**1 最基本的服务器与客户端**

**1.1 套接字：通信端点**

**1.1.1 套接字**

  套接字是计算机网络数据结构，在任何通信开始之前，网络应用程序必须创建套接字。可以将它们比作电话的插孔，没有它将无法通信。

  套接字最初是为同一主机上的应用程序所创建，使主机上运行的一个程序（进程）与另一个运行的程序进行通信。这就是所谓的进程间通信。有两种类型的套接字：基于文件的和面向网络的。

**1.1.1.1 基于文件的套接字**

  UNIX 套接字是我们所讲的套接字的第一个家族，并且拥有一个“家族名字”：**AF\_UNIX**（又名 **AF\_LOCAL** ，在 POSIX1.g 标准中指定），它代表地址家族（address family）：**UNIX**。

  包括 Python 在内的大多数受欢迎的平台都使用术语地址家族及其缩写 **AF**；其他比较旧的系统可能会将地址家族表示成域(domain)或协议家族(protocol family)，并使用其缩写 **PF** 而非 **AF**。类似地，**AF\_LOCAL** （在 2000~2001 年标准化）将代替 **AF\_UNIX**。然而，考虑到后向兼容性，很多系统都同时使用二者，只是对同一个常数使用不同的别名。Python 本身仍然在使用 **AF\_UNIX**

**1.1.1.2 面向网络的套接字**

  第二种类型的套接字是基于网络的，它也有自己的家族名字 **AF\_INET**，或者地址家族：**因特网**。另一个地址家族 **AF\_INET6** 用于第 6 版因特网协议(IPv6)寻址。此外，还有其他的地址家族，这些要么是专业的、过时的、很少使用的,要么是仍未实现的。在所有的地址家族之中,目前 AF\_INET 是使用得最广泛的。

  总的来说，**Python** 包含 **AF\_UNIX**、**AF\_NETLINK**、**AF\_TIPC** 和 **AF\_INET** 等家族。下面的内容中,我们将使用 **AF\_INET**。

**1.1.2 套接字地址：主机-端口对**

  如果一个套接字像一个电话插孔一允许通信的一 些基础设施，那么主机名和端口号就像区号和电话号码的组合。然而，拥有硬件和通信的能力本身并没有任何好处，除非你知道电话打给谁以及如何拨打电话。一个网络地址由主机名和端口号对组成，而这是网络通信所需要的。此外，并未事先说明必须有其他人在另一端接听；否则，你将听到这个熟悉的声音“对不起，您所拨打的电话是空号，请核对后再拨”。你可能已经在浏览网页的过程中见过一个网络类比，例如“无法连接服务器，服务器没有响应或者服务器不可达。”

  有效的端口号范围为0~65535 (尽管小于1024的端口号预留给了系统)。如果你正在使用POSIX兼容系统(如Linux、MacOSX等)，那么可以在/etc/services文件中找到预留端口号的列表(以及服务器/协议和套接字类型)。众所周知的端口号列表可以在这个网站中查看：[http://www.iana.org/assignments/port-numbers。](http://www.iana.org/assignments/port-numbers%E3%80%82)

**1.1.3 面向连接的套接字与无连接的套接字**

**1.1.3.1 面向连接的套接字**

  不管你采用的是哪种地址家族，都有两种不同风格的套接字连接。第一种是面向连接的，这意味着在进行通信之前必须先建立一个连接，例如，使用电话系统给一个朋友打电话。 这种类型的通信也称为**虚拟电路**或**流套接字**.。

  面向连接的通信提供序列化的、可靠的和不重复的数据交付，而没有记录边界。这基本上意味着每条消息可以拆分成多个片段，并且每一条消息片段都能确保能够到达目的地，然后将他们按顺序组合在一起，最后将完整消息传递给正在等候的应用程序。

  实现这种连接类型的主要协议是**传输控制协议**（更为人熟知的是它的缩写 **TCP**）。为了创建 **TCP** 套校字，必须使用**SOCK\_STREAM** 作为套接字类型。**TCP** 套接字的名字 **SOCK\_SIREAM** 基于流套接字的其中一种表示。 因为这些套接字（ **AF\_INHT**）的网络版本使用**因特网协议**（ **IP** ） 来搜寻网络中的主句，所以整个系统通常结合这两种协议（ **TCP** 和 **IP** ）来进行（当然，也可以使用 **TCP** 和本地[非网络的 **AF\_LOCALAF**/ **AF\_UNIX**]套接字，但是很明显此时并没有使用 **IP** ）。

**1.1.3.2 无连接的套接字**

  与虚拟电路形成鲜明对比的是**数据报类型**的套接字，它是种无连接的套接字。 这意味着，在通信开始之前并不需要建立连接。此时，在数据传输过程中并无法保证它的顺序性、可靠性或重复性。然而，数据报确实保存了记录边界，这就意味着消息是以整体发送的，而并非首先分成多个片段，例如，使用面向连接的协议。

