要理解select.select模块其实主要就是要理解它的参数, 以及其三个返回值。select()方法接收并监控3个通信列表， 第一个是所有的输入的data,就是指外部发过来的数据，第2个是监控和接收所有要发出去的data(outgoing data),第3个监控错误信息  
  
在网上一直在找这个select.select的参数解释, 但实在是没有, 哎...自己硬着头皮分析了一下。  
readable, writable, exceptional = select.select(inputs, outputs, inputs)  
  
第一个参数就是服务器端的socket, 第二个是我们在运行过程中存储的客户端的socket, 第三个存储错误信息。  
重点是在返回值, 第一个返回的是可读的list, 第二个存储的是可写的list, 第三个存储的是错误信息的  
list。  
这个也不必深究, 看看代码自己分析下就能有大概理解。  
网上所有关于select.select的代码都是差不多的, 但是有些不能运行, 或是不全。我自己重新写了一份能运行的程序, 做了很多注释, 好好看看就能搞懂

服务器端:

# coding: utf-8

import select

import socket

import Queue

from time import sleep

# Create a TCP/IP

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

server.setblocking(False)

# Bind the socket to the port

server\_address = ('localhost', 8090)

print ('starting up on %s port %s' % server\_address)

server.bind(server\_address)

# Listen for incoming connections

server.listen(5)

# Sockets from which we expect to read

inputs = [server]

# Sockets to which we expect to write

# 处理要发送的消息

outputs = []

# Outgoing message queues (socket: Queue)

message\_queues = {}

while inputs:

# Wait for at least one of the sockets to be ready for processing

print ('waiting for the next event')

# 开始select 监听, 对input\_list 中的服务器端server 进行监听

readable, writable, exceptional = select.select(inputs, outputs, inputs)

# Handle inputs

# 循环判断是否有客户端连接进来, 当有客户端连接进来时select 将触发

for s in readable:

# 判断当前触发的是不是服务端对象, 当触发的对象是服务端对象时,说明有新客户端连接进来了

# 表示有新用户来连接

if s is server:

# A "readable" socket is ready to accept a connection

connection, client\_address = s.accept()

print ('connection from', client\_address)

# this is connection not server

connection.setblocking(0)

# 将客户端对象也加入到监听的列表中, 当客户端发送消息时 select 将触发

inputs.append(connection)

# Give the connection a queue for data we want to send

# 为连接的客户端单独创建一个消息队列，用来保存客户端发送的消息

message\_queues[connection] = Queue.Queue()

else:

# 有老用户发消息, 处理接受

# 由于客户端连接进来时服务端接收客户端连接请求，将客户端加入到了监听列表中(input\_list), 客户端发送消息将触发

# 所以判断是否是客户端对象触发

data = s.recv(1024)

# 客户端未断开

if data != '':

# A readable client socket has data

print ('received "%s" from %s' % (data, s.getpeername()))

# 将收到的消息放入到相对应的socket客户端的消息队列中

message\_queues[s].put(data)

# Add output channel for response

# 将需要进行回复操作socket放到output 列表中, 让select监听

if s not in outputs:

outputs.append(s)

else:

# 客户端断开了连接, 将客户端的监听从input列表中移除

# Interpret empty result as closed connection

print ('closing', client\_address)

# Stop listening for input on the connection

if s in outputs:

outputs.remove(s)

inputs.remove(s)

s.close()

# Remove message queue

# 移除对应socket客户端对象的消息队列

del message\_queues[s]

# Handle outputs

# 如果现在没有客户端请求, 也没有客户端发送消息时, 开始对发送消息列表进行处理, 是否需要发送消息

# 存储哪个客户端发送过消息

for s in writable:

try:

# 如果消息队列中有消息,从消息队列中获取要发送的消息

message\_queue = message\_queues.get(s)

send\_data = ''

if message\_queue is not None:

send\_data = message\_queue.get\_nowait()

else:

# 客户端连接断开了

print "has closed "

except Queue.Empty:

# 客户端连接断开了

print "%s" % (s.getpeername())

outputs.remove(s)

else:

