互联网程序设计课程报告

张义飞

201821080630

1 实验要求

Question 1

利用IO复用+非阻塞技术实现web集群,master利用http重定向将流量分散到slave服务器上。

2 实验原理

2.1 IO多路复用技术

IO多路复用是指内核一旦发现进程指定的一个或者多个IO条件准备读取，它就通知该进程。IO多路复用适用如下场合:

- 当客户处理多个描述符时（一般是交互式输入和网络套接口），必须使用I/O复用。
- 当一个客户同时处理多个套接口时，而这种情况是可能的，但很少出现。
- 如果一个TCP服务器既要处理监听套接口，又要处理已连接套接口，一般也要用到I/O复用。
- 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。
- 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。

与多进程和多线程技术相比，I/O多路复用技术的最大优势是系统开销小，系统不必创建进程/线程，也不必维护这些进程/线程，从而大大减小了系统的开销。目前支持I/O多路复用的系统调用有 select，pselect，poll，epoll，I/O多路复用就是通过一种机制，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。但select，pselect，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，而异步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。

2.1.1 select简介

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。

select的工作流程：单个进程就可以同时处理多个网络连接的io请求（同时阻塞多个io操作）。基本原理就是程序呼叫select，然后整个程序就阻塞了，这时候，kernel就会轮询检查所有select负责的fd，当找到一个client中的数据准备好了，select就会返回，这个时候程序就会系统调用，将数据从kernel复制到进程缓冲区。

2.1.2 poll简介

poll的原理与select非常相似，差别如下:

- 描述fd集合的方式不同，poll使用 pollfd 结构而不是select结构fd_set结构，所以poll是链式的，没有最大连接数的限制;
- poll有一个特点是水平触发，也就是通知程序fd就绪后，这次没有被处理，那么下次poll的时候会再次通知同个fd已经就绪。

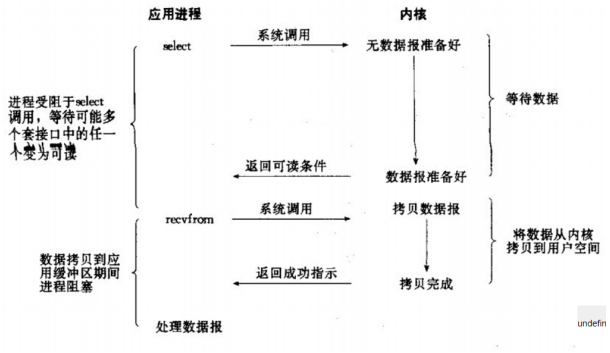

Figure 1: select工作流程

2.1.3 epoll简介

epoll在Linux2.6内核正式提出，是基于事件驱动的I/O方式，相对于select和poll来说，epoll没有描述符个数限制，使用一个文件描述符管理多个描述符，将用户关心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。优点如下：

1. 没有最大并发连接的限制，能打开的fd上限远大于1024（1G的内存能监听约10万个端口）
2. 采用回调的方式，效率提升。只有活跃可用的fd才会调用callback函数，也就是说 epoll 只管你“活跃”的连接，而跟连接总数无关，因此在实际的网络环境中，epoll的效率就会远远高于select和poll。
3. 内存拷贝。使用mmap()文件映射内存来加速与内核空间的消息传递，减少复制开销。

epoll对文件描述符的操作有两种模式：LT(level trigger，水平触发)和ET(edge trigger)。

水平触发：默认工作模式，即当epoll_wait检测到某描述符事件就绪并通知应用程序时，应用程序可以不立即处理该事件；下次调用epoll_wait时，会再次通知此事件。

边缘触发：当epoll_wait检测到某描述符事件就绪并通知应用程序时，应用程序必须立即处理该事件。如果不处理，下次调用epoll_wait时，不会再次通知此事件。（直到你做了某些操作导致该描述符变成未就绪状态了，也就是说边缘触发只在状态由未就绪变为就绪时通知一次）。

