

Socket介绍

Socket套接字。Socket是一种通用的网络编程接口，和网络层次没有一一对应的关系。

Python中标准库中提供了socket模块。socket模块中也提供了socket类，实现了对底层接口的封装，socket模块是非常底层的接口库。

socket类定义为

```
1 | socket(self, family=AF_INET, type=SOCK_STREAM, proto=0, fileno=None)
```

协议族

AF表示Address Family，用于socket()第一个参数

名称	含义
AF_INET	IPV4
AF_INET6	IPV6
AF_UNIX	Unix Domain Socket，windows没有

Socket类型

名称	含义
SOCK_STREAM	面向连接的流套接字。默认值，TCP协议
SOCK_DGRAM	无连接的数据报文套接字。UDP协议

TCP协议是流协议，也就是一大段数据看做字节流，一段段持续发送这些字节。

UDP协议是数据报协议，每一份数据封在一个单独的数据报中，一份一份发送数据。

socket常用方法

socket类创建出socket对象，这个对象常用方法如下

名称	含义
socket.recv(bufsize[, flags])	获取数据。默认是阻塞的方式
socket.recvfrom(bufsize[, flags])	获取数据，返回一个二元组(bytes, address)
socket.recv_into(buffer[, nbytes[, flags]])	获取到nbytes的数据后，存储到buffer中。如果nbytes没有指定或0，将buffer大小的数据存入buffer中。返回接收的字节数。
socket.recvfrom_into(buffer[, nbytes[, flags]])	获取数据，返回一个二元组(bytes, address)到buffer中
socket.send(bytes[, flags])	TCP发送数据，发送成功返回发送字节数
socket.sendall(bytes[, flags])	TCP发送全部数据，成功返回None
socket.sendto(string[, flag], address)	UDP发送数据
socket.sendfile(file, offset=0, count=None)	发送一个文件直到EOF，使用高性能的os.sendfile机制，返回发送的字节数。如果win下不支持sendfile，或者不是普通文件，使用send()发送文件。offset告诉起始位置。3.5版本开始

名称	含义
socket.getpeername()	返回连接套接字的远程地址。返回值通常是元组(ipaddr, port)
socket.getsockname()	返回套接字自己的地址。通常是一个元组(ipaddr, port)
socket.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
socket.settimeout(value)	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()）
socket.setsockopt(level, optname, value)	设置套接字选项的值。比如缓冲区大小。太多了，去看文档。不同系统，不同版本都不尽相同

