

msg=input('>>:')

c.sendto(msg.encode('utf-8'),('127.0.0.1',8080))

# 发信息

data,server\_addr\_port=c.recvfrom(1024)

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

# 收信息

c.close()

#客户端代码都一样，可以copy这个代码为客户端二

[复制代码](javascript:void(0);)

通过测试可以看出来，udp貌似实现了并发，其实并不然。tcp是基于连接的，处理完一个再连接下一个连接，而udp是没有连接的，没有连接的效果就是，来一个我收一个再返回，下一个再来，还是一样。因为没有连接所以能达到这种效果

**远程执行命令**

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket

import subprocess #使用这个模块可以调用操作系统的命令

s=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)

s.bind(('127.0.0.1',8080))

s.listen(5)

while True:

#连接循环

conn,addr=s.accept()

print('新的客户端连接',addr)

while True:

try:

# 通信循环

cmd = conn.recv(1024)

cmd=cmd.decode('utf-8')

if not cmd: break

print('客户端发来的命令是:%s' % cmd)

cmd\_res=subprocess.Popen(cmd,shell=True,

stderr=subprocess.PIPE,

stdout=subprocess.PIPE,

)

#第一个参数执行这个命令，第二个参数以shell形式，第三个参数，如果有错误将错误丢到管道中，第四个参数，将输出结果丢到管道中

#PIPE这个管道就是给我们取拿结果，可能是对的也可能是错误的结果

err=cmd\_res.stderr.read() #读取cmd\_res里面是否有错误内容

if err: #如果err里面有值，那么就返回res

res=err

else:

res=cmd\_res.stdout.read()

conn.send(res) #像客户端返回结果

except ConnectionResetError:

break

conn.close()

s.close()

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket

clinet=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)

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

while True:

cmd=input('>>:')

if not cmd:continue

clinet.send(cmd.encode('utf-8'))

data=clinet.recv(1024)

print(data.decode('gbk')) #由于在windows上默认的字符编码为gbk所以解码的时候也是gbk格式

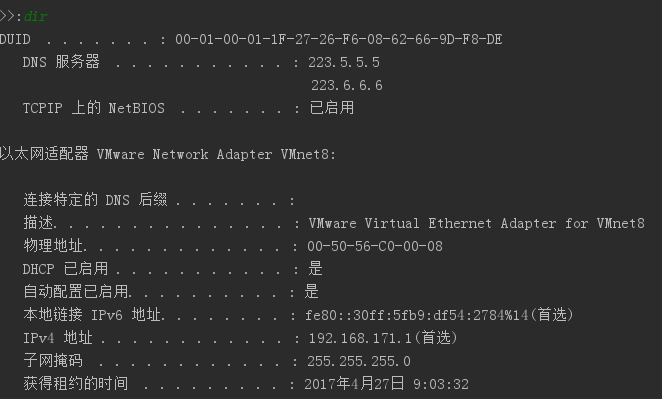
clinet.close()

[复制代码](javascript:void(0);)

上面的服务端也可是写在linux上，体现出一种跨平台性，前提要改好编码格式，还有IP地址哦

上面的代码已经可以基本实现功能，但是各位有没有发现问题？

我执行dir命令后，执行ipconfig /all 再次执行dir命令发现最后一次dir命令是下面那个样子，输出的还是ipconfig的命令

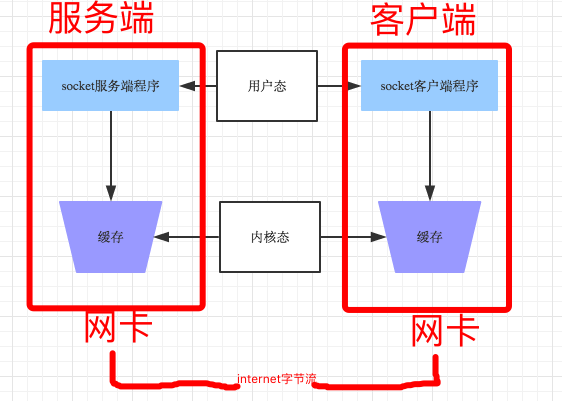


为什么会出现这种效果了，这就涉及到一个粘包的问题

**什么是粘包**

**只有TCP有粘包现象，UDP永远不会粘包**

**首先需要掌握一个socket收发消息的原理**



　　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。若连续几次需要send的数据都很少，通常TCP会根据优化[算法](http://lib.csdn.net/base/datastructure" \o "算法与数据结构知识库" \t "_blank)把这些数据合成一个TCP段后一次发送出去，这样接收方就收到了粘包数据。