ET模式很大程度上减少了epoll事件的触发次数，因此效率比LT模式下高。

2.2 HTTP重定向

HTTP重定向响应报文状态码一般为3xx，这类状态码代表需要客户端采取进一步的操作才能完成请求。通常，这些状态码用来重定向，后续的请求地址（重定向目标）在本次响应的Location域中指明。

当且仅当后续的请求所使用的方法是GET或者HEAD时，用户浏览器才可以在没有用户介入的情况下自动提交所需要的后续请求。客户端应当自动监测无限循环重定向（例如：A→B→C→.....→A或A→A），因为这会导致服务器和客户端大量不必要的资源消耗。按照HTTP/1.0版规范的建议，浏览器不应自动访问超过5次的重定向。

2.2.1 300 Multiple Choices

被请求的资源有一系列可供选择的回馈信息，每个都有自己特定的地址和浏览器驱动的商议信息。用户或浏览器能够自行选择一个首选的地址进行重定向。除非这是一个HEAD请求，否则该响应应当包括一个资源特性及地址的列表的实体，以便用户或浏览器从中选择最合适的重定向地址。这个实体的格式由Content-Type定义的格式所决定。浏览器可能根据响应的格式以及浏览器自身能力，自动作出最合适的选择。当然，RFC 2616规范并没有规定这样的自动选择该如何进行。如果服务器本身已经有了首选的回馈选择，那么在Location中应当指明这个回馈的URI；浏览器可能会将这个Location值作为自动重定向的地址。此外，除非额外指定，否则这个响应也是可缓存的。

2.2.2 301 Moved Permanently

被请求的资源已永久移动到新位置，并且将来任何对此资源的引用都应该使用本响应返回的若干个URI之一。如果可能，拥有链接编辑功能的客户端应当自动把请求的地址修改为从服务器反馈回来的地址。除非额外指定，否则这个响应也是可缓存的。新的永久性的URI应当在响应的Location域中返回。除非这是一个HEAD请求，否则响应的实体中应当包含指向新的URI的超链接及简短说明。如果这不是一

个GET或者HEAD请求，那么浏览器禁止自动进行重定向，除非得到用户的确认，因为请求的条件可能因此发生变化。

2.2.3 302 Found

要求客户端执行临时重定向（原始描述短语为“Moved Temporarily”）。由于这样的重定向是临时的，客户端应当继续向原有地址发送以后的请求。只有在Cache-Control或Expires中进行了指定的情况下，这个响应才是可缓存的。新的临时性的URI应当在响应的Location域中返回。除非这是一个HEAD请求，否则响应的实体中应当包含指向新的URI的超链接及简短说明。如果这不是一个GET或者HEAD请求，那么浏览器禁止自动进行重定向，除非得到用户的确认，因为请求的条件可能因此发生变化。

2.2.4 303 See Other

对应当前请求的响应可以在另一个URI上被找到，当响应于POST（或PUT / DELETE）接收到响应时，客户端应该假定服务器已经收到数据，并且应该使用单独的GET消息发出重定向。这个方法的存在主要是为了允许由脚本激活的POST请求输出重定向到一个新的资源。这个新的URI不是原始资源的替代引用。同时，303响应禁止被缓存。当然，第二个请求（重定向）可能被缓存。新的URI应当在响应的Location域中返回。除非这是一个HEAD请求，否则响应的实体中应当包含指向新的URI的超链接及简短说明。

2.2.5 304 Not Modified

表示资源在由请求头中的If-Modified-Since或If-None-Match参数指定的这一版本之后，未曾被修改。在这种情况下，由于客户端仍然具有以前下载的副本，因此不需要重新传输资源。

3 实验结果

3.1 实验环境

本实验在Linux环境下使用python完成，主要使用python自带的socket库调用epoll函数处理客户端访问，主服务器维护各子服务器的状态，通过HTTP重定向将客户端请求分流到各个子服务器。