TCP编程

C/S编程

Socket编程，是完成一端和另一端通信的，注意一般来说这两端分别处在不同的进程中，也就是说网络通信是一个进程发消息到另外一个进程。

我们写代码的时候，每一个socket对象只表示了其中的一端。

从业务角度来说，这两端从角色上分为：

- 主动发送请求的一端，称为客户端Client

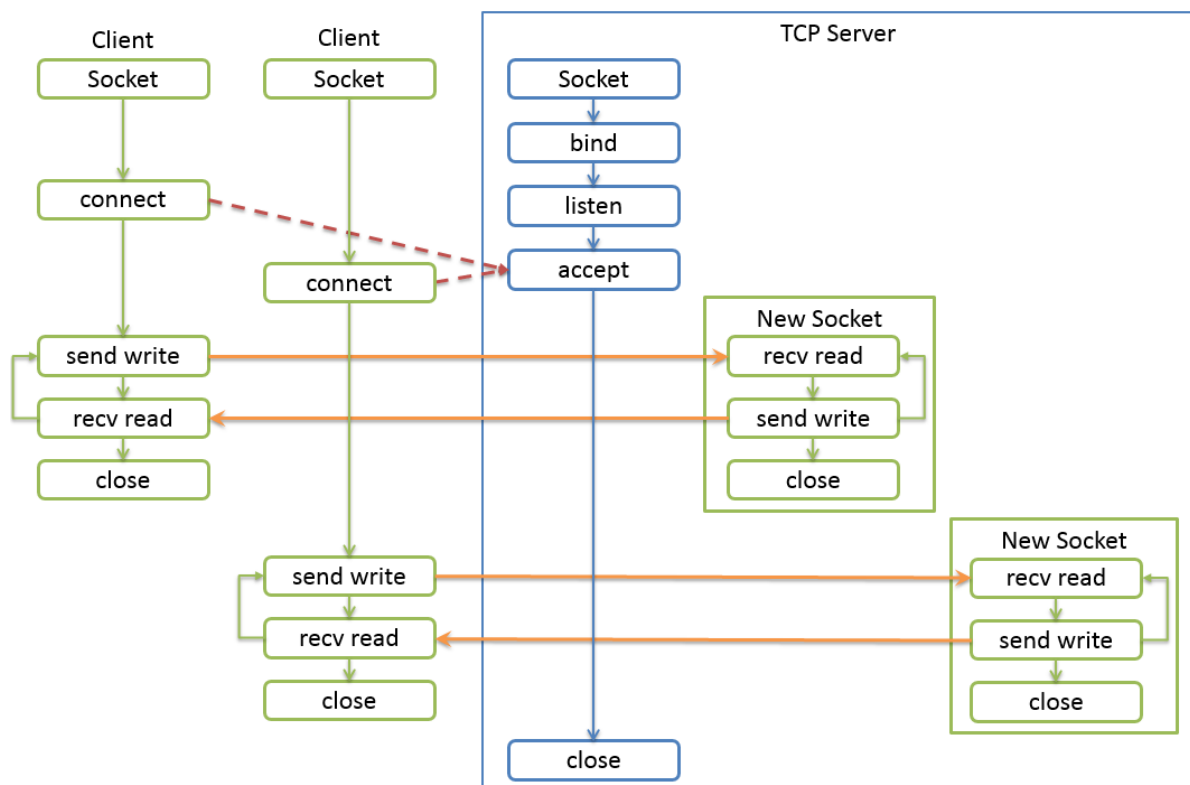
- 被动接受请求并回应的一端，称为服务端Server

这种编程模式也称为**C/S编程**。

服务器端编程步骤

- 创建Socket对象
- 绑定IP地址Address和端口Port。bind()方法
IPv4地址为一个二元组('IP地址字符串', Port)
- 开始监听，将在指定的IP的端口上监听。listen()方法
- 获取用于传送数据的Socket对象
socket.accept() -> (socket object, address info)
accept方法阻塞等待客户端建立连接，返回一个新的Socket对象和客户端地址的二元组
地址是远程客户端的地址，IPv4中它是一个二元组(clientaddr, port)
 - 接收数据
recv(bufsize[, flags]) 使用缓冲区接收数据
 - 发送数据
send(bytes)发送数据

```
1 Server端开发
2 socket对象 --> bind((IP, PORT)) --> listen --> accept --> close
3 |--> recv or send --> close
```



问题

两次绑定同一个监听端口会怎么样？

socket初识

```
1 import socket
2
3 s = socket.socket() # 创建socket对象
4 s.bind(('127.0.0.1', 9999)) # 一个地址和端口二元组
5 s.listen() # 开始监听，等待客户端连接到来，准备accept
6
7 # 接入一个到来的连接
8 s1, info = s.accept() # 阻塞，直到和客户端成功建立连接，返回一个新的socket对象和客户端地址
9 print(type(s1), type(info))
10 print(s1)
11 print(info)
12 sockname = s1.getsockname()
13 peername = s1.getpeername()
14 print(type(sockname), sockname) # 本地地址
15 print(type(peername), peername) # 对端地址
16 print('-' * 30)
17
18 # 使用缓冲区获取数据
19 data = s1.recv(1024) # 阻塞
20 print(type(data), data)
21 s1.send(b'magedu.com ack') # bytes
22 s1.close() # 关闭
23
24 # 接入另外一个连接
25 s2, info = s.accept() # 阻塞
26 data = s2.recv(1024)
27 print(info, data)
28 s2.send(b'hello python ack')
29 s2.close() # 关闭
30
31 s.close() # 关闭
```

上例accept和recv是阻塞的，主线程经常被阻塞住而不能工作。怎么办？

查看监听端口

```
1 windows 命令
2 # netstat -an -p tcp | findstr 9999
3
4 linux命令
5 # netstat -tanl | grep 9999
6 # ss -tanl | grep 9999
```

实战——写一个群聊程序

需求分析

聊天工具是CS程序，C是每一个客户端client，S是服务器端server。

服务器应该具有的功能：

1. 启动服务，包括绑定地址和端口，并监听
2. 建立连接，能和多个客户端建立连接
3. 接收不同用户的信息
4. 分发，将接收的某个用户的信息转发到已连接的所有客户端
5. 停止服务
6. 记录连接的客户端