　　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（客户端和服务器端）都要有一一成对的socket，因此，发送端为了将多个发往接收端的包，更有效的发到对方，使用了优化方法（Nagle算法），将多次间隔较小且数据量小的数据，合并成一个大的数据块，然后进行封包。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.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。**

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是基于数据报的，即便是你输入的是空内容（直接回车），那也不是空消息，udp协议会帮你封装上消息头**

粘包测试一

发生在客户端的粘包

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket

s=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)

s.bind(('127.0.0.1',8080))

s.listen(5)

conn,addr=s.accept()

data1=conn.recv(10)

data2=conn.recv(10)

print(data1)

print(data2)

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket

c=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)

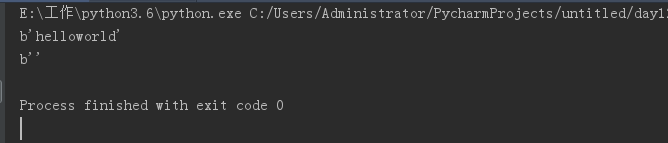
c.connect(('127.0.0.1',8080))

c.send('hello'.encode('utf-8'))

c.send('world'.encode('utf-8'))

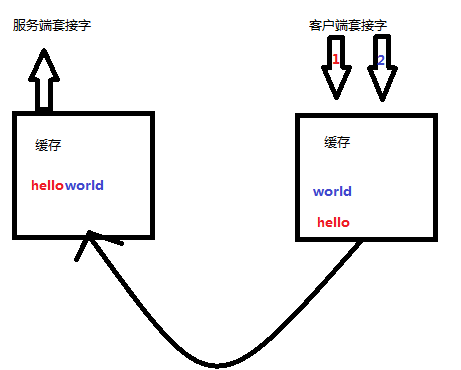
[复制代码](javascript:void(0);)

分别启动服务端与客户端，发现server端执行结果为下图



按理来说应该data1打印一部分，data2打印一部分。为什么data1都打印了？

原因在于clinet端，客户端send两次的结果并不是直接send到服务端，而是发给了自己的缓存，再由缓存发送到对端的缓存中，然后服务端的套接字再到自己的缓存中取出数据。可以看上面标红的TCP那段



粘包测试二

发生在服务端的粘包

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket

s=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)

s.bind(('127.0.0.1',8080))

s.listen(5)

conn,addr=s.accept()

data1=conn.recv(1) #只收一个，收不完。自己的缓存区里还有elloworld，

data2=conn.recv(100)

print(data1)

print(data2)

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket,time

c=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)

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

c.send('hello world'.encode('utf-8'))

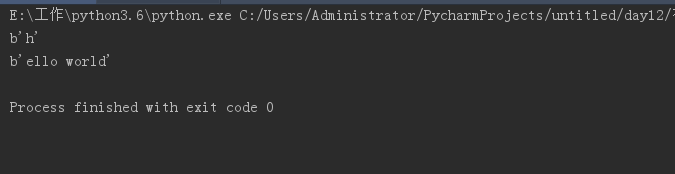
#发一条消息不会粘包

time.sleep(5)#设置间隔时间，防止在客户端粘包

c.send('nihao'.encode('utf-8'))

[复制代码](javascript:void(0);)

分别启动服务端与客户端，发现执行结果如下图。明明第二次recv的字节为100，可以把缓存区剩下的数据全部提取出来，为什么只打印了一个'ello world'