3.2 结果展示

```
2019-05-31 18:05:50,563 - Socket Logger - DEBUG - accept connection from 127.0.0.1, 40338, fd = 19
2019-05-31 18:06:30,451 - Socket Logger - DEBUG - accept connection from 127.0.0.1, 40342, fd = 19
2019-05-31 18:06:30,452 - Socket Logger - DEBUG - 19 receive b'test data\n'
2019-05-31 18:06:30,452 - Socket Logger - DEBUG - 127.0.0.1, 40342 closed
```

Figure 2: epoll服务端工作日志

图2展示了使用epoll多路复用技术实现的服务端的工作记录，这个服务端会简单地将客户端发来的数据原封不动地发送回去。

图3展示了web服务集群的架构设计，在实现时，为了模拟子服务器集群，在本地启动了若干个socket服务器，并通过不同的端口进行区分，由主服务器维护每个子服务器的流量，每次有新客户端访问时，主服务器将在子服务器列表中查询空闲服务器，并返回一个HTTP 301重定向请求，使得客户端访问空闲的目标服务器，从而实现负载均衡的目的。

从图4可以看到，多次通过客户端访问主服务器，每次都会返回不同的301重定向地址，这表明主服务器的分流作用已实现，达到了负载均衡的目的。

附录代码[1]

```
import socket, logging
import select, errno

logger = logging.getLogger('Socket␣Logger')
```

[1]本项目文件托管在：https://github.com/Yidadaa/Web-Application-Design-2019

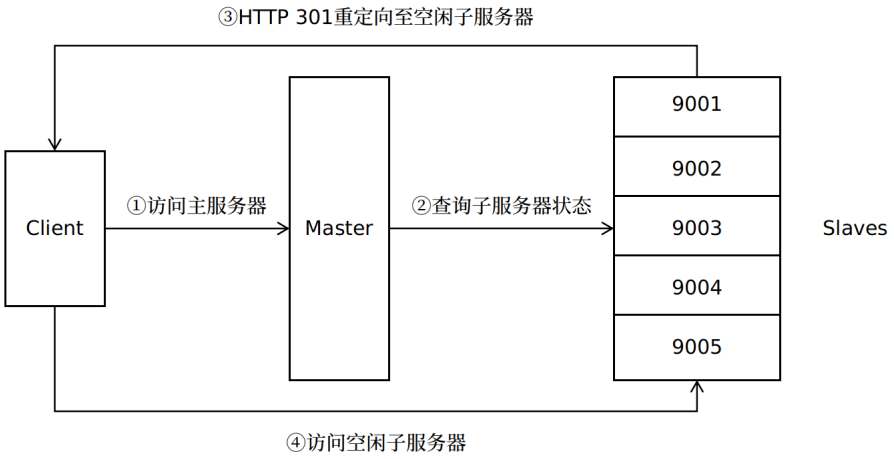

Figure 3: web服务集群架构设计

```
(base) yida@yida-PC:~/Desktop/Web-Application-Design-2019/src$ python client.py
Sent: test data
Received: HTTP/1.1 301 Moved Permanently
Location:http://127.0.0.1:9000
(base) yida@yida-PC:~/Desktop/Web-Application-Design-2019/src$ python client.py
Sent: test data
Received: HTTP/1.1 301 Moved Permanently
Location:http://127.0.0.1:9001
(base) yida@yida-PC:~/Desktop/Web-Application-Design-2019/src$ python client.py
Sent: test data
Received: HTTP/1.1 301 Moved Permanently
Location:http://127.0.0.1:9002
(base) yida@yida-PC:~/Desktop/Web-Application-Design-2019/src$ python client.py
Sent: test data
Received: HTTP/1.1 301 Moved Permanently
Location:http://127.0.0.1:9003
(base) yida@yida-PC:~/Desktop/Web-Application-Design-2019/src$ python client.py
Sent: test data
Received: HTTP/1.1 301 Moved Permanently
Location:http://127.0.0.1:9004
(base) yida@yida-PC:~/Desktop/Web-Application-Design-2019/src$ python client.py
Sent: test data
Received: HTTP/1.1 301 Moved Permanently
Location:http://127.0.0.1:9000
```