代码实现

服务端应该设计为一个类

```
1 class ChatServer:
2     def __init__(self, ip, port): # 启动服务
3         self.sock = socket.socket()
4         self.addr = (ip, port)
5
6     def start(self): # 启动监听
7         pass
8
9     def accept(self): # 多人连接
10        pass
11
12    def recv(self): # 接收客户端数据
13        pass
14
15    def stop(self): # 停止服务
16        pass
```

在此基础上，扩展完成

```
1 import logging
2 import socket
3 import threading
4 import datetime
5
6 logging.basicConfig(level=logging.INFO, format="%(asctime)s %(thread)d %
7 (message)s")
8
9 class ChatServer:
10    def __init__(self, ip='127.0.0.1', port=9999): # 启动服务
11        self.sock = socket.socket()
12        self.addr = (ip, port)
13        self.clients = {} # 客户端
14
15
16    def start(self): # 启动监听
17        self.sock.bind(self.addr) # 绑定
18        self.sock.listen() # 监听
19        # accept会阻塞主线程，所以开一个新线程
20        threading.Thread(target=self.accept).start()
```

```

21
22     def accept(self): # 多人连接
23         while True:
24             sock, client = self.sock.accept() # 阻塞
25             self.clients[client] = sock # 添加到客户端字典
26             # 准备接收数据, recv是阻塞的, 开启新的线程
27             threading.Thread(target=self.recv, args=(sock, client)).start()
28
29     def recv(self, sock:socket.socket, client): # 接收客户端数据
30         while True:
31             data = sock.recv(1024) # 阻塞到数据到来
32             msg = "{:%Y/%m/%d %H:%M:%S} {}:
33             {}\n{}\n".format(datetime.datetime.now(), *client, data.decode())
34             logging.info(msg)
35             msg = msg.encode()
36             for s in self.clients.values():
37                 s.send(msg)
38
39     def stop(self): # 停止服务
40         for s in self.clients.values():
41             s.close()
42         self.sock.close()
43
44 cs = ChatServer()
45 cs.start()

```

基本功能完成, 但是有问题。使用Event改进。先实现单独聊, 然后改成群聊

```

1  import logging
2  import socket
3  import threading
4  import datetime
5
6  logging.basicConfig(level=logging.INFO, format="%(asctime)s %(thread)d %
7  (message)s")
8
9  class ChatServer:
10     def __init__(self, ip='127.0.0.1', port=9999): # 启动服务
11         self.sock = socket.socket()
12         self.addr = (ip, port)
13         self.clients = {} # 客户端
14         self.event = threading.Event()
15
16
17     def start(self): # 启动监听
18         self.sock.bind(self.addr) # 绑定
19         self.sock.listen() # 监听
20         # accept会阻塞主线程, 所以开一个新线程
21         threading.Thread(target=self.accept).start()
22
23     def accept(self): # 多人连接
24         while not self.event.is_set():
25             sock, client = self.sock.accept() # 阻塞
26             self.clients[client] = sock # 添加到客户端字典
27             # 准备接收数据, recv是阻塞的, 开启新的线程
28             threading.Thread(target=self.recv, args=(sock, client)).start()

```

```

29
30     def recv(self, sock:socket.socket, client): # 接收客户端数据
31         while not self.event.is_set():
32             data = sock.recv(1024) # 阻塞到数据到来
33             msg = "{:%Y/%m/%d %H:%M:%S} {}:
{{}\n{}\n".format(datetime.datetime.now(), *client, data.decode())
34             logging.info(msg)
35             msg = msg.encode()
36             for s in self.clients.values():
37                 s.send(msg)
38
39     def stop(self): # 停止服务
40         self.event.set()
41         for s in self.clients.values():
42             s.close()
43         self.sock.close()
44
45 cs = ChatServer()
46 cs.start()
47
48 while True:
49     cmd = input('>>').strip()
50     if cmd == 'quit':
51         cs.stop()
52         threading.Event().wait(3)
53         break

```

这一版基本能用了，测试通过。但是还有要完善的地方。
例如各种异常的判断，客户端断开连接后字典中的移除客户端数据等。

客户端主动断开带来的问题

服务端知道自己何时断开，如果客户端断开，服务器不知道。（客户端主动断开，服务端recv会得到一个空串）

所以，好的做法是，客户端断开发出特殊消息通知服务器端断开连接。但是，如果客户端主动断开，服务端主动发送一个空消息，超时返回异常，捕获异常并清理连接。

即使为客户端提供了断开命令，也不能保证客户端会使用它断开连接。但是还是要增加这个退出功能。

增加客户端退出命令

```

1  import logging
2  import socket
3  import threading
4  import datetime
5
6  logging.basicConfig(level=logging.INFO, format="%(asctime)s %(thread)d %
(message)s")
7
8
9  class ChatServer:
10     def __init__(self, ip='127.0.0.1', port=9999): # 启动服务
11         self.sock = socket.socket()
12         self.addr = (ip, port)
13         self.clients = {} # 客户端
14         self.event = threading.Event()
15
16