  使用数据报的消息传输可以比作邮政服务。信件和包裹或许并不能以发送顺序到达。事实上，它们可能不会到达。为了将其添加到并发通信中，在网络中甚至有可能存在重复的消息。

  既然有这么多副作用，为什么还使用数据报呢（使用流套接字肯定有一些优势） ？由于面向连接的套接字所提供的保证，因此它们的设置以及对虚拟电路连接的维护需要大量的开销。然而，数据报不需要这些开销，即它的成本更加“低廉”。因此，它们通常能提供更好的性能，并且可能适合一些类型的应用程序。

  实现这种连接类型的主要协议是**用户数据报协议**（更为人熟知的是其缩写 **UDP**）。 为了创建 **UDP** 套接字，必须使用**SOCK\_DGRAM** 作为套接字类型。你可能知道，**UDP** 套接字的 **SOCK\_ DGRAM** 名字来自于单词“**datagram**”（数据报）。因为这些套接字也使用因特网协议来寻找网络中的主机，所以这个系统也有一个更加普通的名字，即这两种协议（ **UDP** 和 **IP** ) 的组合名字，或**UDP/IP**。

**1.2 socket 模块函数**

  要创建套接字，必须使用 socket.socket()函数，它一般语法如下。

*socket.socket(****family****=AF\_INET,****type****=SOCK\_STREAM,****proto****=0,****fileno****=None)*

  其中，*family*(家族)应该是 **AF\_INET**(默认)，**AF\_INET6**，**AF\_UNIX** 等等。*type*(套接字类型)应该是**SOCK\_STREAM**(默认)，**SOCK\_DGRAM** 等等，*proto*(协议号)通常为0。

  所以，为了创建 **TCP/IP** 套接字，可以用下面的方式调用 **socket.socket()**。

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

  同样，为了创建 **UDP/IP** 套接字，需要执行以下语句

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

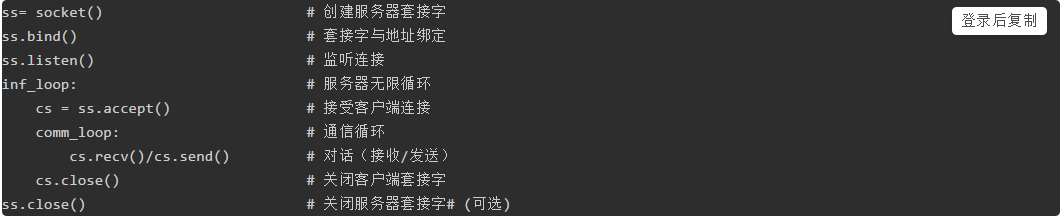
**套接字对象（内置）部分方法**

| **名称** | **描述** |
| --- | --- |
| **服务器套接字方法** |  |
| s.bind(*address*) | 将地址（主机名、端口号对）绑定到套接字上 |
| s.listen(*[backlog]*) | 设置并启动 TCP 监听器 |
| s.accept() | 被动接受 TCP 客户端连接，一直等待到连接到达（阻塞） |
| **客户端套接字方法** |  |
| s.connect(*address*) | 主动发起 TCP 服务器连接 |
| **普通的套接字方法** |  |
| s.recv(*bufsize [,flags]*) | 接收 TCP 消息 |
| s.send(*bytes[, flags]*) | 发送 TCP 消息 |
| s.recvfrom(*bufsize[, flags]*) | 接收 UDP 消息 |
| s.sendto(*bytes, address*) | 发送 UDP消息 |
| s.shutdown(how) | 关闭连接（SHUT\_RD / SHUT\_WR / SHUT\_RDWR） |
| s.close() | 关闭套接字 |
| **面向阻塞的套接字方法** |  |
| s.setblocking(*flag*) | 设置套接字的阻塞或非阻塞模式 |
| s.settimeout(*value*) | 设置阻塞套接字操作的超时时间 |
| s.gettimeout() | 获取阻塞套接字操作的超时时间 |
| **数据属性** |  |
| s.family | 套接字家族 |
| s.type | 套接字类型 |
| s.proto | 套接字协议 |