原因在于，服务端接收两次后程序已经走完。而客户端发完第一次数据后间隔了五秒又再次发包，这时服务端已经结束，客户端还在发包。所以服务端根本不会鸟它。

解决方法很简单，客户端睡五秒，那么我服务端也跟着睡五秒

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket

import time

s=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)

s.bind(('127.0.0.1',8080))

s.listen(5)

conn,addr=s.accept()

data1=conn.recv(1)

time.sleep(6)

data2=conn.recv(100)

print(data1)

print(data2)

[复制代码](javascript:void(0);)

TCP粘包，为什么TCP会粘包，就是因为TCP基于流(stream)，数据流工作的。数据流的特点是没有消息的边界，所以导致服务端根本不知道收多少个字节

**那么问题来了？怎么解决这种问题呢？上面的睡眠是在你知道的情况下，如果一切都是未知的情况那该如果解决呢？**

解决方法：还是用上边的远程执行命令代码来测试

比较LOW逼的方法

在真实数据返回之前，先len一下要返回的数据的长度，之后将长度先返回给客户端。客户端根据返回的大小，再进行修改取值

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket

import subprocess #使用这个模块可以调用操作系统的命令

import struct

s=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)

s.bind(('127.0.0.1',8080))

s.listen(5)

while True:

#连接循环

conn,addr=s.accept()

print('新的客户端连接',addr)

while True:

try:

# 通信循环

cmd = conn.recv(1024)

cmd=cmd.decode('utf-8')

if not cmd: break

print('客户端发来的命令是:%s' % cmd)

cmd\_res=subprocess.Popen(cmd,shell=True,

stderr=subprocess.PIPE,

stdout=subprocess.PIPE,

)

err=cmd\_res.stderr.read()

if err:

res=err

else:

res=cmd\_res.stdout.read()

print('命令执行结果长度',len(res))

# conn.send(str(len(res)).encode('utf-8'))#虽然这样就实现了，但是这个send与下面的send必然会粘到一块

#解决方法：将长度做成一个固定字节的beyts。使用struct模块

#导入模块后就不能用上面的方法send

conn.send(struct.pack('i',len(res))) #使用struct模块打包，i代表打包成整型，固定长度为4，客户端第一次接受4个字节再解包就知道真实数据要多少字节

conn.send(res) #发送真实数据

except ConnectionResetError:

break

conn.close()

s.close()

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket

import struct

clinet=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)

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

while True:

cmd=input('>>:')

if not cmd:continue

clinet.send(cmd.encode('utf-8'))

#第一阶段：收数据的长度

x=clinet.recv(4) #struct模块打包后固定长度就为4，所以接收4个字节那么就不会与后面的数据发生粘包

data\_len=struct.unpack('i',x)[0] #怎么打包就怎么解包，解包后是一个元组的形式，第一个元素就是真实数据的字节

#第二阶段：根据数据的长度，收真实数据

data=clinet.recv(data\_len)

print(data.decode('gbk'))

clinet.close()

[复制代码](javascript:void(0);)

上面的代码已经可以实现不粘包了，只不过比较low，后面还有更牛逼的方法，大家稍安勿躁。上面的代码是通过tcp实现的远程执行命令，那么我再用一个udp协议写一个基于udp的远程执行命令

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket

import subprocess #执行命令模块

import struct #打包模块,在udp中就不需要了

s=socket.socket(socket.AF\_INET,socket.SOCK\_DGRAM) #数据报形式，所有udp没有粘包

#创建基于UDP的套接字

s.bind(('127.0.0.1',8080))

while True:

cmd,addr=s.recvfrom(1024)

print('客户端连接信息是:',addr)

cmd=cmd.decode('utf-8')

print('客户端发来的命令是:',cmd)

cmd\_res=subprocess.Popen(cmd,shell=True,

stderr=subprocess.PIPE,

stdout=subprocess.PIPE)

err=cmd\_res.stderr.read()

if err:

res=err

else:

res=cmd\_res.stdout.read()

print(len(res)) #打印数据长度

s.sendto(res,addr)

s.close()