Figure 4: web服务集群分流

```
def InitLog():
    logger.setLevel(logging.DEBUG)
    fh = logging.FileHandler('Socket.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)

def HttpResp(port=9000):
    headers = 'HTTP/1.1 301 Moved Permanently\nLocation:\
        http://127.0.0.1:{}'.format(port)
```

```
    return bytes(headers, 'utf-8')

if __name__ == "__main__":
    InitLog()

    HOST, PORT = '127.0.0.1', 9000

    try:
        listen_fd = socket.socket(socket.AF_INET, socket.SOCK_STREAM, 0)
    except socket.error as msg:
        logger.error('create␣socket␣failed.')

    try:
        listen_fd.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    except socket.error as msg:
        logger.error('setsocketopt␣reuseaddr␣failed')

    try:
        listen_fd.bind((HOST, PORT))
    except socket.error as msg:
        logger.error('bind␣failed')

    try:
        listen_fd.listen(10)
    except socket.error as msg:
        logger.error(msg)

    try:
        #   epoll
        epoll_fd = select.epoll()
        #   epoll       socket
        epoll_fd.register(listen_fd.fileno(), select.EPOLLIN)
    except select.error as  msg:
        logger.error(msg)

    connections = {}
    addresses = {}
    datalist = {}
    serverlist = [9000, 9001, 9002, 9003, 9004]

    index = 0

    while True:
        # epoll   fd     --
        epoll_list = epoll_fd.poll()

        for fd, events in epoll_list:
            #    fd
            if fd == listen_fd.fileno():
                #   accept --     client   ip    port socket
                conn, addr = listen_fd.accept()
                logger.debug("accept␣connection␣from␣%s,␣%d,␣fd␣=␣%d" % (addr[0]
                    , addr[1], conn.fileno()))
                #    socket
                conn.setblocking(0)
                #   epoll       socket
                epoll_fd.register(conn.fileno(), select.EPOLLIN |
                    select.EPOLLET)
                #   conn   addr
                connections[conn.fileno()] = conn
                addresses[conn.fileno()] = addr
```

```
elif select.EPOLLIN & events:
    #
    datas = b''
    while True:
        try:
            #    fd   recv 10
            data = connections[fd].recv(10)
            #
            if not data and not datas:
                #   epoll       fd
                epoll_fd.unregister(fd)
                # server       fd
                connections[fd].close()
                logger.debug("%s, %d closed" % (addresses[fd][0],
                    addresses[fd][1]))
                break
            else:
                #        datas
                datas += data
        except socket.error as  msg:
            #     socket    recv
            #
            if msg.errno == errno.EAGAIN:
                logger.debug("%s receive %s" % (fd, datas))
                #
                datalist[fd] = datas
                #   epoll    d
                epoll_fd.modify(fd, select.EPOLLET |
                    select.EPOLLOUT)
                break
            else:
                #
                epoll_fd.unregister(fd)
                connections[fd].close()
                logger.error(msg)
                break
elif select.EPOLLHUP & events:
    #   HUP
    epoll_fd.unregister(fd)
    connections[fd].close()
    logger.debug("%s, %d closed" % (addresses[fd][0],
        addresses[fd][1]))
elif select.EPOLLOUT & events:
    connections[fd].send(HttpResp(serverlist[index]))
    #   epoll     fd
    epoll_fd.modify(fd, select.EPOLLIN | select.EPOLLET)

    index += 1
    index %= len(serverlist)
else:
    #   epoll
    continue
```