

TCP编程

socket编程

Python中提供了socket标准库，非常底层的接口库。Socket是一种通用的网络编程接口，和网络层次没有一一对应的关系。

协议族 AF表示Address Family，用于socket()第一个参数

名称	含义
AF_INET	IPV4
AF_INET6	IPV6
AF_UNIX	Unix Domain Socket，windows没有

Socket类型

名称	含义
SOCK_STREAM	面向连接的流套接字。默认值，TCP协议
SOCK_DGRAM	无连接的数据报文套接字。UDP协议

TCP协议是流协议，也就是一大段数据看做字节流，一段段持续发送这些字节。

UDP协议是数据报协议，每一份数据封在一个单独的数据报中，一份一份发送数据。

注意：一开始学习网络编程，不要陷入协议的细节中。

CS编程

Socket编程，是完成一端和另一端通信的，注意一般来说这两端分别处在不同的进程中，也就是说网络通信是一个进程发消息到另外一个进程。

我们写代码的时候，每一个socket对象只表示了其中的一端。

从业务角度来说，这两端从角色上分为：

- 主动发送请求的一端，称为客户端Client
- 被动接受请求并回应的一端，称为服务端Server

这种编程模式也称为**C/S编程**。

TCP服务端编程

服务器端编程步骤

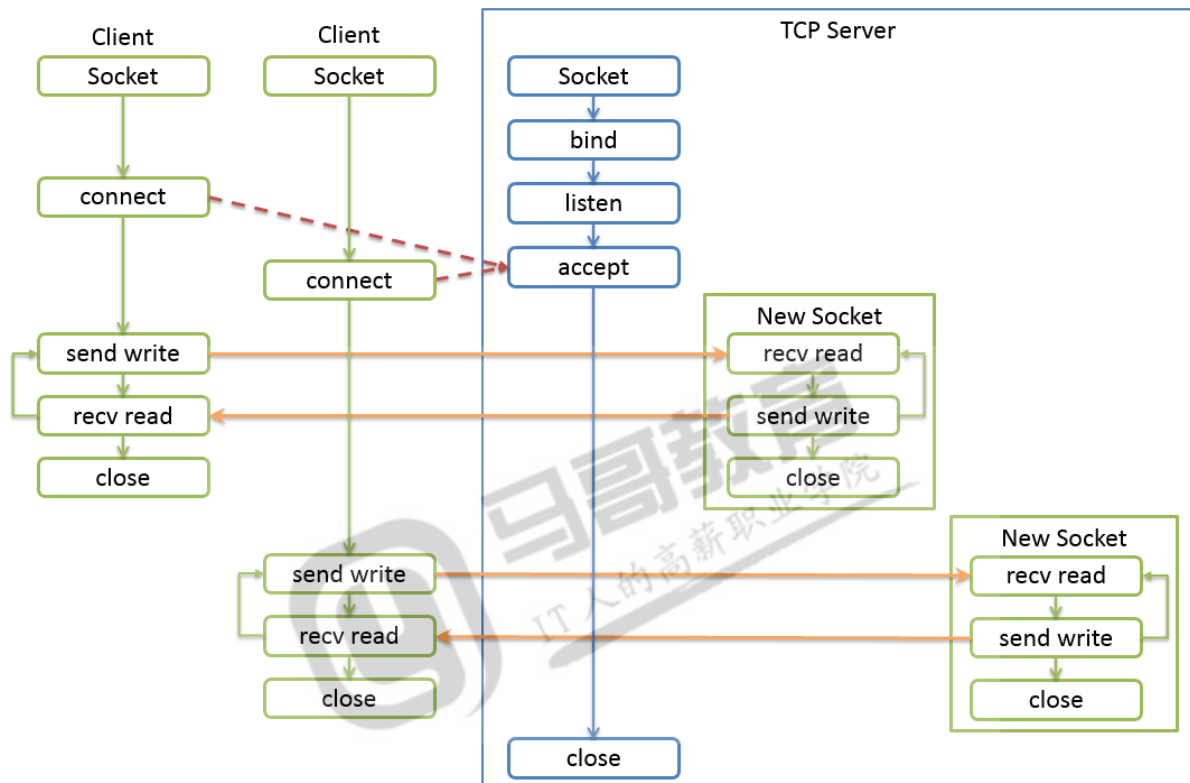
- 创建Socket对象
- 绑定IP地址Address和端口Port。bind()方法 IPv4地址为一个二元组('IP地址字符串', Port)
- 开始监听，将在指定的IP的端口上监听

- listen([backlog])方法。未完成连接队列和完成连接队列长度不能超过backlog，如果accept不拿走就满了，就会直接拒绝连接请求。backlog可以不写，默认为5
- 获取用于传送数据的**新的**Socket对象 socket.accept() -> (socket object, address info) accept方法阻塞等待客户端建立连接，返回一个新的Socket对象和客户端地址的二元组 地址是远程客户端的地址，IPv4中它是一个二元组(clientaddr, port)
 - 接收数据 recv(bufsize[, flags]) 使用缓冲区接收数据
 - 发送数据 send(bytes)发送数据

```

1 Server端开发
2 socket对象 --> bind((IP, PORT)) --> listen --> accept --> close
3 |--> recv or send --> close

