

网络编程

Socket介绍

Socket套接字

Python中提供socket.py标准库，非常底层的接口库。

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。

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编程

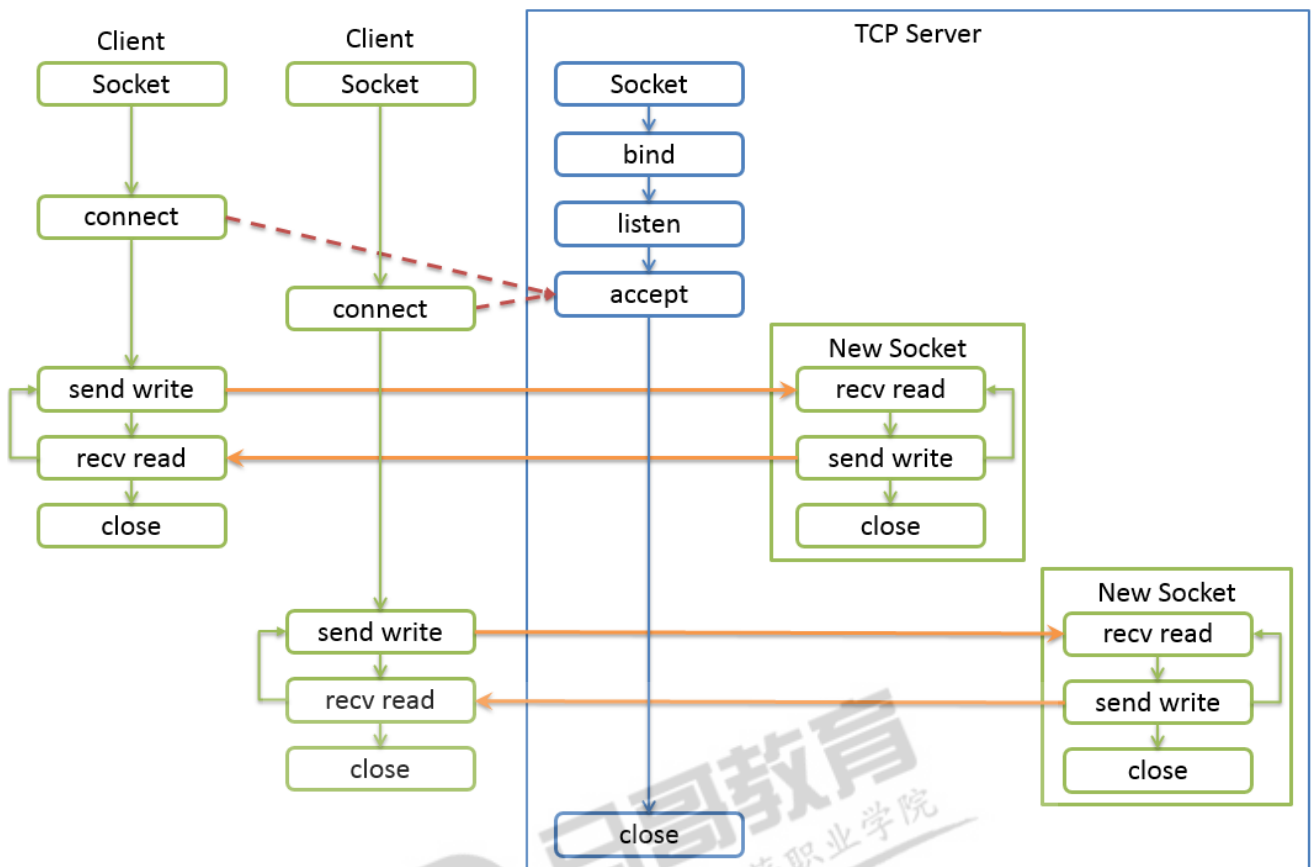
Socket编程，需要两端，一般来说需要一个服务端、一个客户端，服务端称为Server，客户端称为Client

TCP服务端编程

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



问题

两次绑定同一个监听端口会怎么样？

```
import socket

s = socket.socket() # 创建socket对象
s.bind(('127.0.0.1', 9999)) # 一个二元组
s.listen() # 开始监听
# 开启一个连接
s1, info = s.accept() # 阻塞直到和客户端成功建立连接，返回一个socket对象和客户端地址

# 使用缓冲区获取数据
data = s1.recv(1024)
print(data, info)
s1.send(b'magedu.com ack')

# 开启另外一个连接
s2, _ = s.accept()

data = s2.recv(1024)
s2.send(b'hello python')

s.close()
```