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket

c=socket.socket(socket.AF\_INET,socket.SOCK\_DGRAM)

while True:

cmd=input('>>:')

c.sendto(cmd.encode('utf-8'),('127.0.0.1',8080))

data,addr=c.recvfrom(1024)

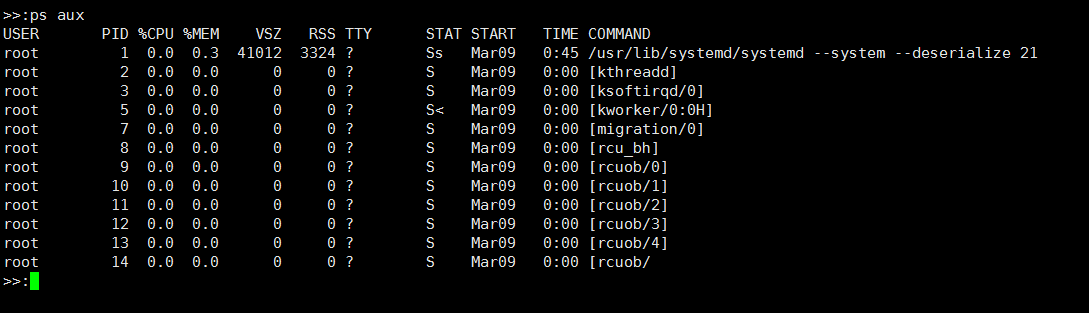
print(data.decode('gbk'))

[复制代码](javascript:void(0);)

在测试的时候，如果返回一个小于1024字节的内容，windows会输出，但是如果返回结果大于接收端要接收的字节时，那么就会报错。这是Windows的一种机制，可以看到，我执行 'ipconfig /all' 时返回长度为2338，而接收端只收1024，剩下的字节就丢包了。因为udp属于不可靠传输，不可靠就在于我数据发送出去后，不管对端有没有收到，我都把我的缓冲区清空。即使接收端收不全，服务端也不会给你重发了。Windows有限制，所以我们只能在linux上做测试了，注意客户端接收的编码为utf-8。然后我执行一个内容比较长的命令，'ps aux' 正常情况下是8000多字节(图一)，通过python测试后(图二)

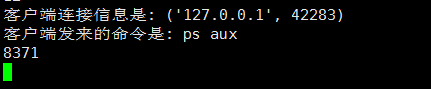
https://images2015.cnblogs.com/blog/1092539/201704/1092539-20170428103347194-453992569.png

　　　　　　　　　　　　图一



　　　　　　　　　　　　　　　　　　　　　　　　　　图二

服务端接收的信息



可以看到图二跟服务端接收的字节完全不相符，也就是不管服务端发多少，我就接收1024个字节

所以说udp是不会发生粘包的 ，并且是不可靠传输

　　刚刚我们基于两种不同的协议分别编写了远程执行命令的脚本，发现tcp有粘包现象，我们也用了一种比较low的方法解决了，下面我们就用一个比较牛X的方法来解决这个粘包问题

**自定义报头**

　　上面我们用struct模块自定义报头，我们之前直接用数据的长度打包，这样会有问题的，我们上面打包命令是：'struct.pack('i',123456)，这里面的'i'代表整型，整型最多可以打包成4个字节，如果不够用，也可以用'q'为长整型，打包成8个字节。但是问题就出现在这，如果我的数据很大，上T了。比如'num=1234567891011121314151617181920'这么长，那么我struct.pack就无法打包进去。

解决方法:

您想，上述的方法要定义报头，报头是在真实数据前面的一段描述信息，里面差不多包含了一些数据的介绍，文件名，文件大小，哈希值等等

所以我们就用字典的形式来定义报头

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket

import subprocess #使用这个模块可以调用操作系统的命令

import struct

import json

s=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)

s.bind(('127.0.0.1',8080))

s.listen(5)

while True:

#连接循环

conn,addr=s.accept()

print('新的客户端连接',addr)

while True:

try:

# 通信循环

cmd = conn.recv(1024)

cmd=cmd.decode('utf-8')

if not cmd: break

print('客户端发来的命令是:%s' % cmd)

cmd\_res=subprocess.Popen(cmd,shell=True,

stderr=subprocess.PIPE,

stdout=subprocess.PIPE,

)

err=cmd\_res.stderr.read()

if err:

res=err

else:

res=cmd\_res.stdout.read()

#自定义报头信息

head\_dic={'filename':'a.txt','size':len(res)}#将报头设置为字典格式，定义文件名，文件大小，查看长度

head\_json=json.dumps(head\_dic) #序列化字典，将字典转换成字符串

head\_bytes=head\_json.encode('utf-8') #将字符串转换成二进制。进行网络通信

conn.send(struct.pack('i',len(head\_bytes)))

#第一次发送head\_bytes的长度信息到客户端

conn.send(head\_bytes)

#第二次，发送报头

conn.send(res)

#第三次，发送真实数据

except ConnectionResetError:

break

conn.close()

s.close()

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket

import struct

import json

clinet=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)

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

while True:

cmd=input('>>:')

if not cmd:continue

clinet.send(cmd.encode('utf-8'))

#第一阶段：收数据的长度

x=clinet.recv(4) #报头的长度

head\_len=struct.unpack('i',x)[0] #解出报头的长度

#第二次接收数据

head\_bytes=clinet.recv(head\_len) #接收报头的长度，为bytes格式

head\_json=head\_bytes.decode('utf-8') #将bytes格式报头解码为json格式

head\_dic=json.loads(head\_json) #使用json序列化将字符串转换成字典

data\_len=head\_dic['size'] #取出真实数据的长度

#第二阶段：根据上面取出的长度，收真实数据

recv\_size= 0 #定义一个空值

res=b'' #定义一个空bytes

while recv\_size < data\_len: #做一个while循环取值，我每次只取1024

recv\_data=clinet.recv(1024)

res+=recv\_data #res每次加取到的值

recv\_size+=len(recv\_data) #len一下 取到数据，并不是每次都是1024，可能最后一次只去了10个字节

print('收到的大小:%s 数据的总大小:%s' %(recv\_size,data\_len))

print(res.decode('gbk'))

clinet.close()

[复制代码](javascript:void(0);)

**socketserver**

　　知道大家有没有了解到，基于socket编程并没有实现并发的效果，有人可能会问，基于udp协议的难道不是么？ 基于udp协议的也不是并发，只不过是执行的速度很快，给人的感觉是并发效果。所以这篇博客有一点是来处理这个问题的，大家请往下看

想要实现并发的效果就需要使用上面的socketserver了，而不是socket了

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socketserver

#导入socketserver

class MyServer(socketserver.BaseRequestHandler):

#自定义一个类，继承BaseRequestHandler(通信请求)

def handle(self):

#在类中必须定义一个handle方法

print(self)

#查看一下self

if \_\_name\_\_ == '\_\_main\_\_':

s=socketserver.ThreadingTCPServer(('127.0.0.1',8080),MyServer)#ThreadingTCPServer(基于TCP的多线程)

#括号里面要写上ip地址与端口，还有自己定义的类名

s.serve\_forever()

#永久提供服务

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket

c=socket.socket()

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

while True:

inp=input('>>:')

if not inp:continue

c.send(inp.encode('utf-8'))

data=c.recv(1024)

print(data)

[复制代码](javascript:void(0);)

　　可以复制多个客户端进行测试，是否实现并发。

　　执行上面代码后，服务端会打印一个对象，这个对象就是客户端来一个连接，ThreadingTCPServer会拿到，并向客户端丢一个叫MyServer的自定义类，并实例化连接，触发里面的handle方法

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socketserver

class MyServer(socketserver.BaseRequestHandler):

def handle(self):

print(self.request)

#进一步查看self下面的request

if \_\_name\_\_ == '\_\_main\_\_':

s=socketserver.ThreadingTCPServer(('127.0.0.1',8080),MyServer)

s.serve\_forever()

[复制代码](javascript:void(0);)

