select/selectors模块

操作系统提供了一个功能，当你的某个socket可读或者可写的时候，它可以给你一个通知。这样当配合非阻塞的socket使用时，只有当系统通知我哪个描述符可读了，才去执行read操作，可以保证每次read都能读到有效数据而不做纯返回-1之类无用功。写操作类似。操作系统的这个功能通过支持I/O多路复用的系统调用来使用，这些函数都可以同时监视多个描述符的读写就绪状况，这样，多个描述符的I/O操作都能在一个线程内并发交替地顺序完成，这就叫I/O多路复用，这里的「复用」指的是复用同一个线程。

IO多路复用适用场景

1. 当客户处理多个描述符时（一般是交互式输入和网络套接字），必须使用I/O复用
2. 当一个客户同时处理多个套接字时，而这种情况是可能的，但很少出现
3. 如果一个TCP服务器既要处理监听套接字，又要处理已连接套接字，一般也要用到I/O复用
4. 如果一个服务器即要处理TCP，又要处理UDP，一般要使用I/O复用
5. 如果一个服务器要处理多个服务或多个协议，一般要使用I/O复用

支持I/O多路复用的系统调用

1. select
2. poll
3. epoll
4. kqueue

```python
import socket
import select
from queue import Queue, Empty

HOST = '127.0.0.1'
PORT = 8001

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.setblocking(0) # 设置为非阻塞
server.bind((HOST, PORT))  # 绑定套接字到本地IP与端口
server.listen(5)  # 监听连接
inputs = [server]
outputs = []
message_queues = {}
print(f'Server start at: {HOST}:{PORT}')

while inputs:
    readable, writable, exceptional = select.select(inputs, outputs, inputs)
    for s in readable:
        if s is server:
            conn, addr = s.accept()  # 接受客户端连接
            print(f'Connected by {addr}')
            conn.setblocking(0)  # 设置连接为非阻塞
            inputs.append(conn)
            message_queues[conn] = Queue()
        else:
            data = s.recv(1024)  # 接收1024字节的内容
            if data:
                print(f'received "{data}" from {s.getpeername()}')
                message_queues[s].put(data)
                if s not in outputs:
                    outputs.append(s)
            else:
                if s in outputs:
                    outputs.remove(s)
                inputs.remove(s)
                del message_queues[s]
                s.close()
    for s in writable:
        try:
            next_msg = message_queues[s].get_nowait()
        except Empty:
            outputs.remove(s)
        else:
            s.send(bytes(f'Server received {next_msg}', 'utf-8'))
    for s in exceptional:
        inputs.remove(s)
        if s in outputs:
            outputs.remove(s)
        s.close()
        del message_queues[s]
```

1. select

客户端例子(client.py)

```
import socket

HOST = '127.0.0.1'
PORT = 8001

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

socks = [
    socket.socket(socket.AF_INET, socket.SOCK_STREAM),
    socket.socket(socket.AF_INET, socket.SOCK_STREAM),
]

print(f'connecting to {HOST} port {PORT}')

for s in socks:
    s.connect((HOST, PORT))

for index, message in enumerate(messages):
    _, is_odd = divmod(index, 2)
    outgoing_data = message.encode()

    for index, s in enumerate(socks):
        if divmod(index, 2)[1] != is_odd:
            continue
        print(f'{s.getsockname()}: sending {outgoing_data}')
        s.send(outgoing_data)

    for index, s in enumerate(socks):
        if divmod(index, 2)[1] != is_odd:
            continue
        data = s.recv(1024)
        print(f'{s.getsockname()}: received {data}')
        if not data:
            s.close()
```

1. 启动服务端

```
❯ python select_server.py
Server start at: 127.0.0.1:8001
Connected by ('127.0.0.1', 50442)
Connected by ('127.0.0.1', 50443)
received "b'This is '" from ('127.0.0.1', 50442)
received "b'the message. '" from ('127.0.0.1', 50443)
received "b'It will be sent '" from ('127.0.0.1', 50442)
received "b'in parts.'" from ('127.0.0.1', 50443)
```