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

```
windows 命令
# netstat -anp tcp | findstr 9999

linux命令
# netstat -tanl | grep 9999
# 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

服务端应该对应一个类

```
class ChatServer:
    def __init__(self, ip, port): # 启动服务
        self.sock = socket.socket()
        self.addr = (ip, port)

    def start(self): # 启动监听
        pass

    def accept(self): # 多人连接
        pass

    def recv(self): # 接收客户端数据
        pass

    def stop(self): # 停止服务
        pass
```

在此基础上，扩展完成

```
import logging
import socket
import threading
```

```

import datetime

logging.basicConfig(level=logging.INFO, format="%(asctime)s %(thread)d %(message)s")

class ChatServer:
    def __init__(self, ip='127.0.0.1', port=9999): # 启动服务
        self.sock = socket.socket()
        self.addr = (ip, port)
        self.clients = {} # 客户端

    def start(self): # 启动监听
        self.sock.bind(self.addr) # 绑定
        self.sock.listen() # 监听
        # accept会阻塞主线程，所以开一个新线程
        threading.Thread(target=self.accept).start()

    def accept(self): # 多人连接
        while True:
            sock, client = self.sock.accept() # 阻塞
            self.clients[client] = sock # 添加到客户端字典
            # 准备接收数据，recv是阻塞的，开启新的线程
            threading.Thread(target=self.recv, args=(sock, client)).start()

    def recv(self, sock:socket.socket, client): # 接收客户端数据
        while True:
            data = sock.recv(1024) # 阻塞到数据到来
            msg = "{:%Y/%m/%d %H:%M:%S} {}:{}\n{}\n".format(datetime.datetime.now(), *client,
data.decode())
            logging.info(msg)
            msg = msg.encode()
            for s in self.clients.values():
                s.send(msg)

    def stop(self): # 停止服务
        for s in self.clients.values():
            s.close()
        self.sock.close()

cs = ChatServer()
cs.start()

```

基本功能完成，但是有问题。使用Event改进。

```

import logging
import socket
import threading
import datetime

logging.basicConfig(level=logging.INFO, format="%(asctime)s %(thread)d %(message)s")

```

```

class ChatServer:
    def __init__(self, ip='127.0.0.1', port=9999): # 启动服务
        self.sock = socket.socket()
        self.addr = (ip, port)
        self.clients = {} # 客户端
        self.event = threading.Event()

    def start(self): # 启动监听
        self.sock.bind(self.addr) # 绑定
        self.sock.listen() # 监听
        # accept会阻塞主线程, 所以开一个新线程
        threading.Thread(target=self.accept).start()

    def accept(self): # 多人连接
        while not self.event.is_set():
            sock, client = self.sock.accept() # 阻塞
            self.clients[client] = sock # 添加到客户端字典
            # 准备接收数据, recv是阻塞的, 开启新的线程
            threading.Thread(target=self.recv, args=(sock, client)).start()

    def recv(self, sock:socket.socket, client): # 接收客户端数据
        while not self.event.is_set():
            data = sock.recv(1024) # 阻塞到数据到来
            msg = "{:%Y/%m/%d %H:%M:%S} { }:{ }\n{ }\n".format(datetime.datetime.now(), *client,
data.decode())
            logging.info(msg)
            msg = msg.encode()
            for s in self.clients.values():
                s.send(msg)

    def stop(self): # 停止服务
        for s in self.clients.values():
            s.close()
        self.sock.close()
        self.event.set()

cs = ChatServer()
cs.start()