**1.3 创建服务器**

**1.3.1 一般通用模板**

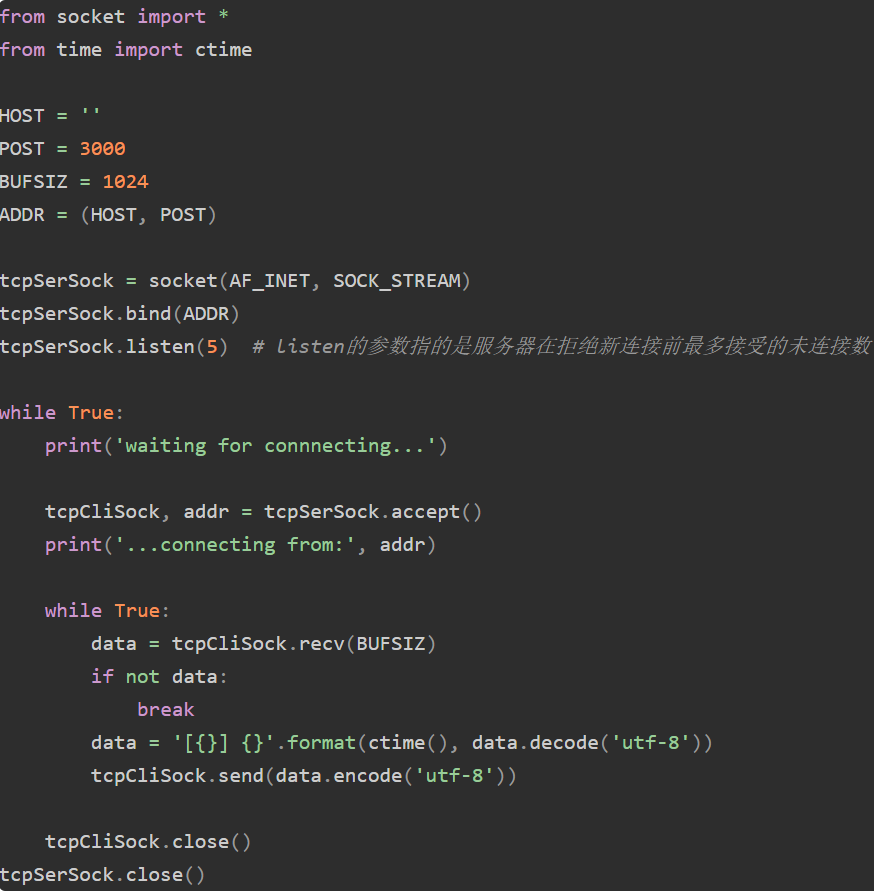
  首先，将展现创建通用 **TCP** 服务器的一般伪代码，然后对这些代码进行一般性的描述。需要记住的是，这仅仅是设计服务器的一种方式。一旦熟悉了服务器的设计，那么你将能够按照自己的要求修改下面的伪代码来操作服务器。

  所有的套接字都是通过socket.socket()函数创建的。

  当调用accept()函数之后，就开启了一个简单的（单线程）服务器，它会等待客户端连接，accept()函数在默认情况下是阻塞的，可以通过setblocking(False)设置为非阻塞模式。

  一旦创建了套接字，通信就开始了，通过这个套接字，客户端与服务器就可以参与发送和接收的对话中，直到连接终止。**当一方关闭连接或者向对方发送一个空字符串时，通常就会关闭连接。**

**1.3.2 编写服务器**



from socket import \*

from time import ctime

HOST = ''

POST = 3000

BUFSIZ = 1024

ADDR = (HOST, POST)

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

tcpSerSock.bind(ADDR)

tcpSerSock.listen(5) # listen的参数指的是服务器在拒绝新连接前最多接受的未连接数

while True:

print('waiting for connnecting...')

tcpCliSock, addr = tcpSerSock.accept()

print('...connecting from:', addr)

while True:

data = tcpCliSock.recv(BUFSIZ)

if not data:

break

data = '[{}] {}'.format(ctime(), data.decode('utf-8'))

tcpCliSock.send(data.encode('utf-8'))

tcpCliSock.close()

tcpSerSock.close()

**逐行解释**  
  第 1~2 行  
  导入了 time.ctime() 和 socket 模块的所有属性

  第 4~11 行  
  **HOST** 变量是空白的，表示使用任何可用的地址，**POST** 应是一个没有被使用或被系统保留的端口号。另外， 对于该应用程序，将缓冲区大小设置为 **1KB**，可用根据网络性能和程序需要该表 **BUFSIZ** 这个变量 。**listen()**方法的参数指的是在连接被转移或拒绝前，传入连接请求的最大数。  
  在第 9 行，分配了 **TCP** 服务器套接字（**tcpSerSock**），紧随其后的是将套接字绑定到服务器地址以及开启 **TCP** 监听器的调用。

  第 13~26 行  
  一旦进入服务器的无限循环之中，我们就（被动地）等待客户端的连接。当一个请求连接出现时，我们进入对话循环中，在该循环中我们等待客户端发送消息。如果消息是空白的，这意味着客户端已经退出，所以我们此时跳出循环，关闭当前客户端连接，然后等待另一个客户端的连接。如果确实得到你客户端发送的信息，就将此消息加上当前时间返回给客户端。**tcpSerSock.accept()** 方法返回的是一个新套接字和客户端地址，服务器与客户端的通信都基于这个新的套接字上。当用户关闭套接字时，**tcpCliSock.recv()** 会接收到一个空字节，我们以此判断是否应该关闭套接字（相应的，服务器关闭套接字也会给客户端发送一个空字节）