　　通过打印self.request发现 self.request就是我们之前使用socket编程时的那个conn，也就是客户端与服务端连接的那条线

　　那么我们进一步就知道了，handle方法就是为了解决通信问题，但是上面的代码中并没有通信循环那么一说，怎么解决呢？加while啊

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socketserver

class MyServer(socketserver.BaseRequestHandler):

def handle(self):

print(self.request)

while True:

data=self.request.recv(1024)

#之前我们使用socket编程时，接收消息是conn.recv，而现在就是self.request.recv,效果都是在接受消息

self.request.send(data.upper())

#同理，回包的时候也是使用self.request.send

if \_\_name\_\_ == '\_\_main\_\_':

s=socketserver.ThreadingTCPServer(('127.0.0.1',8080),MyServer)

s.serve\_forever()

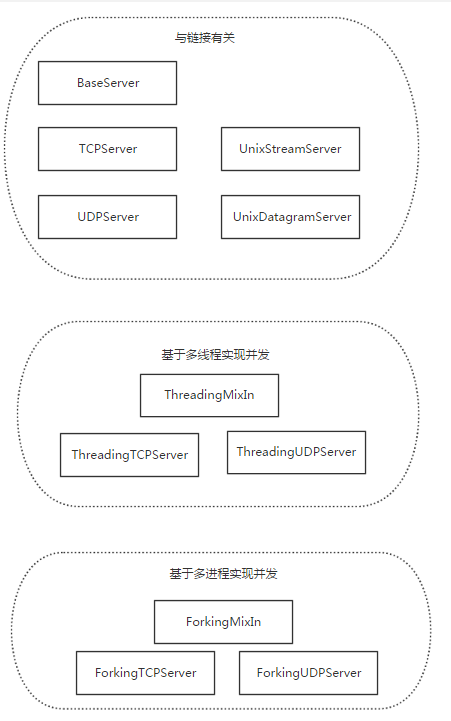
[复制代码](javascript:void(0);)

