原创

# python之IO多路复用（二）——select、poll、epoll详解

忘情OK 关注

2017-02-06 18:45:27 2649人阅读 4人评论

select，poll，epoll都是IO多路复用的机制。I/O多路复用就是通过一种机制使一个进程可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。

select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的

异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间。

**sellect、poll、epoll三者的区别：**

**select：**

目前支持几乎所有的平台

默认单个进程能够监视的文件描述符的数量存在最大限制，在linux上默认只支持1024个socket

可以通过修改宏定义或重新编译内核（修改系统最大支持的端口数）的方式提升这一限制

内核准备好数据后通知用户有数据了，但不告诉用户是哪个连接有数据，用户只能通过轮询的方式来获取数据

假定select让内核监视100个socket连接，当有1个连接有数据后，内核就通知用户100个连接中有数据了

但是不告诉用户是哪个连接有数据了，此时用户只能通过轮询的方式一个个去检查然后获取数据

这里是假定有100个socket连接，那么如果有上万个，上十万个呢?

那你就得轮询上万次，上十万次，而你所取的结果仅仅就那么1个。这样就会浪费很多没用的开销

只支持水平触发

每次调用select，都需要把fd集合从用户态拷贝到内核态，这个开销在fd很多时会很大

同时每次调用select都需要在内核遍历传递进来的所有fd，这个开销在fd很多时也会很大

**poll：**

与select没有本质上的差别，仅仅是没有了最大文件描述符数量的限制

只支持水平触发

只是一个过渡版本，很少用

**epoll：**

Linux2.6才出现的epoll，具备了select和poll的一切优点，公认为性能最好的多路IO就绪通知方法

没有最大文件描述符数量的限制

同时支持水平触发和边缘触发

IO效率不随fd数目增加而线性下降

使用mmap加速内核与用户空间的消息传递

**水平触发与边缘触发：**

水平触发：将就绪的文件描述符告诉进程后，如果进程没有对其进行IO操作，那么下次调用epoll时将再次报告这些文件描述符，这种方式称为水平触发

边缘触发：只告诉进程哪些文件描述符刚刚变为就绪状态，它只说一遍，如果我们没有采取行动，那么它将不会再次告知，这种方式称为边缘触发

理论上边缘触发的性能要更高一些，但是代码实现相当复杂。

**select和epoll的特点：**

select：

select通过一个select()系统调用来监视多个文件描述符的数组，当select()返回后，该数组中就绪的文件描述符便会被内核修改标志位，使得进程可以获得这些文件描述符从而进行后续的读写操作。

由于网络响应时间的延迟使得大量TCP连接处于非活跃状态，但调用select()会对所有socket进行一次线性扫描，所以这也浪费了一定的开销。

epoll:

epoll同样只告知那些就绪的文件描述符，而且当我们调用epoll_wait()获得就绪文件描述符时，返回的不是实际的描述符，而是一个代表就绪描述符数量的值，你只需要去epoll指定的一个数组中依次取得相应数量的文件描述符即可，这里也使用了内存映射（mmap）技术，这样便彻底省掉了这些文件描述符在系统调用时复制的开销。

另一个本质的改进在于epoll采用基于事件的就绪通知方式。在select/poll中，进程只有在调用一定的方法后，内核才对所有监视的文件描述符进行扫描，而epoll事先通过epoll_ctl()来注册一个文件描述符，一旦基于某个文件描述符就绪时，内核会采用类似callback的回调机制，迅速激活这个文件描述符，当进程调用epoll_wait()时便得到通知。

**select**

```
select(rlist, wlist, xlist, timeout=None)
```

select函数监视的文件描述符分3类，分别是writefds、readfds、和exceptfds。

调用后select函数会阻塞，直到有描述符就绪（有数据可读、可写、或者有except），或者超时（timeout指定等待时间，如果立即返回设为null即可），函数返回。当select函数返回后，可以通过遍历fdset，来找到就绪的描述符。