**注：**  
  **recv()** 收到的是二进制数据，同样的，**send()** 发送的也是二进制数据，所以我们需要在通信时编码和解码。

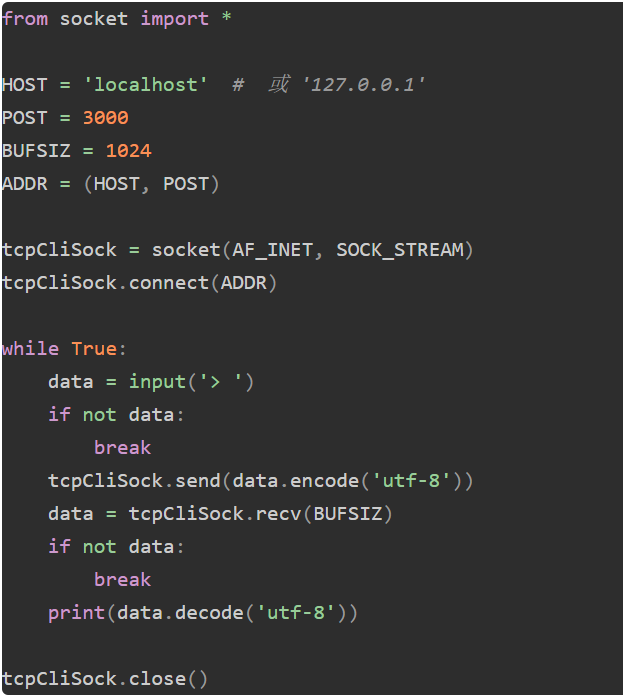
**1.4 创建客户端**

**1.4.1 一般通用模板**

  创建 **TCP** 客户端与服务器类似，先展示伪代码

  和前面一样，socket.socket()创建套接字。套接字的 connect() 方法尝试与服务器建立连接，当与服务器成功建立连接时，就可以参与到与服务器的一个对话中。

**1.4.2 编写客户端**



from socket import \*

HOST = 'localhost' # 或 '127.0.0.1'

POST = 3000

BUFSIZ = 1024

ADDR = (HOST, POST)

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

tcpCliSock.connect(ADDR)

while True:

data = input('> ')

if not data:

break

tcpCliSock.send(data.encode('utf-8'))

data = tcpCliSock.recv(BUFSIZ)

if not data:

break

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

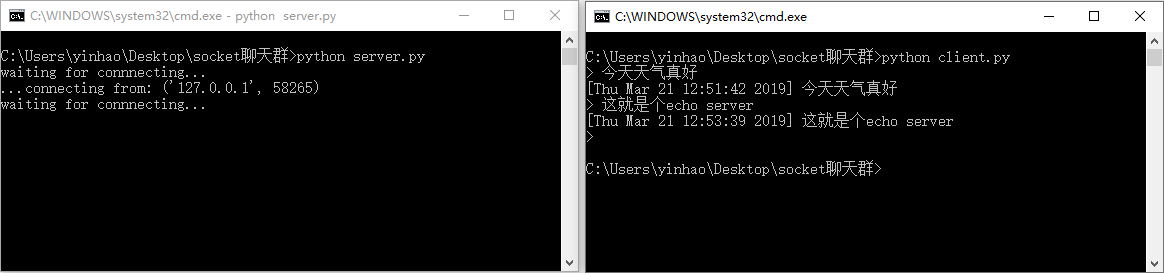
tcpCliSock.close()

**逐行解释**  
  第 1 行  
  导入了 socket 模块的所有属性

  第 3~9 行  
  **HOST** 与 **POST** 变量指服务器的主机号与端口号。因为是本机测试，所以 **HOST** 包含本机主机名。**POST** 需要与服务器设置的一致。缓冲区大小设置为 1KB。

  第 11~21 行  
  和服务器一样，客户端也有一个无线循环。但不同的是，当用户没有输入，或服务器关闭套接字（接收到空字节）的时候，跳出循环。

**1.5 测试结果**

  我将服务器代码保存在server.py里，客户端代码保存在client.py里。先运行服务器，再运行客户端，结果如下：  
  
> 如果出现错误 “OSError: [WinError 10048] 通常每个套接字地址(协议/网络地址/端口)只允许使用一次。” 那么就换个端口再次尝试。

参考文献：  
[1] Python核心编程 第3版

**2 使用select模块管理多个套接字**

  在之前的程序中，服务器一次只能与一个客户端通信。为了使服务器能与多个客户端通信，我们需要对程序稍加改造。

**2.1 select 模块函数**

**select** 模块专注于I/O多路复用，就是确定一个或多个套接字的状态，检查它们的可读性、可写性和错误状态信息。此模块中，提供了三个方法 **select**、**poll** 和 **epoll** （在windows系统中只能用第一个）。

**select.select()** 函数的定义如下：

*select.select(****rlist****,****wlist****,****xlist****[, timeout])*

  前三个参数是“等待检查”的套接字列表，**rlist** 检查可读性，**wlist** 检查可写性，**xlist** 检查错误信息，**timeout** 指定超时时间（浮点数，以秒为单位）

  此函数返回满足一定条件的套接字列表的子集，用法如下：

from select import select

...

readList = [...]

writeList = [...]

errorList = [...]