2. 启动客户端

```
❯ python client.py
connecting to 127.0.0.1 port 8001
('127.0.0.1', 50442): sending b'This is '
('127.0.0.1', 50442): received b"Server received b'This is '"
('127.0.0.1', 50443): sending b'the message. '
('127.0.0.1', 50443): received b"Server received b'the message. '"
('127.0.0.1', 50442): sending b'It will be sent '
('127.0.0.1', 50442): received b"Server received b'It will be sent '"
('127.0.0.1', 50443): sending b'in parts.'
('127.0.0.1', 50443): received b"Server received b'in parts.'"
```

select有3个缺点

1. 单个进程所打开的文件描述符数量是有一定限制的，通常默认值是1024。 这对于高并发的网络服务来说太小了
2. 对socket进行扫描时是线性扫描，即采用轮询的方法，效率较低。看代码可知，套接字数量多时浪费很多CPU时间
3. 需要维护一个用来存放大量的数据结构，这样会使得用户空间和内核空间在传递该结构时复制开销大

2. poll

```
import socket
import select
from queue import Queue, Empty

HOST = '127.0.0.1'
PORT = 8001

server = socket.socket(socket.AF_INET,
                       socket.SOCK_STREAM)
server.setblocking(0)
server.bind((HOST, PORT))
server.listen(5)

message_queues = {}
TIMEOUT = 500  # 超时时间0.5秒
_to_socket = {  # 一个文件描述符到套接字的映射
    server.fileno(): server,
}
READ_ONLY = (
    select.POLLIN |
    select.POLLPRI |
    select.POLLHUP |
    select.POLLERR
) # 4种事件的并集
READ_WRITE = READ_ONLY | select.POLLOUT

poller = select.poll()
# 给server套接字注册，它会关注READ_ONLY列出的4种事件
poller.register(server, READ_ONLY)

print(f'Server start at: {HOST}:{PORT}')
```

```
while 1:
    events = poller.poll(TIMEOUT)
    for , flag in events:
        s = _to_socket[]
        if flag & (select.POLLIN | select.POLLPRI):  # 输入准备就绪了，也就是可读了
            if s is server:
                conn, addr = s.accept()
                print(f'Connected by {addr}')
                conn.setblocking(0)
                _to_socket[conn.fileno()] = conn
                poller.register(conn, READ_ONLY)  # 新注册的套接字都关注READ_ONLY事件
                message_queues[conn] = Queue()
            else:
                data = s.recv(1024)
                if data:
                    print(f'received "{data}" from {s.getpeername()}')
                    message_queues[s].put(data)
                    poller.modify(s, READ_WRITE)  # 从缓冲区获取内容后，也关注POLLOUT事件了
                else:
                    poller.unregister(s)  # 没有可用数据的套接字说明客户端关闭了，取消注册
                    s.close()
        elif flag & select.POLLHUP:  # 套接字关闭了
            poller.unregister(s)
            s.close()
        elif flag & select.POLLOUT:  # 能够输出了，也就是可写了
            try:
                next_msg = message_queues[s].get_nowait()
            except Empty:
                # 修改套接字关注的时间类型，因为它已经恢复不可写状态了
                poller.modify(s, READ_ONLY)
            else:
                s.send(bytes(f'Server received {next_msg}', 'utf-8'))
        elif flag & select.POLLERR:  # 错误的套接字
            poller.unregister(s)
            s.close()
            del message_queues[s]
```

2. poll

POLLIN | 有数据读取
POLLPRI | 有优先级数据读取
POLLOUT | 能够输出
POLLHUP | 挂起
POLLERR | 错误
POLLNVAL |套接字未打开

水平触发 Level Triggered (LT) / 边缘触发 Edge Triggered (ET)

3. epoll

1. epoll支持的上限是最大可以打开文件的数目

2. epoll的解决方法不像select和poll每次对所有进行遍历轮询所有集合，而是在注册新的事件时，为每个指定一个回调函数，当设备就绪的时候，调用这个回调函数，这个回调函数就会把就绪的加入一个就绪表中。（所以epoll实际只需要遍历就绪表）