**poll**

```
int poll (struct pollfd *fds, unsigned int nfds, int timeout);
```

不同于select使用三个位图来表示三个fdset的方式，poll使用一个pollfd的指针实现。

```
struct pollfd {
    int fd; /* file descriptor */
    short events; /* requested events to watch */
    short revents; /* returned events witnessed */
};
```

pollfd结构包含了要监视的event和发生的event，不再使用select“参数-值”传递的方式。

同时，pollfd并没有最大数量限制（但是数量过大后性能也是会下降）。

和select函数一样，poll返回后，需要轮询pollfd来获取就绪的描述符。

从上面看，select和poll都需要在返回后，通过遍历文件描述符来获取已经就绪的socket。

事实上，同时连接的大量客户端在一时刻可能只有很少的处于就绪状态，因此随着监视的描述符数量的增长，其效率也会线性下降。

### epoll

epoll是在2.6内核中提出的，是之前的select和poll的增强版本。相对于select和poll来说，epoll更加灵活，没有描述符限制。

epoll使用一个文件描述符管理多个描述符，将用户关系的文件描述符的事件存放到内核的一个事件表中，这样在用户空间和内核空间的copy只需一次。

**epoll操作过程**

epoll操作过程需要三个接口，分别如下：

```
int epoll_create(int size);//创建一个epoll的句柄，size用来告诉内核这个监听的数目一共有多大
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout);
```

1. int epoll_create(int size);

创建一个epoll的句柄，size用来告诉内核这个监听的数目一共有多大，这个参数不同于select()中的第一个参数，给出最大监听的fd+1的值，参数size并不是限制了epoll所能监听的描述符最大个数，只是对内核初始分配内部数据结构的一个建议。

当创建好epoll句柄后，它就会占用一个fd值，在linux下如果查看/proc/进程id/fd/，是能够看到这个fd的，所以在使用完epoll后，必须调用close()关闭，否则可能导致fd被耗尽。

2. int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);

函数是对指定描述符fd执行op操作。

epfd：是epoll_create()的返回值。

op：表示op操作，用三个宏来表示：

添加EPOLL_CTL_ADD，删除EPOLL_CTL_DEL，修改EPOLL_CTL_MOD。

分别添加、删除和修改对fd的监听事件。

fd：是需要监听的fd（文件描述符）

epoll_event：是告诉内核需要监听什么事

3. int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout);

等待epfd上的io事件，最多返回maxevents个事件。

参数events用来从内核得到事件的集合，maxevents告之内核这个events有多大，这个maxevents的值不能大于创建epoll_create()时的size，参数timeout是超时时间（毫秒，0会立即返回，-1将不确定，也有说法说是永久阻塞）。该函数返回需要处理的事件数目，如返回0表示已超时。