# print "sending %s to %s " % (send\_data, s.getpeername)

# print "send something"

if message\_queue is not None:

s.send(send\_data)

else:

print "has closed "

# del message\_queues[s]

# writable.remove(s)

# print "Client %s disconnected" % (client\_address)

# # Handle "exceptional conditions"

# 处理异常的情况

for s in exceptional:

print ('exception condition on', s.getpeername())

# Stop listening for input on the connection

inputs.remove(s)

if s in outputs:

outputs.remove(s)

s.close()

# Remove message queue

del message\_queues[s]

sleep(1)

客户端:

# coding: utf-8

import socket

messages = ['This is the message ', 'It will be sent ', 'in parts ', ]

server\_address = ('localhost', 8090)

# Create aTCP/IP socket

socks = [socket.socket(socket.AF\_INET, socket.SOCK\_STREAM), socket.socket(socket.AF\_INET, socket.SOCK\_STREAM), ]

# Connect thesocket to the port where the server is listening

print ('connecting to %s port %s' % server\_address)

# 连接到服务器

for s in socks:

s.connect(server\_address)

for index, message in enumerate(messages):

# Send messages on both sockets

for s in socks:

print ('%s: sending "%s"' % (s.getsockname(), message + str(index)))

s.send(bytes(message + str(index)).decode('utf-8'))

# Read responses on both sockets

for s in socks:

data = s.recv(1024)

print ('%s: received "%s"' % (s.getsockname(), data))

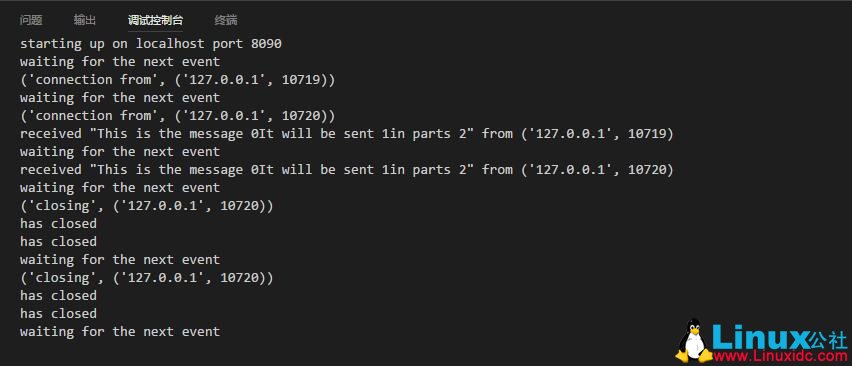
if data != "":

print ('closingsocket', s.getsockname())

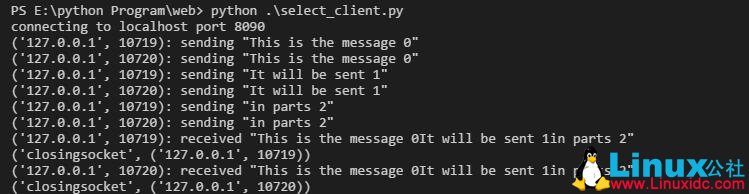
s.close()

写代码过程中遇到了两个问题, 一是如何判断客户端已经关闭了socket连接, 后来自己分析了下, 如果关闭了客户端socket, 那么此时服务器端接收到的data就是'', 加个这个判断。二是如果服务器端关闭了socket, 一旦在调用socket的相关方法都会报错, 不管socket是不是用不同的容器存储的(意思是说list\_1存储了socket1, list\_2存储了socket1, 我关闭了socket1, 两者都不能在调用这个socket了)

服务器端：



客户端：



[**python中的select模块**](https://www.cnblogs.com/MnCu8261/p/6403340.html)

**介绍：**

Python中的select模块专注于I/O多路复用，提供了select  poll  epoll三个方法(其中后两个在Linux中可用，windows仅支持select)，另外也提供了kqueue方法(freeBSD系统)