3. epoll的解决方法是每次注册新的事件到epoll中，会把所有的拷贝进内核，而不是在等待的时候重复拷贝，保证了每个在整个过程中只会拷贝1次

```python
import socket
import select
from queue import Queue, Empty

HOST = '127.0.0.1'
PORT = 8001

server = socket.socket(socket.AF_INET,
                       socket.SOCK_STREAM)
server.setblocking(0)
server.bind((HOST, PORT))
server.listen(5)

message_queues = {}
TIMEOUT = 500

epoller = select.epoll()
# epoller = select.kqueue()
epoller.register(server, select.EPOLLIN)

print(f'Server start at: {HOST}:{PORT}')
```

```python
while 1:
    events = epoller.poll(TIMEOUT)
    for , flag in events:
        if  == server.fileno():
            conn, addr = s.accept()
            print(f'Connected by {addr}')
            conn.setblocking(0)
            epoller.register(conn, select.EPOLLIN)
            message_queues[conn] = Queue()
        if flag & (select.POLLIN | select.POLLPRI):
            data = s.recv(1024)
            if data:
                print(f'received "{data}" from {s.getpeername()}')
                message_queues[s].put(data)
                epoller.modify(s, select.EPOLLOUT)
            else:
                epoller.unregister(s)
                s.close()
        elif flag & select.POLLHUP:
            epoller.unregister(s)
            s.close()
        elif flag & select.POLLOUT:
            try:
                next_msg = message_queues[s].get_nowait()
            except Empty:
                epoller.modify(s, select.EPOLLIN)
            else:
                s.send(bytes(f'Server received {next_msg}', 'utf-8'))
        elif flag & select.POLLERR:
            poller.unregister(s)
            s.close()
            del message_queues[s]
```

selectors模块

1. SelectSelector
2. PollSelector
3. EpollSelector
4. DevpollSelector
5. KqueueSelector

事件

1. EVENT_READ 可读
2. EVENT_WRITE 可写

```python
import socket
import selectors
from queue import Queue, Empty

HOST = '127.0.0.1'
PORT = 8001

sock = socket.socket(socket.AF_INET,
                     socket.SOCK_STREAM)
sock.setblocking(0)
sock.bind((HOST, PORT))
sock.listen(5)

sel = selectors.DefaultSelector()
message_queues = {}

print(f'Server start at: {HOST}:{PORT}')

sel.register(
    sock,
    selectors.EVENT_READ | selectors.EVENT_WRITE,
)
```

```python
while 1:
    for key, mask in sel.select(timeout=0.5):
        conn = key.fileobj
        if conn is sock:
            conn, addr = sock.accept()
            print(f'Connected by {addr}')
            conn.setblocking(0)
            message_queues[conn] = Queue()
            sel.register(
                conn, selectors.EVENT_READ | selectors.EVENT_WRITE)
        elif mask & selectors.EVENT_READ:
            data = conn.recv(1024)
            if data:
                print(f'received "{data}" from {conn.getpeername()}')
                message_queues[conn].put(data)
        elif mask & selectors.EVENT_WRITE:
            try:
                next_msg = message_queues[conn].get_nowait()
            except Empty:
                pass
            else:
                conn.send(bytes(f'Server received {next_msg}', 'utf-8'))
                sel.modify(sock, selectors.EVENT_READ) # 从可写切换到可读状态
```

```python
import socket
import selectors

HOST = '127.0.0.1'
PORT = 8001

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.setblocking(0)
sock.bind((HOST, PORT))
sock.listen(5)

sel = selectors.DefaultSelector()

print(f'Server start at: {HOST}:{PORT}')

def read(conn, mask):
    data = conn.recv(1024)
    if data:
        print(f'received "{data}" from {conn.getpeername()}')
        conn.send(bytes(f'Server received {data}', 'utf-8'))
    else:
        sel.unregister(conn)
        conn.close()


def accept(sock, mask):
    conn, addr = sock.accept()
    print(f'Connected by {addr}')
    conn.setblocking(0)
    sel.register(conn, selectors.EVENT_READ, read)


sel.register(sock, selectors.EVENT_READ, accept)

while 1:
    events = sel.select(0.5)
    for key, mask in events:
        callback = key.data
        callback(key.fileobj, mask)
```

延伸阅读

1. 《UNIX网络编程卷1：套接字联网API》 第6章
2. https://pymotw.com/3/select/
3. https://www.jianshu.com/p/d940e7fca2
4. https://pymotw.com/3/selectors/
5. https://docs.python.org/3/library/selectors.html
6. https://docs.python.org/3/library/select.html