一个简单的select多并发socket服务端代码如下：

```
#!/usr/bin/python
#Author:sean

import select
import socket
import queue

server = socket.socket()
HOST = 'localhost'
```

```
PORT = 8080
print("start up %s on port: %s",% (HOST,PORT))
server.bind((HOST,PORT))
server.listen()

server.setblocking(False)    #不阻塞

msg_dic_queue = {}     #这是一个队列字典，存放要返回给客户端的数据

inputs = [server]    #inputs里存放要让内核监测的连接，这里的server是指监测server本身的连接状态
#inputs = [server,conn]
outputs = []     #outputs里存放要返回给客户端的数据连接对象

while True:
    print("waiting for next connect...")
    readable,writeable,exceptional = select.select(inputs,outputs,inputs)    #如果没有任何fd就绪，程
    # print(readable,writeable,exceptional)
    for r in readable:  #处理活跃的连接，每个r就是一个socket连接对象
        if r is server: #代表来了一个新连接
            conn,client_addr = server.accept()
            print("arrived a new connect: ",client_addr)
            conn.setblocking(False)
            inputs.append(conn) #因为这个新建立的连接还没发数据来，现在就接收的话，程序就报异常
            #所以要想实现这个客户端发数据来时server端能知道，就需要让select再监测这个conn
            msg_dic_queue[conn] = queue.Queue()    #初始化一个队列，后面存要返回给客户端的数据
        else:    #r不是server的话就代表是一个与客户端建立的文件描述符了
            #客户端的数据过来了，在这里接收
            data = r.recv(1024)
            if data:
                print("received data from [%s]: "% r.getpeername()[0],data)
                msg_dic_queue[r].put(data)  #收到的数据先放到队列字典里，之后再返回给客户端
                if r not in outputs:
                    outputs.append(r)    #放入返回的连接队列里。为了不影响处理与其它客户端的连接，
            else:    #如果收不到data就代表客户端已经断开了
                print("Client is disconnect",r)
                if r in outputs:
                    outputs.remove(r)    #清理已断开的连接
                inputs.remove(r)
                del msg_dic_queue[r]

    for w in writeable: #处理要返回给客户端的连接列表
        try:
            next_msg = msg_dic_queue[w].get_nowait()
        except queue.Empty:
            print("client [%s]"% w.getpeername()[0],"queue is empty...")
            outputs.remove(w)    #确保下次循环时writeable不返回已经处理完的连接
        else:
            print("sending message to [%s]"% w.getpeername()[0],next_msg)
            w.send(next_msg)     #返回给客户端源数据

    for e in exceptional:    #处理异常连接
        if e in outputs:
            outputs.remove(e)
        inputs.remove(e)
        del msg_dic_queue[e]
```

select多并发socket客户端代码如下：

```
#!/usr/bin/python
#Author:sean

import socket

msgs = [ b'This is the message. ',
```

```
                b'It will be sent ',
                b'in parts.',
                ]
SERVER_ADDRESS = 'localhost'
SERVER_PORT = 8080

# Create a few TCP/IP socket
socks = [ socket.socket(socket.AF_INET, socket.SOCK_STREAM) for i in range(500) ]

# Connect the socket to the port where the server is listening
print('connecting to %s port %s' % (SERVER_ADDRESS,SERVER_PORT))
for s in socks:
    s.connect((SERVER_ADDRESS,SERVER_PORT))

for message in msgs:

    # Send messages on both sockets
    for s in socks:
        print('%s: sending "%s"' % (s.getsockname(), message) )
        s.send(message)

    # Read responses on both sockets
    for s in socks:
        data = s.recv(1024)
        print( '%s: received "%s"' % (s.getsockname(), data) )
        if not data:
            print(sys.stderr, 'closing socket', s.getsockname() )
```