```

```

17     def start(self): # 启动监听
18         self.sock.bind(self.addr) # 绑定
19         self.sock.listen() # 监听
20         # accept会阻塞主线程，所以开一个新线程
21         threading.Thread(target=self.accept).start()
22
23     def accept(self): # 多人连接
24         while not self.event.is_set():
25             sock, client = self.sock.accept() # 阻塞
26             self.clients[client] = sock # 添加到客户端字典
27             # 准备接收数据，recv是阻塞的，开启新的线程
28             threading.Thread(target=self.recv, args=(sock, client)).start()
29
30     def recv(self, sock:socket.socket, client): # 接收客户端数据
31         while not self.event.is_set():
32             data = sock.recv(1024) # 阻塞到数据到来
33             msg = data.decode().strip()
34             # 客户端退出命令
35             if msg == 'quit' or msg == '': # 主动断开得到空串
36                 self.clients.pop(client)
37                 sock.close()
38                 logging.info('{} quits'.format(client))
39                 break
40             msg = "{:%Y/%m/%d %H:%M:%S} {}:".format(datetime.datetime.now(), *client, data.decode())
41             logging.info(msg)
42             msg = msg.encode()
43             for s in self.clients.values():
44                 s.send(msg)
45
46     def stop(self): # 停止服务
47         self.event.set()
48         for s in self.clients.values():
49             s.close()
50         self.sock.close()
51
52 cs = ChatServer()
53 cs.start()
54
55 while True:
56     cmd = input('>>').strip()
57     if cmd == 'quit':
58         cs.stop()
59         threading.Event().wait(3)
60         break
61     logging.info(threading.enumerate()) # 用来观察断开后线程的变化

```

程序还有瑕疵，但是业务功能基本完成了

线程安全

由于GIL和内置数据结构的读写原子性，单独操作字典的某一项item是安全的。但是遍历过程是线程不安全的，遍历中有可能被打断，其他线程如果对字典元素进行增加、弹出，都会影响字典的size，就会抛出异常。所以还是要加锁Lock。

加锁后的代码如下


```

1  import logging
2  import socket
3  import threading
4  import datetime
5
6  logging.basicConfig(level=logging.INFO, format="%(asctime)s %(thread)d %(message)s")
7
8
9  class ChatServer:
10     def __init__(self, ip='127.0.0.1', port=9999): # 启动服务
11         self.sock = socket.socket()
12         self.addr = (ip, port)
13         self.clients = {} # 客户端
14         self.event = threading.Event()
15         self.lock = threading.Lock()
16
17     def start(self): # 启动监听
18         self.sock.bind(self.addr) # 绑定
19         self.sock.listen() # 监听
20         # accept会阻塞主线程，所以开一个新线程
21         threading.Thread(target=self.accept).start()
22
23     def accept(self): # 多人连接
24         while not self.event.is_set():
25             sock, client = self.sock.accept() # 阻塞
26             with self.lock:
27                 self.clients[client] = sock # 添加到客户端字典
28             # 准备接收数据，recv是阻塞的，开启新的线程
29             threading.Thread(target=self.recv, args=(sock, client)).start()
30
31     def recv(self, sock:socket.socket, client): # 接收客户端数据
32         while not self.event.is_set():
33             data = sock.recv(1024) # 阻塞到数据到来
34             msg = data.decode().strip()
35             # 客户端退出命令
36             if msg == 'quit' or msg == '': # 主动断开得到空串
37                 with self.lock:
38                     self.clients.pop(client)
39                     sock.close()
40                 logging.info('{} quits'.format(client))
41                 break
42             msg = "{:%Y/%m/%d %H:%M:%S} {}: \n{}\n".format(datetime.datetime.now(), *client, data.decode())
43             logging.info(msg)
44             msg = msg.encode()
45
46             with self.lock:
47                 for s in self.clients.values():
48                     s.send(msg)
49
50     def stop(self): # 停止服务
51         self.event.set()
52         with self.lock:
53             for s in self.clients.values():
54                 s.close()
55         self.sock.close()
56