While True:

rlist, wlist, xlist = select(readList, writeList, errorList)

for s in rlist: # 处理可读的套接字

...

for s in wlist: # 处理可写的套接字

...

for s in xlist: # 处理错误信息

...

**2.2 将 select() 应用于服务器**

from socket import \*

from time import ctime

from select import select

HOST = ''

POST = 3000

BUFSIZ = 1024

ADDR = (HOST, POST)

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

tcpSerSock.bind(ADDR)

tcpSerSock.listen(5)

tcpSerSock.setblocking(False) # 将tcpSerSock设置为非租塞模式

inputs = [tcpSerSock]

print('waiting for connnecting...')

while True:

rlist, wlist, xlist = select(inputs, [], [])

for s in rlist:

if s is tcpSerSock:

tcpCliSock, addr = s.accept()

print('...connecting from:', addr)

tcpCliSock.setblocking(False) # 将tcpCliSock设置为非租塞模式

inputs.append(tcpCliSock) # 将tcpCliSock插入inputs中

else:

data = s.recv(BUFSIZ)

if not data:

inputs.remove(s)

s.close()

continue

data = '[{}] {}'.format(ctime(), data.decode('utf-8'))

s.send(data.encode('utf-8')) # 通常要放到 select() 的第二个参数列表中处理

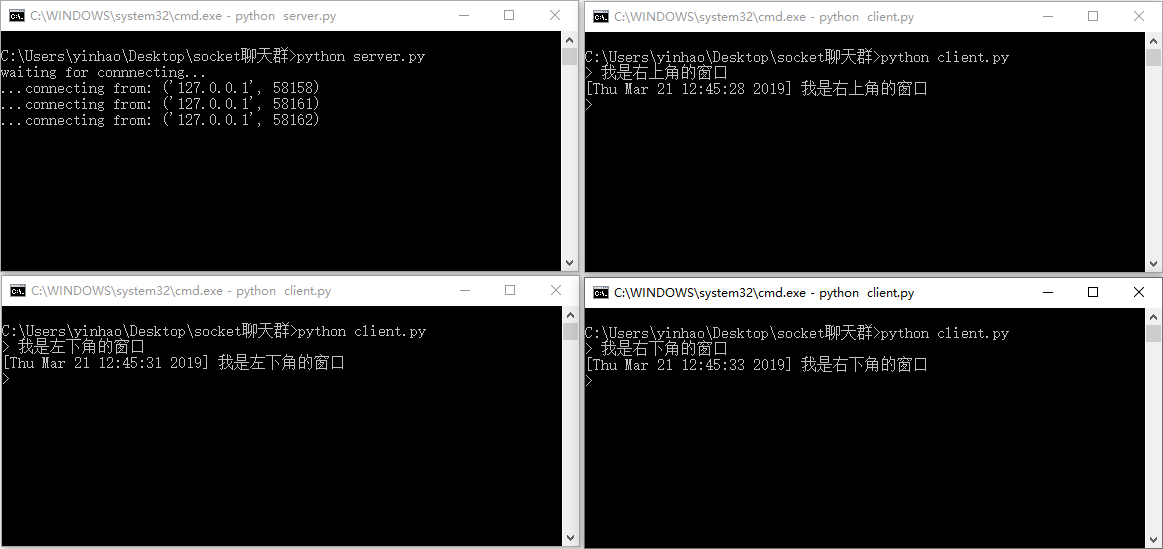
**逐行解释**  
  第 1~12 行  
  与之前的操作相同，不同是，导入了 **select()** 函数。

  第 14~15 行  
  **setblocking()** 函数将套接字设置为非阻塞的。**inputs** 包含需要被 **select()** 检查可读性的套接字变量。

  第 19~39 行  
  无限循环检查 **inputs** 中套接字的可读性，当有满足条件套接字时（客户端连接请求和客户端发送消息）返回 **rlist**、**wlist**、**xlist** 三个变量，在这里我们只用到了 **rlist**。在第 22 行，遍历 **rlist** 中所有套接字，如果 **s** 是连接套接字（**tcpCliSock**），那么就接受客户端的连接请求，并将返回的新套接字插入 **inputs** 中。如果 **s** 是通信套接字，那么就接受信息，处理并返回。