```



```

1 # 最简单的服务器例子
2 import socket
3
4 # TCP服务端编程
5 server = socket.socket() # 创建socket对象
6 laddr = ('0.0.0.0', 9999) # 地址和端口的元组
7 server.bind(laddr) # 绑定
8 server.listen(1024) # 监听
9 # 等待建立连接的客户端
10 conn, raddr = server.accept() # 阻塞
11
12 data = conn.recv(4096) # 接收客户端信息
13 print(conn.getpeername(), data)
14 conn.send(b"Hello magedu.com") # 回应客户端
15
16 conn.close()
17 server.close()

```

实战：实现WEB服务器——多线程阻塞IO版

```
1  import threading
2  import time
3  import socket
4
5  html = """\
6  <!DOCTYPE html>
7  <html lang="en">
8  <head>
9      <meta charset="UTF-8">
10     <title>magedu</title>
11 </head>
12 <body>
13     <h1>马哥教育www.magedu.com -- Multithread + Blocking IO</h1>
14 </body>
15 </html>\
16 """.encode()
17
18 response = """\
19 HTTP/1.1 200 OK
20 Date: Mon, 24 Oct 2022 20:04:23 GMT
21 Content-Type: text/html
22 Content-Length: {}
23 Connection: keep-alive
24 Server: wayne.magedu.com
25
26 """.format(len(html)).replace('\n', '\r\n').encode() + html
27
28 def accept(server):
29     i = 1
30     while True:
31         conn, raddr = server.accept()
32         threading.Thread(target=recv, name="recv-{}".format(i), args=(conn,
33 raddr)).start()
34         i += 1
35
36 def recv(conn: socket.socket, raddr):
37     try:
38         data = conn.recv(4096)
39         if not data:
40             print(raddr, 'bye~~~~')
41             return
42         # print(data)
43         conn.send(response)
44     except Exception as e:
45         print(e, '~~~~~')
46
47
48 if __name__ == '__main__':
49     server = socket.socket()
50     laddr = ('0.0.0.0', 9999)
51     server.bind(laddr)
52     server.listen(1024)
```

```

53
54     threading.Thread(target=accept, name="accept", args=(server,),
55 daemon=True).start()
56
57     while True:
58         time.sleep(60)
59         print(threading.active_count())

```

阻塞的IO导致该线程进入阻塞态，就该让出CPU，这对性能影响不大。此多线程程序最大的问题在于，当高并发到来，连接非常多，多线程的频繁地创建和销毁，以及管理线程的成本太高了。

接下来，我们用线程池来简单优化一下，看看能否提升性能？IO多路复用又是什么东西，它能提高多少性能？

实战：实现WEB服务器——线程池版

上例实现了多线程加阻塞IO版本

- 一个客户端请求到达后端，开启一个线程为之服务
- 线程内运行函数代码，接收HTTP请求并解析，返回HTTP响应报文

问题

- 大量的线程为HTTP连接服务，用完就断，而创建和销毁线程的代价太高
 - 解决的方案就是利用线程池
- 如果拥有海量线程来处理并发客户端请求，线程调度时上下文切换将给系统造成巨大的性能消耗
 - 程序层面解决不了
 - 操作系统解决：非阻塞IO、IO多路复用

下面用Python高级异步线程池ThreadPoolExecutor来改造代码。

```

1  import threading
2  import time
3  import socket
4
5  html = """\
6  <!DOCTYPE html>
7  <html lang="en">
8  <head>
9      <meta charset="UTF-8">
10     <title>magedu</title>
11 </head>
12 <body>
13     <h1>马哥教育www.magedu.com -- Multithread Pool</h1>
14 </body>
15 </html>\
16 """.encode()
17
18 response = """\
19 HTTP/1.1 200 OK
20 Date: Mon, 24 Oct 2022 20:04:23 GMT
21 Content-Type: text/html
22 Content-Length: {}

```

```

23 Connection: keep-alive
24 Server: wayne.magedu.com
25
26 """.format(len(html)).replace('\n', '\r\n').encode() + html
27
28 from concurrent.futures import ThreadPoolExecutor
29
30 count = 10
31 executor = ThreadPoolExecutor(count)
32 # executor = ThreadPoolExecutor(max_workers=count)
33
34 def accept(server):
35     # i = 1
36     while True:
37         conn, raddr = server.accept()
38         # threading.Thread(target=recv, name="recv-{}".format(i), args=
(conn, raddr)).start()
39         # i += 1
40         executor.submit(recv, conn, raddr)
41
42 def recv(conn: socket.socket, raddr):
43     try:
44         data = conn.recv(4096)
45         if not data:
46             print(raddr, 'bye~~~~')
47             return
48         # print(data)
49         conn.send(response)
50     except Exception as e:
51         print(e, '~~~~~')
52
53 if __name__ == '__main__':
54     server = socket.socket()
55     laddr = ('0.0.0.0', 9999)
56     server.bind(laddr)
57     server.listen(1024)
58
59     # threading.Thread(target=accept, name="accept", args=(server,),
daemon=True).start()
60     executor.submit(accept, server)
61
62     while True:
63         time.sleep(60)
64         print(threading.active_count())

```