```

```

57 cs = ChatServer()
58 cs.start()
59
60 while True:
61     cmd = input('>>').strip()
62     if cmd == 'quit':
63         cs.stop()
64         threading.Event().wait(3)
65         break
66     logging.info(threading.enumerate()) # 用来观察断开后线程的变化
67     logging.info(cs.clients)

```

也可以把recv和accept线程设置为daemon线程。

MakeFile

```

1 socket.makefile(mode='r', buffering=None, *, encoding=None, errors=None,
  newline=None)

```

创建一个与该套接字相关连的文件对象，将recv方法看做读方法，将send方法看做写方法。

```

1 # 使用makefile简单例子
2 import socket
3
4 server = socket.socket()
5 server.bind(('127.0.0.1', 9999))
6 server.listen()
7 print('-' * 30)
8
9 s, _ = server.accept()
10 print('-' * 30)
11 f = s.makefile(mode='rw')
12
13 line = f.read(10) # 按行读取要使用readline方法
14 print('-' * 30)
15 print(line)
16 f.write('return your msg: {}'.format(line))
17 f.flush()
18
19 f.close()
20 print(f.closed, s._closed)
21 s.close()
22 print(f.closed, s._closed)
23
24 server.close()

```

makefile练习

使用makefile改写群聊类

```

1 import logging
2 import socket
3 import threading
4 import datetime
5

```

```

6 logging.basicConfig(level=logging.INFO, format="%asctime)s %(thread)d %(
  (message)s")
7
8
9 class ChatServer:
10     def __init__(self, ip='127.0.0.1', port=9999): # 启动服务
11         self.sock = socket.socket()
12         self.addr = (ip, port)
13         self.clients = {} # 客户端
14         self.event = threading.Event()
15         self.lock = threading.Lock()
16
17     def start(self): # 启动监听
18         self.sock.bind(self.addr) # 绑定
19         self.sock.listen() # 监听
20         # accept会阻塞主线程，所以开一个新线程
21         threading.Thread(target=self.accept).start()
22
23     def accept(self): # 多人连接
24         while not self.event.is_set():
25             sock, client = self.sock.accept() # 阻塞
26             f = sock.makefile('rw') # 支持读写
27             with self.lock:
28                 self.clients[client] = f, sock # 添加到客户端字典
29             # 准备接收数据，recv是阻塞的，开启新的线程
30             threading.Thread(target=self.recv, args=(f, client)).start()
31
32     def recv(self, f, client): # 接收客户端数据
33         while not self.event.is_set():
34             data = f.readline() # 阻塞等一行来，换行符
35             msg = data.strip()
36             print(msg, '+++++')
37             # 客户端退出命令
38             if msg == 'quit' or msg == '': # 主动断开得到空串
39                 with self.lock:
40                     _, sock = self.clients.pop(client)
41                     sock.close()
42                     f.close()
43                     logging.info('{} quits'.format(client))
44                     break
45             msg = "{:%Y/%m/%d %H:%M:%S} {}:".
  {} \n {} \n".format(datetime.datetime.now(), *client, data)
46             logging.info(msg)
47
48             with self.lock:
49                 for ff, _ in self.clients.values():
50                     ff.write(msg)
51                     ff.flush()
52
53     def stop(self): # 停止服务
54         self.event.set()
55         with self.lock:
56             for f, s in self.clients.values():
57                 s.close()
58                 f.close()
59         self.sock.close()
60
61 cs = ChatServer()

```

```

62 cs.start()
63
64 while True:
65     cmd = input('>>').strip()
66     if cmd == 'quit':
67         cs.stop()
68         threading.Event().wait(3)
69         break
70     logging.info(threading.enumerate()) # 用来观察断开后线程的变化
71     logging.info(cs.clients)

