# 课程介绍

* 1. 传统并发服务器;（掌握）
* 2. 多路复用类并发;（掌握）

1. 传统并发服务器
   1. 服务器的工作方式

服务器可工作在两种不同的方式：

**循环方式(iterative mode)**：计算机一次只运行一个服务器进程。当有多个客户进程请求服务时，服务器进程就按请求的先后顺序依次做出响应。

**并发方式(concurrent)**：计算机同时运行多个服务器进程，而每一个服务器进程都对某个特定的客户进程做出响应。

* 1. 无连接的循环服务器

使用无连接的 UDP 的服务器通常都工作在循环方式，其主要特点是： 一个服务器在同一时间只能向一个客户提供服务。

服务器收到客户的请求后，就发送 UDP 用户数据报响应该客户。但对其他客户发来的请求则暂时不予理睬，这些请求都在服务器端的队列中排队等候服务器的处理。

当服务器进程处理完毕一个请求时，就从队列中读取来自下一个客户的请求，然后继续处理。

* 1. 面向连接的并发服务器

TCP 是面向连接的，因此在服务器和多个客户之间必须建立多条 TCP 连接，而每一条 TCP 连接要在其数据传送完毕后才能释放。

使用 TCP 的服务器只能有一个熟知端口。因此主服务器在熟知端口等待客户发出的请求。一旦收到客户的请求，就立即创建一个从属服务器，并指明从属服务器使用临时端口和该客户建立 TCP 连接，然后主服务器继续在原来的熟知端口等待向其他客户提供服务。

我们之前所写的服务器模型，都是单任务服务器，同一时间，只能处理一条请求，如果这个请求没有释放，那么其他连接就得不到服务。

从传统意义来讲，只要能够让每条请求（事务）用操作系统提供的调度器来进行调度就可以达到并发访问了。

* 1. 多进程并发服务器

利用进程把客户端和服务器的连接进行管理，这样监听socket对象用一个进程来管理，通过操作系统的进程调度，来达到了并发操作。

import socket

from multiprocessing import Process

def init\_listen\_sock(port, maxfd=3):

tsock = socket.socket()

tsock.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

tsock.bind(('', port))

tsock.listen(maxfd)

return tsock

# 子进程处理新连接

def process\_handler(nsock):

raw\_data = nsock.recv(1024)

while raw\_data:

print(f"收到{raw\_data.decode('utf-8')} 内容")

raw\_data = nsock.recv(1024)

nsock.close()

def main\_handler():

listen\_sock = init\_listen\_sock(16688)

while True:

nsock, \_ = listen\_sock.accept()

print(f"一个新的连接{nsock}")

client\_pid = Process(target=process\_handler, args=(nsock,))

client\_pid.start()

nsock.close()

if \_\_name\_\_ == '\_\_main\_\_':

main\_handler()

* 1. 多线程并发服务器

多线程和多进程类似，只是利用了线程进行任务调度，注意全局变量的保护即可。

import socket

import threading

def init\_listen\_sock(port, maxfd=3):

tsock = socket.socket()

tsock.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

tsock.bind(('', port))

tsock.listen(maxfd)

return tsock

# 子线程处理新连接

def thread\_handler(nsock):

raw\_data = nsock.recv(1024)

while raw\_data:

print(f"收到{raw\_data.decode('utf-8')} 内容")

raw\_data = nsock.recv(1024)

nsock.close()

def main\_handler():

listen\_sock = init\_listen\_sock(16688)

while True:

nsock, \_ = listen\_sock.accept()

print(f"一个新的连接{nsock}")

client\_pid = threading.Thread(target=thread\_handler, args=(nsock,))

client\_pid.start()

# nsock.close()

if \_\_name\_\_ == '\_\_main\_\_':

main\_handler()

1. 多路复用IO
   1. select系统调用

在操作系统层面上，提供了一个叫做select的系统调用接口，他的目的是把IO作为集合方式，进行内核层面上的轮询来判断每个IO是否具备可读可写可执行的功能。

select最早于1983年出现在4.2BSD中，它通过一个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。

* 1. python的select服务器模型

python提供了操作系统底层select接口的封装版本，使用起来会更加方便。

select(rlist, wlist, xlist[, timeout]) -> (rlist, wlist, xlist)

rlist wlist xlist都是列表数据结构。返回的列表则是对应的可读表，可写表，有异常表。

import socket

import select

def init\_listen\_sock(port, maxfd=3):

tsock = socket.socket()

tsock.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

tsock.bind(('', port))

tsock.listen(maxfd)

return tsock

def select\_modle():

listen\_sock = init\_listen\_sock(16688)

input\_lists = [listen\_sock]

while True:

rtab, \_, \_ = select.select(input\_lists, [], [])

for sock in rtab:

if listen\_sock is sock:

# 可以被监听

nsock, \_ = sock.accept()

print(f"接收到新连接{nsock}")

input\_lists.append(nsock)

else:

# 开始接收到客户端的数据

raw\_data = sock.recv(1024)

if raw\_data:

print(f"收到{raw\_data.decode('utf-8')} 内容")

else:

# 客户关闭连接

sock.close()

input\_lists.remove(sock)

if \_\_name\_\_ == '\_\_main\_\_':

select\_modle()

**【优点】：**

* select目前几乎在所有的平台上支持，其良好跨平台支持也是它的一个优点。

**【缺点】：**

* select的一个缺点在于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上一般为1024，可以通过修改宏定义甚至重新编译内核的方式提升这一限制，但是这样也会造成效率的降低。
* 一般来说这个数目和系统内存关系很大，具体数目可以cat /proc/sys/fs/file-max察看。32位机默认是1024个。64位机默认是2048.
* 对socket进行扫描时是依次扫描的，即采用轮询的方法，效率较低。

当套接字比较多的时候，每次select()都要通过遍历FD\_SETSIZE个Socket来完成调度，不管哪个Socket是活跃的，都遍历一遍。这会浪费很多CPU时间。

**【注意事项】：**

* 为了改进最大连接数的限制，1986年实现了poll机制，本质上没有什么变化，只是没有最大文件描述符数量的限制。
* select()和poll()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select()和poll()的时候将再次报告这些文件描述符，所以它们一般不会丢失就绪的消息，这种方式称为水平触发（Level Triggered）。
  1. python的epoll服务器模型

为了解决select和poll轮询机制的效率问题，从Linux2.6内核开始，引入了epoll接口。

1、没有最大并发连接的限制，能打开的文件描述符的上限远大于1024。

2、效率提升，不是轮询的方式，而是采用了事件通知机制，当文件描述符配置为epoll方式后，一旦事件发生，设备驱动会主动通知内核，此时应用空间会得到通知。

3、应用空间的处理机制几乎不变，主要是内核结构的修改。

import socket

import select

def init\_listen\_sock(port, maxfd=3):

tsock = socket.socket()

tsock.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)

tsock.bind(('', port))

tsock.listen(maxfd)

return tsock

def select\_modle():

listen\_sock = init\_listen\_sock(16688)

epoll = select.epoll()

epoll.register(listen\_sock.fileno(), select.EPOLLIN)

client\_socks = {}

while True:

epoll\_list = epoll.poll()

for fd, events in epoll\_list:

if listen\_sock.fileno() == fd:

# 可以被监听

nsock, \_ = listen\_sock.accept()

print(f"接收到新连接{nsock}")

epoll.register(nsock.fileno(), select.EPOLLIN)

client\_socks[nsock.fileno()] = nsock

else:

# 开始接收到客户端的数据

csock = client\_socks[fd]

raw\_data = csock.recv(1024)

if raw\_data:

print(f"收到{raw\_data.decode('utf-8')} 内容")

else:

# 客户关闭连接

csock.close()

epoll.unregister(fd)

if \_\_name\_\_ == '\_\_main\_\_':

select\_modle()

epoll对文件描述符的操作有两种模式：LT（level trigger）和ET（edge trigger）。LT模式是默认模式，LT模式与ET模式的区别如下：

LT模式：当epoll检测到描述符事件发生并将此事件通知应用程序，应用程序可以不立即处理该事件。下次调用epoll时，会再次响应应用程序并通知此事件。

ET模式：当epoll检测到描述符事件发生并将此事件通知应用程序，应用程序必须立即处理该事件。如果不处理，下次调用epoll时，不会再次响应应用程序并通知此事件。

1. 课程总结
   1. 重点
      * 1. 掌握多进程并发的方法
        2. 掌握多线程并发的方法
        3. 了解select/epoll的使用方法
   2. 难点
      * 1. 并发模型的设计
2. 课后练习
   * + 1. 仿照课堂案例，写一个多对1的聊天服务器和客户端；
3. 面试题
   * + 1. 谈谈select和epoll的区别。
4. 扩展知识或课外阅读推荐（可选）