while True:
    cmd = input('>>').strip()
    if cmd == 'quit':
        cs.stop()
        threading.Event().wait(3)
        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

```
import logging
import socket
import threading
import datetime

logging.basicConfig(level=logging.INFO, format="%(asctime)s %(thread)d %(message)s")

class ChatServer:
    def __init__(self, ip='127.0.0.1', port=9999): # 启动服务
        self.sock = socket.socket()
        self.addr = (ip, port)
        self.clients = {} # 客户端
        self.event = threading.Event()

    def start(self): # 启动监听
        self.sock.bind(self.addr) # 绑定
        self.sock.listen() # 监听
        # accept会阻塞主线程，所以开一个新线程
        threading.Thread(target=self.accept).start()

    def accept(self): # 多人连接
        while not self.event.is_set():
            sock, client = self.sock.accept() # 阻塞
            self.clients[client] = sock # 添加到客户端字典
            # 准备接收数据，recv是阻塞的，开启新的线程
            threading.Thread(target=self.recv, args=(sock, client)).start()

    def recv(self, sock:socket.socket, client): # 接收客户端数据
        while not self.event.is_set():
            data = sock.recv(1024) # 阻塞到数据到来
            msg = data.decode().strip()
            # 客户端退出命令
            if msg == 'quit' or msg == '': # 主动断开得到空串
                self.clients.pop(client)
                sock.close()
                logging.info('{} quits'.format(client))
                break
            msg = "{:%Y/%m/%d %H:%M:%S} {}:{}\n{}\n".format(datetime.datetime.now(), *client,
data.decode())
            logging.info(msg)
            msg = msg.encode()
            for s in self.clients.values():
                s.send(msg)
```

```

def stop(self): # 停止服务
    for s in self.clients.values():
        s.close()
    self.sock.close()
    self.event.set()

cs = ChatServer()
cs.start()

while True:
    cmd = input('>>').strip()
    if cmd == 'quit':
        cs.stop()
        threading.Event().wait(3)
        break
    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

注意：

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。出错测试代码如下

```

import threading
import time, random

globaldict = {}

def add(d):
    count = 1
    while True:
        #print(count)
        d[count] = random.randint(1,10)
        count += 1
        time.sleep(0.001)

def iterdict(d):
    while True:
        for k,v in d.items(): # 会抛异常
            print(k,v)
            d[k] = random.randint(100, 110)

t1 = threading.Thread(target=add, args=(globaldict,), daemon=True)
t2 = threading.Thread(target=iterdict, args=(globaldict,), daemon=True)
t1.start()
t2.start()

while True:
    time.sleep(1)
    print(threading.enumerate())

```

```

if threading.active_count() == 2:
    print('=====end=====')
    break

```

socket常用方法

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
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

MakeFile

```

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。


```

# 使用makefile简单例子
import socket

sockserver = socket.socket()
ip = '127.0.0.1'
port = 9999
addr = (ip, port)
sockserver.bind(addr)
sockserver.listen()
print('-'*30)
s, _ = sockserver.accept()
print('-'*30)
f = s.makefile(mode='rw')

line = f.read(10) # 阻塞等
print('-'*30)
print(line)
f.write('Return your msg: {}'.format(line))
f.flush()

```

上例不能循环接收消息，修改一下

```

import socket
import threading

sockserver = socket.socket()
ip = '127.0.0.1'
port = 9999
addr = (ip, port)
sockserver.bind(addr)
sockserver.listen()
print('-'*30)

event = threading.Event()

def accept(sock:socket.socket, e:threading.Event):
    s, _ = sock.accept()
    f = s.makefile(mode='rw')

    while True:
        line = f.readline()
        print(line)
        if line.strip() == "quit": # 注意要发quit\n
            break
        f.write('Return your msg: {}'.format(line))
        f.flush()
    f.close()
    sock.close()
    e.wait(3)

t = threading.Thread(target=accept, args=(sockserver, event))

```

```
t.start()
t.join()

print(sockserver)
```

makefile练习

使用makefile改写群聊类

```
import logging
import socket
import threading
import datetime

logging.basicConfig(level=logging.INFO, format="%(asctime)s %(thread)d %(message)s")

class ChatServer:
    def __init__(self, ip='127.0.0.1', port=9999): # 启动服务
        self.sock = socket.socket()
        self.addr = (ip, port)
        self.clients = {} # 客户端
        self.event = threading.Event()

    def start(self): # 启动监听
        self.sock.bind(self.addr) # 绑定
        self.sock.listen() # 监听
        # accept会阻塞主线程，所以开一个新线程
        threading.Thread(target=self.accept).start()

    def accept(self): # 多人连接
        while not self.event.is_set():
            sock, client = self.sock.accept() # 阻塞
            # 准备接收数据，recv是阻塞的，开启新的线程
            f = sock.makefile('rw') # 支持读写
            self.clients[client] = f, sock # 添加到客户端字典
            threading.Thread(target=self.recv, args=(f, client), name='recv').start()

    def recv(self, f, client): # 接收客户端数据
        while not self.event.is_set():
            data = f.readline() # 阻塞到换行符
            msg = data.strip()
            # 客户端退出命令
            if msg == 'quit':
                _, s = self.clients.pop(client)
                f.close()
                s.close()
                logging.info('{} quits'.format(client))
                break
            msg = "{:%Y/%m/%d %H:%M:%S} {}:{}\n{}\n".format(datetime.datetime.now(), *client,
data)
            logging.info(msg)
```

```

        for ff,s in self.clients.values():
            ff.write(msg)
            ff.flush()

    def stop(self): # 停止服务
        for f,s in self.clients.values():
            f.close()
            s.close()
        self.sock.close()
        self.event.set()

cs = ChatServer()
cs.start()

while True:
    cmd = input('>>').strip()
    if cmd == 'quit':
        cs.stop()
        threading.Event().wait(3)
        break
    logging.info(threading.enumerate()) # 用来观察断开后线程的变化