epoll多并发socket服务端代码如下：

```
#!/usr/bin/python
#Author:sean

import socket, logging
import select, errno

logger = logging.getLogger("network-server")

def InitLog():
    logger.setLevel(logging.DEBUG)

    fh = logging.FileHandler("network-server.log")
    fh.setLevel(logging.DEBUG)
    ch = logging.StreamHandler()
    ch.setLevel(logging.ERROR)

    formatter = logging.Formatter("%(asctime)s - %(name)s - %(levelname)s - %(message)s")
    ch.setFormatter(formatter)
    fh.setFormatter(formatter)

    logger.addHandler(fh)
    logger.addHandler(ch)


if __name__ == "__main__":
    InitLog()

    try:
        # 创建 TCP socket 作为监听 socket
        listen_fd = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)
    except socket.error as  msg:
        logger.error("create socket failed")
```

```
try:
    # 设置 SO_REUSEADDR 选项
    listen_fd.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
except socket.error as msg:
    logger.error("setsocketopt SO_REUSEADDR failed")

try:
    # 进行 bind -- 此处未指定 ip 地址，即 bind 了全部网卡 ip 上
    listen_fd.bind(('', 8008))
except socket.error as msg:
    logger.error("bind failed")

try:
    # 设置 listen 的 backlog 数
    listen_fd.listen(10)
except socket.error as msg:
    logger.error(msg)

try:
    # 创建 epoll 句柄
    epoll_fd = select.epoll()
    # 向 epoll 句柄中注册 监听 socket 的 可读 事件
    epoll_fd.register(listen_fd.fileno(), select.EPOLLIN)
except select.error as msg:
    logger.error(msg)

connections = {}
addresses = {}
datalist = {}
while True:
    # epoll 进行 fd 扫描的地方 -- 未指定超时时间则为阻塞等待
    epoll_list = epoll_fd.poll()

    for fd, events in epoll_list:
        # 若为监听 fd 被激活
        if fd == listen_fd.fileno():
            # 进行 accept -- 获得连接上来 client 的 ip 和 port，以及 socket 句柄
            conn, addr = listen_fd.accept()
            logger.debug("accept connection from %s, %d, fd = %d" % (addr[0], addr[1], conn.fil
            # 将连接 socket 设置为 非阻塞
            conn.setblocking(0)
            # 向 epoll 句柄中注册 连接 socket 的 可读 事件
            epoll_fd.register(conn.fileno(), select.EPOLLIN | select.EPOLLET)
            # 将 conn 和 addr 信息分别保存起来
            connections[conn.fileno()] = conn
            addresses[conn.fileno()] = addr
        elif select.EPOLLIN & events:
            # 有 可读 事件激活
            datas = ''
            while True:
                try:
                    # 从激活 fd 上 recv 10 字节数据
                    data = connections[fd].recv(10)
                    # 若当前没有接收到数据，并且之前的累计数据也没有
                    if not data and not datas:
                        # 从 epoll 句柄中移除该 连接 fd
                        epoll_fd.unregister(fd)
                        # server 侧主动关闭该 连接 fd
                        connections[fd].close()
                        logger.debug("%s, %d closed" % (addresses[fd][0], addresses[fd][1]))
                        break
                    else:
                        # 将接收到的数据拼接保存在 datas 中
```

```
                datas += data
            except socket.error as  msg:
                # 在 非阻塞 socket 上进行 recv 需要处理 读穿 的情况
                # 这里实际上是利用 读穿 出 异常 的方式跳到这里进行后续处理
                if msg.errno == errno.EAGAIN:
                    logger.debug("%s receive %s" % (fd, datas))
                    # 将已接收数据保存起来
                    datalist[fd] = datas
                    # 更新 epoll 句柄中连接d 注册事件为 可写
                    epoll_fd.modify(fd, select.EPOLLET | select.EPOLLOUT)
                    break
                else:
                    # 出错处理
                    epoll_fd.unregister(fd)
                    connections[fd].close()
                    logger.error(msg)
                    break
        elif select.EPOLLHUP & events:
            # 有 HUP 事件激活
            epoll_fd.unregister(fd)
            connections[fd].close()
            logger.debug("%s, %d closed" % (addresses[fd][0], addresses[fd][1]))
        elif select.EPOLLOUT & events:
            # 有 可写 事件激活
            sendLen = 0
            # 通过 while 循环确保将 buf 中的数据全部发送出去
            while True:
                # 将之前收到的数据发回 client -- 通过 sendLen 来控制发送位置
                sendLen += connections[fd].send(datalist[fd][sendLen:])
                # 在全部发送完毕后退出 while 循环
                if sendLen == len(datalist[fd]):
                    break
            # 更新 epoll 句柄中连接 fd 注册事件为 可读
            epoll_fd.modify(fd, select.EPOLLIN | select.EPOLLET)
        else:
            # 其他 epoll 事件不进行处理
            continue
```

**代码好多？想用更简单的代码来实现同样的效果？请往这儿走**



python　select　poll　Python

8　收藏　分享

上一篇：python之IO多路复用　下一篇：python之selectors...

忘情OK

83篇文章，18W+人气，22粉丝

关注