```

上例完成了基本功能，但是，如果网络异常，或者readline出现异常，就不会从clients中移除作废的socket。可以使用异常处理解决这个问题。

ChatServer实验用完整代码

注意，这个代码为实验用，代码中瑕疵还有很多。Socket太底层了，实际开发中很少使用这么底层的接口。

增加一些异常处理。

```

1  import logging
2  import socket
3  import threading
4  import datetime
5
6  logging.basicConfig(level=logging.INFO, format="%asctime)s %(thread)d %
    (message)s")
7
8
9  class ChatServer:
10     def __init__(self, ip='127.0.0.1', port=9999): # 启动服务
11         self.sock = socket.socket()
12         self.addr = (ip, port)
13         self.clients = {} # 客户端
14         self.event = threading.Event()
15         self.lock = threading.Lock()
16
17     def start(self): # 启动监听
18         self.sock.bind(self.addr) # 绑定
19         self.sock.listen() # 监听
20         # accept会阻塞主线程，所以开一个新线程
21         threading.Thread(target=self.accept).start()
22
23     def accept(self): # 多人连接
24         while not self.event.is_set():
25             sock, client = self.sock.accept() # 阻塞
26             f = sock.makefile('rw') # 支持读写
27             with self.lock:
28                 self.clients[client] = f, sock # 添加到客户端字典
29             # 准备接收数据，recv是阻塞的，开启新的线程
30             threading.Thread(target=self.recv, args=(f, client)).start()
31
32     def recv(self, f, client): # 接收客户端数据
33         while not self.event.is_set():
34             try: # 异常处理
35                 data = f.readline() # 阻塞等一行来，换行符
36                 except Exception as e:

```

```

37         logging.error(e)
38         data = 'quit'
39
40         msg = data.strip()
41
42         # 客户端退出命令
43         if msg == 'quit' or msg == '': # 主动断开得到空串
44             with self.lock:
45                 _, sock = self.clients.pop(client)
46                 f.close()
47                 sock.close()
48                 logging.info('{} quits'.format(client))
49                 break
50             msg = "{:%Y/%m/%d %H:%M:%S} {}:
51 {}\\n{}\\n".format(datetime.datetime.now(), *client, data)
52             logging.info(msg)
53
54             with self.lock:
55                 for ff,_ in self.clients.values():
56                     ff.write(msg)
57                     ff.flush()
58
59         def stop(self): # 停止服务
60             self.event.set()
61             with self.lock:
62                 for f, s in self.clients.values():
63                     f.close()
64                     s.close()
65             self.sock.close()
66
67     def main():
68         cs = ChatServer()
69         cs.start()
70
71         while True:
72             cmd = input('>>').strip()
73             if cmd == 'quit':
74                 cs.stop()
75                 threading.Event().wait(3)
76                 break
77             logging.info(threading.enumerate()) # 用来观察断开后线程的变化
78             logging.info(cs.clients)
79
80 if __name__ == '__main__':
81     main()

```

TCP客户端编程

客户端编程步骤

- 创建Socket对象
- 连接到远端服务端的ip和port, connect()方法
- 传输数据

- 使用send、recv方法发送、接收数据
- 关闭连接，释放资源

```
1 import socket
2
3 client = socket.socket()
4 ipaddr = ('127.0.0.1', 9999)
5 client.connect(ipaddr) # 直接连接服务器
6
7 client.send(b'abcd\n')
8 data = client.recv(1024) # 阻塞等待
9 print(data)
10
11 client.close()
```

开始编写客户端类

```
1 import socket
2 import threading
3 import datetime
4 import logging
5
6 FORMAT = "%(asctime)s %(threadName)s %(thread)d %(message)s"
7 logging.basicConfig(format=FORMAT, level=logging.INFO)
8
9
10 class ChatClient:
11     def __init__(self, ip='127.0.0.1', port=9999):
12         self.sock = socket.socket()
13         self.addr = (ip, port)
14         self.event = threading.Event()
15
16     def start(self): # 启动对远端服务器的连接
17         self.sock.connect(self.addr)
18         self.send("I'm ready.")
19         # 准备接收数据，recv是阻塞的，开启新的线程
20         threading.Thread(target=self.recv, name="recv").start()
21
22     def recv(self): # 接收服务端的数据
23         while not self.event.is_set():
24             try:
25                 data = self.sock.recv(1024) # 阻塞
26                 except Exception as e:
27                     logging.error(e)
28                     break
29                 msg = "{:%Y/%m/%d %H:%M:%S} {}:".format(datetime.datetime.now(), *self.addr, data.strip())
30                 logging.info(msg)
31
32     def send(self, msg:str):
33         data = "{}\n".format(msg.strip()).encode() # 服务端需要一个换行符
34         self.sock.send(data)
35
36     def stop(self):
37         self.sock.close()
38         self.event.wait(3)
```

```
39         self.event.set()
40         logging.info('Client stops.')
41
42
43     def main():
44         cc = ChatClient()
45         cc.start()
46         while True:
47             cmd = input('>>>')
48             if cmd.strip() == 'quit':
49                 cc.stop()
50                 break
51             cc.send(cmd) # 发送消息
52
53 if __name__ == '__main__':
54     main()
```

同样，这样的客户端还是有些问题的，仅用于测试。