**select方法：**

进程指定内核监听哪些文件描述符(最多监听1024个fd)的哪些事件，当没有文件描述符事件发生时，进程被阻塞；当一个或者多个文件描述符事件发生时，进程被唤醒。

当我们调用select()时：

　　1 上下文切换转换为内核态

　　2 将fd从用户空间复制到内核空间

　　3  内核遍历所有fd，查看其对应事件是否发生

　　4  如果没发生，将进程阻塞，当设备驱动产生中断或者timeout时间后，将进程唤醒，再次进行遍历

　　5 返回遍历后的fd

　　6  将fd从内核空间复制到用户空间

fd:file descriptor 文件描述符

[复制代码](javascript:void(0);)

fd\_r\_list, fd\_w\_list, fd\_e\_list = select.select(rlist, wlist, xlist, [timeout])

参数： 可接受四个参数（前三个必须）  
rlist: wait until ready for reading  
wlist: wait until ready for writing  
xlist: wait for an “exceptional condition”  
timeout: 超时时间

返回值：三个列表

select方法用来监视文件描述符(当文件描述符条件不满足时，select会阻塞)，当某个文件描述符状态改变后，会返回三个列表

1、当参数1 序列中的fd满足“可读”条件时，则获取发生变化的fd并添加到fd\_r\_list中

2、当参数2 序列中含有fd时，则将该序列中所有的fd添加到 fd\_w\_list中

3、当参数3 序列中的fd发生错误时，则将该发生错误的fd添加到 fd\_e\_list中

4、当超时时间为空，则select会一直阻塞，直到监听的句柄发生变化

当超时时间 ＝ n(正整数)时，那么如果监听的句柄均无任何变化，则select会阻塞n秒，之后返回三个空列表，如果监听的句柄有变化，则直接执行。

[复制代码](javascript:void(0);)

实例：利用select实现一个可并发的服务端

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

[复制代码](javascript:void(0);)

import socket

import select

s = socket.socket()

s.bind(('127.0.0.1',8888))

s.listen(5)

r\_list = [s,]

num = 0

while True:

rl, wl, error = select.select(r\_list,[],[],10)

num+=1

print('counts is %s'%num)

print("rl's length is %s"%len(rl))

for fd in rl:

if fd == s:

conn, addr = fd.accept()

r\_list.append(conn)

msg = conn.recv(200)

conn.sendall(('first----%s'%conn.fileno()).encode())

else:

try:

msg = fd.recv(200)

fd.sendall('second'.encode())

except ConnectionAbortedError:

r\_list.remove(fd)

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

flag = 1

s = socket.socket()

s.connect(('127.0.0.1',8888))

while flag:

input\_msg = input('input>>>')

if input\_msg == '0':

break

s.sendall(input\_msg.encode())

msg = s.recv(1024)

print(msg.decode())

s.close()

[复制代码](javascript:void(0);)

在服务端我们可以看到，我们需要不停的调用select， 这就意味着：

**1  当文件描述符过多时，文件描述符在用户空间与内核空间进行copy会很费时**

**2  当文件描述符过多时，内核对文件描述符的遍历也很浪费时间**

**3  select最大仅仅支持1024个文件描述符**

poll与select相差不大，本文不作介绍

**epoll方法：**

epoll很好的改进了select：

**1  epoll的解决方案在epoll\_ctl函数中。每次注册新的事件到epoll句柄中时，会把所有的fd拷贝进内核，而不是在epoll\_wait的时候重复拷贝。epoll保证了每个fd在整个过程中只会拷贝一次。**

**2  epoll会在epoll\_ctl时把指定的fd遍历一遍（这一遍必不可少）并为每个fd指定一个回调函数，当设备就绪，唤醒等待队列上的等待者时，就会调用这个回调函数，而这个回调函数会把就绪的fd加入一个就绪链表。epoll\_wait的工作实际上就是在这个就绪链表中查看有没有就绪的fd**

**3  epoll对文件描述符没有额外限制**

select.epoll(sizehint=-1, flags=0) 创建epoll对象

[复制代码](javascript:void(0);)

epoll.close()

Close the control file descriptor of the epoll object.关闭epoll对象的文件描述符

epoll.closed

True if the epoll object is closed.检测epoll对象是否关闭

epoll.fileno()