**2.3 测试结果**

  我先运行服务器，再运行三个客户端，结果如下：



**3 使用tkinter模块打造聊天群界面**

**3.1 代码**

  这一节我不打算详细讲解（因为会占用太多与主题无关的篇幅，而且我也不喜欢这个库），有兴趣的可以自行了解，我只会讲解界面程序中每一步的用处。

from tkinter import \*

root = Tk()

sw = root.winfo\_screenwidth() # 获取屏幕宽度

sh = root.winfo\_screenheight() # 获取屏幕高度

root.geometry('+{}+{}'.format((sw-460)//2, (sh-400)//2)) # 窗口居中

root.title('Python聊天群') # 设置窗口标题

frameT = Frame(root, width=460, height=320) # 顶部容器(root为父容器)

frameT.pack(expand='yes', fill='both')

frameB = Frame(root, width=460, height=80) # 底部容器(root为父容器)

frameB.pack(expand='yes', fill='both')

Output = Text(frameT) # 显示文本框(frameT为父容器)

Output.pack(expand='yes', fill='both')

Input = Text(frameB, height=6) # 输入文本框(frameB为父容器)

Input.pack(expand='yes', fill='both')

btnFrame = Frame(frameB, height=24, bg='White') # 按钮容器(frameB为父容器)

btnFrame.pack(expand='yes', fill='both')

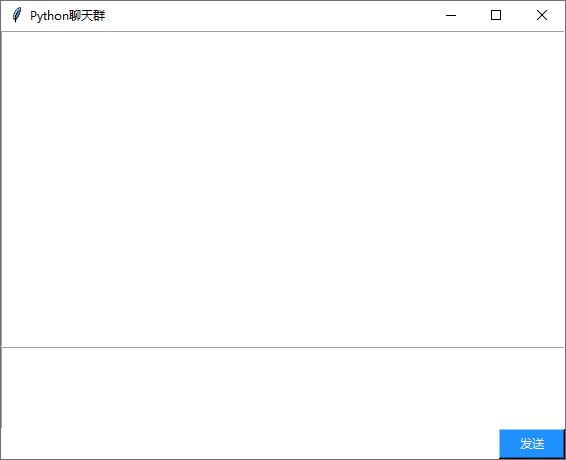
# 发送按钮(btnFrame为父容器)

Button(btnFrame, text='发送', width=8, bg='DodgerBlue', fg='White').pack(side=RIGHT)

root.mainloop() # 窗口主循环

**3.2 运行结果**

  这是最基本的图形界面，运行结果如下：



**4 使用threading模块使客户端收发分离**

from threading import Thread

class ReceiveThread(Thread):

def \_\_init\_\_(self, tcpCliSock, BUFSIZ=1024):

Thread.\_\_init\_\_(self)

self.daemon = True # 守护线程

self.tcpCliSock = tcpCliSock

self.BUFSIZ = BUFSIZ

def run(self):

while True:

data = tcpCliSock.recv(self.BUFSIZ)

if not data:

tcpCliSock.close()

root.destroy() # 销毁窗口

else:

Output.insert(END, data.decode('utf-8')) # END是tk定义的标记

**逐行解释**  
  第 1 行  
  导入 **threading** 中的 **Thread** 类（threading模块是对thread模块的封装，但不建议使用thread模块）。

  第 3 行  
  定义一个消息接收线程，继承于 **Thread** 类。

  第 5~6 行  
  调用父类构造函数，设置线程为守护线程。

  第 10 行  
  重写 **run** 函数，线程启动时调用此函数（注：线程启动时应使用 start() 函数而不是 run() 函数）

  第 17 行  
  将收到的消息显示在 **Output** 文本框中。

**5 聊天群程序的整合与完成**

**5.1 [客户端]点击发送按钮发送消息**

  首先，为了区分各个客户端，所有先给各个客户端的用户取个名。发送消息时将消息和姓名一同发给服务器。

**5.1.1 给客户端用户取名**

from random import choice

NAME = choice(['业冰蝶','同静槐','骑婷然','牧建章','锐理全','达悠逸','倪长逸','侨玉书','符天韵','树修敏'])

root.title('Python聊天群 ({})'.format(NAME))

**逐行解释**  
  第 1 行  
  导入 **random** 模块的 **choice** 函数。

  第 2 行  
  从列表中随机选择一个名字赋值给 **NAME**。

  第 3 行  
  在窗口的标题上显示姓名

**5.1.2 编写“发送”按钮的回调函数**

from json import dumps

def sendMessage():

# 发送消息

msg = Input.get('1.0', END)

tcpCliSock.send(dumps({

'name': NAME,

'msg' : msg

}).encode('utf-8'))