　　通过上面的服务端代码就可以实现并发的效果了。

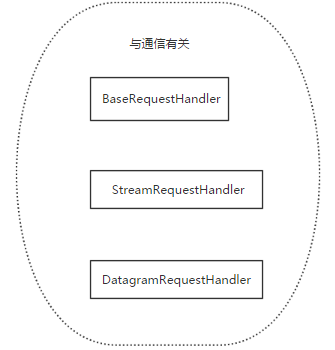
接下来我们简单的了解一下socketserver模块中的类

在socketserver模块中有两大类：server类（解决连接问题），request类（解决通信问题）

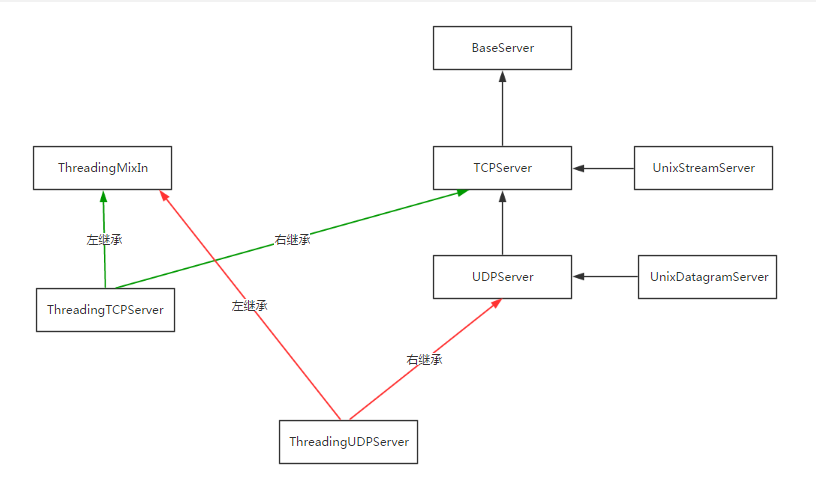
server类

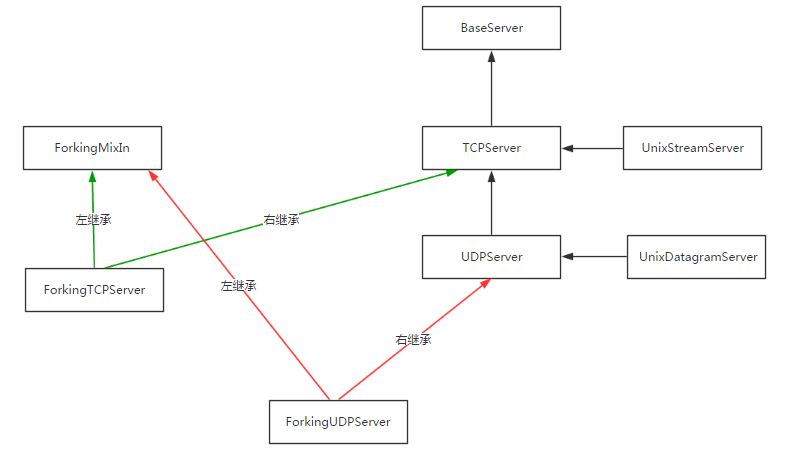


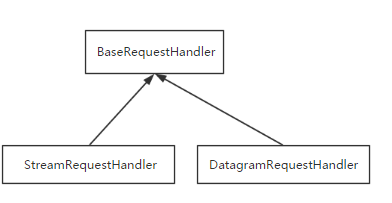
request类



继承关系







以下述代码为例，分析socketserver源码

　　s=socketserver.ThreadingTCPServer(('127.0.0.1',8080),MyServer)

　　s.serve\_forever()

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

　　1.首先实例化，找到TCPServer执行\_\_init\_\_方法，传入两个参数，然后调用BaseServer下面的\_\_init\_\_方法。

　　2.BaseServer接受TCPserver传来的两个参数，定义两个属性(server\_address,RequestHandlerClass)

　　3.实例化完成后，执行TCPserver下面的server\_bind,和server\_activate方法。进行绑定ip地址与端口与监听

　　4.上面的三个步骤就是手机已经开机进行监听状态，在BaseServer中找serve\_forve，进而执行\_handle\_request\_noblock()

　　5.执行self.\_handle\_request\_noblock(),进而执行request, client\_address = self.get\_request()（就是TCPServer中的self.socket.accept()）。然后执行self.process\_request(request, client\_address)

　　6.在ThreadingMixIn中找到process\_request，开启多线程应对并发，进而执行process\_request\_thread。接受两个参数一个request(conn)一个client\_address(addr)。在BaseServer类中执行self.finish\_request(request, client\_address)。调用自己定义的类，传入两个参数。

　　7.然后进行实例化，没有\_\_init\_\_方法。去找继承的BaseRequestHandler类，接着执行self.handle()。也就是MyServer实例化的对象去执行handle()方法。最后执行whilet循环收发消息

源码分析总结：

基于tcp的socketserver我们自己定义的类中的

1. self.server即套接字对象
2. self.request即一个链接
3. self.client\_address即客户端地址

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socketserver

class MyServer(socketserver.BaseRequestHandler):

def handle(self):

print(self)

print(self.client\_address) #查看发送者地址

print(self.request) #一个元组，包括数据与服务端的socket

data=self.request[0] #元组第一个元素为客户端发来的数据

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

addr=self.request[1] #第二个元素就是服务端的socket，通过它进行收发消息

addr.sendto(data.upper(),self.client\_address)

#回包

if \_\_name\_\_ == '\_\_main\_\_':

s=socketserver.ThreadingUDPServer(('127.0.0.1',8080),MyServer)

s.serve\_forever()

[复制代码](javascript:void(0);)

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

import socket

c=socket.socket(socket.AF\_INET,socket.SOCK\_DGRAM)

ip\_port=('127.0.0.1',8080)

while True:

inp=input('>>:')

c.sendto(inp.encode('utf-8'),ip\_port)

data=c.recvfrom(1024)

print(data)

[复制代码](javascript:void(0);)

**标签:**