Return the file descriptor number of the control fd.返回epoll对象的文件描述符

epoll.fromfd(fd)

Create an epoll object from a given file descriptor.根据指定的fd创建epoll对象

epoll.register(fd[, eventmask])

Register a fd descriptor with the epoll object.向epoll对象中注册fd和对应的事件

epoll.modify(fd, eventmask)

Modify a registered file descriptor.修改fd的事件

epoll.unregister(fd)

Remove a registered file descriptor from the epoll object.取消注册

epoll.poll(timeout=-1, maxevents=-1)

Wait for events. timeout in seconds (float)阻塞，直到注册的fd事件发生,会返回一个dict，格式为：{(fd1,event1),(fd2,event2),……(fdn,eventn)}

[复制代码](javascript:void(0);)

事件：

[复制代码](javascript:void(0);)

EPOLLIN Available for read 可读 状态符为1

EPOLLOUT Available for write 可写 状态符为4

EPOLLPRI Urgent data for read

EPOLLERR Error condition happened on the assoc. fd 发生错误 状态符为8

EPOLLHUP Hang up happened on the assoc. fd 挂起状态

EPOLLET Set Edge Trigger behavior, the default is Level Trigger behavior 默认为水平触发，设置该事件后则边缘触发

EPOLLONESHOT Set one-shot behavior. After one event is pulled out, the fd is internally disabled

EPOLLRDNORM Equivalent to EPOLLIN

EPOLLRDBAND Priority data band can be read.

EPOLLWRNORM Equivalent to EPOLLOUT

EPOLLWRBAND Priority data may be written.

EPOLLMSG Ignored.

[复制代码](javascript:void(0);)

水平触发和边缘触发：

Level\_triggered(水平触发，有时也称条件触发)：当被监控的文件描述符上有可读写事件发生时，epoll.poll()会通知处理程序去读写。如果这次没有把数据一次性全部读写完(如读写缓冲区太小)，那么下次调用 epoll.poll()时，它还会通知你在上没读写完的文件描述符上继续读写，当然如果你一直不去读写，它会一直通知你！！！如果系统中有大量你不需要读写的就绪文件描述符，而它们每次都会返回，这样会大大降低处理程序检索自己关心的就绪文件描述符的效率！！！ 优点很明显：稳定可靠

Edge\_triggered(边缘触发，有时也称状态触发)：当被监控的文件描述符上有可读写事件发生时，epoll.poll()会通知处理程序去读写。如果这次没有把数据全部读写完(如读写缓冲区太小)，那么下次调用epoll.poll()时，它不会通知你，也就是它只会通知你一次，直到该文件描述符上出现第二次可读写事件才会通知你！！！这种模式比水平触发效率高，系统不会充斥大量你不关心的就绪文件描述符！！！缺点：某些条件下不可靠

epoll实例：

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

[复制代码](javascript:void(0);)

import socket

import select

s = socket.socket()

s.bind(('127.0.0.1',8888))

s.listen(5)

epoll\_obj = select.epoll()

epoll\_obj.register(s,select.EPOLLIN)

connections = {}

while True:

events = epoll\_obj.poll()

for fd, event in events:

print(fd,event)

if fd == s.fileno():

conn, addr = s.accept()

connections[conn.fileno()] = conn

epoll\_obj.register(conn,select.EPOLLIN)

msg = conn.recv(200)

conn.sendall('ok'.encode())

else:

try:

fd\_obj = connections[fd]

msg = fd\_obj.recv(200)

fd\_obj.sendall('ok'.encode())

except BrokenPipeError:

epoll\_obj.unregister(fd)

connections[fd].close()

del connections[fd]

s.close()

epoll\_obj.close()

[复制代码](javascript:void(0);)

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

[复制代码](javascript:void(0);)

import socket

flag = 1

s = socket.socket()

s.connect(('127.0.0.1',8888))

while flag:

input\_msg = input('input>>>')

if input\_msg == '0':

break

s.sendall(input\_msg.encode())

msg = s.recv(1024)

print(msg.decode())

s.close()

[复制代码](javascript:void(0);)