Input.delete('1.0', END)

Button(..., text='发送', command=sendMessage).pack()

**逐行解释**  
  第 1 行  
  导入 **json** 中的 **dumps** 函数，**dumps** 函数的用处是将**字典(dict)**类型的变量转成**字符串(str)**类型。如：json.dumps({ ‘name’: ‘小明’, ‘msg’: ‘今天天气真好’ }) 会返回字符串 ‘{“name”: “\u5c0f\u660e”, “msg”: “\u4eca\u5929\u5929\u6c14\u771f\u597d”}’。

  第 4 行  
  获取 **Input** 文本框中的字符串

  第 9 行  
  删除 **Input** 文本框中的字符串

  第 11 行  
  这里是对之前的按钮代码进行修改，给它传入回调函数 **command**=sendMessage（注：为了直观的展现，所以隐藏了其他属性，在这里只需要给command参数传入回调函数），完整写法如下：

Button(btnFrame, text='发送', width=8, bg='DodgerBlue', fg='White', command=sendMessage).pack(side=RIGHT)

**5.2 [客户端]处理窗口退出事件**

  当程序退出前，应当关闭套接字，否则会导致客户端与服务器崩溃。当然，这并不难解决，就三行代码。

def onClosing():

tcpCliSock.shutdown(SHUT\_WR)

root.protocol("WM\_DELETE\_WINDOW", onClosing) # 退出时处理

**shutdown()** 函数在 [#1.2 socket 模块函数](https://blog.csdn.net/weixin_40522523/article/details/88688803#12_socket__41) 的表格中提到过，它用来关闭连接。可以传入三个值：  
    **SHUT\_RDWR** ：关闭读写，即不可以使用 *send*、*write*、*recv*、*read*  
    **SHUT\_RD** ：关闭读，即不可以使用 *recv*、*read*  
    **SHUT\_WR** ：关闭写，即不可以使用 *send*、*write*

  当 **shutdown()** 函数运行后，客户端回向服务器发送一个空字节，服务器收到空字节后关闭服务器端的通信套接字，同时也向客户端发送一个空字节，接收线程接收到空字节后，关闭套接字、销毁窗口，然后退出线程。（注：不要在此对套接字使用 close() 函数，至于理由可以自行尝试）

  第 3 行，就是对窗口关闭事件添加回调函数（注：添加回调函数后必须显式添加窗口销毁函数，如 [#4 使用threading模块使客户端收发分离](https://blog.csdn.net/weixin_40522523/article/details/88688803#4_threading_323) 倒数第三行代码 “root.destroy()”）

**5.3 最终的客户端**

  各部分在之前章节都有讲解

from tkinter import \*

from socket import \*

from random import choice

from json import dumps

from threading import Thread

class ReceiveThread(Thread):

def \_\_init\_\_(self, tcpCliSock, BUFSIZ=1024):

Thread.\_\_init\_\_(self)

self.daemon = True # 守护线程

self.tcpCliSock = tcpCliSock

self.BUFSIZ = BUFSIZ

def run(self):

while True:

data = tcpCliSock.recv(self.BUFSIZ)

if not data:

tcpCliSock.close()

root.destroy()

else:

Output.insert(END, data.decode('utf-8'))

def sendMessage():

# 发送消息

msg = Input.get('1.0', END)

tcpCliSock.send(dumps({

'name': NAME,

'msg' : msg

}).encode('utf-8'))

Input.delete('1.0', END)

def onClosing():

tcpCliSock.shutdown(SHUT\_WR)

HOST = 'localhost'

POST = 3000

BUFSIZ = 1024

ADDR = (HOST, POST)

NAME = choice(['业冰蝶','同静槐','骑婷然','牧建章','锐理全','达悠逸','倪长逸','侨玉书','符天韵','树修敏'])

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

tcpCliSock.connect(ADDR)

root = Tk()

sw = root.winfo\_screenwidth()

sh = root.winfo\_screenheight()

root.geometry('+{}+{}'.format((sw-430)//2, (sh-340)//2))

root.title('Python聊天群 ({})'.format(NAME))

frameT = Frame(root, width=460, height=320)

frameB = Frame(root, width=460, height=80)

frameT.pack(expand='yes', fill='both')

frameB.pack(expand='yes', fill='both')

Input = Text(frameB, height=6)

Output = Text(frameT)

Input.pack(expand='yes', fill='both')

Output.pack(expand='yes', fill='both')

btnFrame = Frame(frameB, height=24, background='White')

btnFrame.pack(expand='yes', fill='both')

Button(btnFrame, text='发送', width=8, bg='DodgerBlue', fg='White', command=sendMessage).pack(side=RIGHT)

ReceiveThread(tcpCliSock).start() # 启动消息接收线程

root.protocol("WM\_DELETE\_WINDOW", onClosing) # 退出时处理

root.mainloop()