```

上例完成了基本功能，但是，如果客户端主动断开，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。

```

import logging
import socket
import threading
import datetime

logging.basicConfig(level=logging.INFO, format="%(asctime)s %(thread)d %(message)s")

class ChatServer:
    def __init__(self, ip='127.0.0.1', port=9999): # 启动服务
        self.sock = socket.socket()
        self.addr = (ip, port)
        self.clients = {} # 客户端
        self.event = threading.Event()

    def start(self): # 启动监听
        self.sock.bind(self.addr) # 绑定
        self.sock.listen() # 监听
        # accept会阻塞主线程，所以开一个新线程
        threading.Thread(target=self.accept).start()

```

```

def accept(self): # 多人连接
    while not self.event.is_set():
        sock, client = self.sock.accept() # 阻塞
        # 准备接收数据, recv是阻塞的, 开启新的线程
        f = sock.makefile('rw') # 支持读写
        self.clients[client] = f # 添加到客户端字典
        threading.Thread(target=self.recv, args=(f, client), name='recv').start()

def recv(self, f, client): # 接收客户端数据
    while not self.event.is_set():
        try:
            data = f.readline() # 阻塞到换行符
        except Exception as e:
            logging.error(e) # 有任何异常, 退出
            data = 'quit'
        msg = data.strip()

        # 客户端退出命令
        if msg == 'quit':
            _, s = self.clients.pop(client)
            f.close()
            s.close()
            logging.info('{} quits'.format(client))
            break
        msg = "{:%Y/%m/%d %H:%M:%S} {}:{}\n{}\n".format(datetime.datetime.now(), *client,
data)
        logging.info(msg)

        for ff,s in self.clients.values():
            ff.write(msg)
            ff.flush()

def stop(self): # 停止服务
    for f,s in self.clients.values():
        f.close()
        s.close()
    self.sock.close()
    self.event.set()

def main():
    cs = ChatServer()
    cs.start()

    while True:
        cmd = input('>>').strip()
        if cmd == 'quit':
            cs.stop()
            threading.Event().wait(3)
            break
        logging.info(threading.enumerate()) # 用来观察断开后线程的变化

if __name__ == '__main__':

```

```
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

```
import socket

client = socket.socket()
ipaddr = ('127.0.0.1', 9999)
client.connect(ipaddr) # 直接连接服务器

client.send(b'abcd\n')
data = client.recv(1024) # 阻塞等待
print(data)

client.close()
```

开始编写客户端类

```
import socket
import threading
import datetime
import logging

FORMAT = "%(asctime)s %(threadName)s %(thread)d %(message)s"
logging.basicConfig(format=FORMAT, level=logging.INFO)

class ChatClient:
    def __init__(self, ip='127.0.0.1', port=9999):
        self.sock = socket.socket()
        self.addr = (ip, port)
        self.event = threading.Event()

    def start(self): # 启动对远端服务器的连接
        self.sock.connect(self.addr)
        self.send("I'm ready.")
        # 准备接收数据, recv是阻塞的, 开启新的线程
        threading.Thread(target=self.recv, name="recv").start()

    def recv(self): # 接收客户端的数据
```

```

while not self.event.is_set():
    try:
        data = self.sock.recv(1024) # 阻塞
    except Exception as e:
        logging.error(e)
        break
    msg = "{:%Y/%m/%d %H:%M:%S} {}:{}\n{}\n".format(datetime.datetime.now(), *self.addr,
data.strip())
    logging.info(msg)

def send(self, msg:str):
    data = "{}\n".format(msg.strip()).encode() # 服务端需要一个换行符
    self.sock.send(data)

def stop(self):
    self.sock.close()
    self.event.wait(3)
    self.event.set()
    logging.info('Client stops.')

def main():
    cc = ChatClient()
    cc.start()
    while True:
        cmd = input('>>>')
        if cmd.strip() == 'quit':
            cc.stop()
            break
        cc.send(cmd) # 发送消息

if __name__ == '__main__':
    main()

```

同样，这样的客户端还是有些问题的，仅用于测试。