**5.4 最终的服务器**

from socket import \*

from time import strftime, localtime

from select import select

from json import loads

HOST = ''

POST = 3000

BUFSIZ = 1024

ADDR = (HOST, POST)

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

tcpSerSock.bind(ADDR)

tcpSerSock.listen(5)

tcpSerSock.setblocking(False)

inputs = [tcpSerSock]

print('waiting for connnecting...')

while True:

rlist, wlist, xlist = select(inputs, [], [])

for s in rlist:

if s is tcpSerSock:

tcpCliSock, addr = s.accept()

print('...connecting from:', addr)

tcpCliSock.setblocking(False)

inputs.append(tcpCliSock)

else:

data = s.recv(BUFSIZ)

if not data:

inputs.remove(s)

s.close()

continue

obj = loads(data.decode('utf-8'))

time = strftime("%Y-%m-%d %H:%M:%S", localtime())

data = '[{}]{}: {}'.format(time, obj['name'], obj['msg'])

for sock in inputs:

if sock is not tcpSerSock:

sock.send(data.encode('utf-8'))

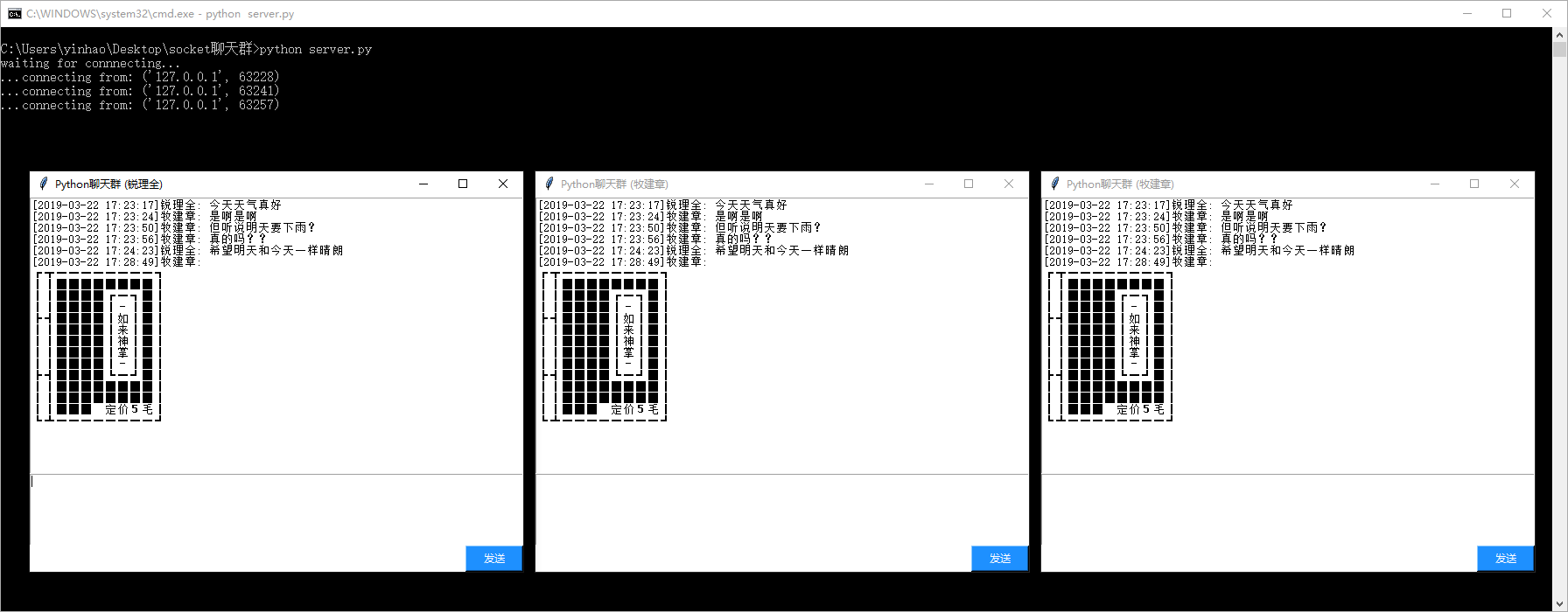
  此处与 [#2.2 将 select() 应用于服务器](https://blog.csdn.net/weixin_40522523/article/details/88688803#22__select__228) 稍有不同。

  倒数第 6 行  
  **json.loads()** 将 **json.dumps()** 转成的字符串又重新解析成 **字典(dict)** 类型。

  倒数第 5~4 行  
  对数据进行一些处理。

  倒数第 3~1 行  
  对所有已经连接的客户端发送数据。

**5.5 最终成果展示**

  
[代码已上传至github](https://github.com/yinhaox/socket_chat)（虽然就俩文件）