Yuan先生

随笔 - 1 文章 - 141 评论 - 143

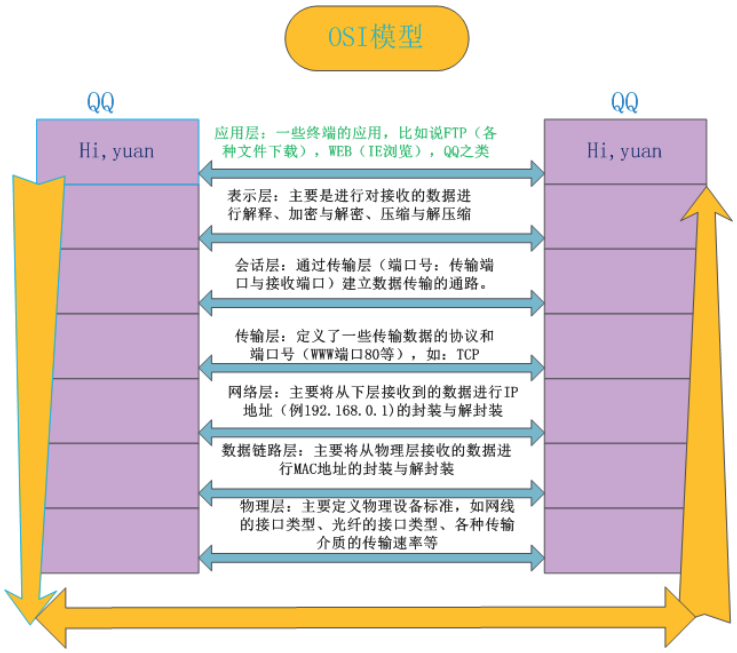
Py西游攻关之Socket网络编程

知识预览

-

计算机网络

回到顶部



网络通信要素：

A:IP地址 (1) 用来标识网络上一台独立的主机

(2) IP地址 = 网络地址 + 主机地址（网络号：用于识别主机所在的网络/网段。主机号：用于识别该网络中的主机）

(3) **特殊的IP地址：127.0.0.1（本地回环地址、保留地址，点分十进制）**可用于简单的测试网卡是否故障。表示本机。

B:端口号：(1) 用于标识进程的逻辑地址。不同的进程都有不同的端口标识。

(2) **端口：**要将数据发送到对方指定的应用程序上，为了标识这些应用程序，所以给这些网络应用程序都用数字进行标识。为了方便称呼这些数字，则将这些数字称为端口。（此端口是一个逻辑端口）

C: 传输协议：通讯的规则。例如：TCP、UDP协议（好比两个人得用同一种语言进行交流）

①、UDP：User Datagram Protocol用户数据报协议

特点：

- 面向无连接：传输数据之前源端和目的端不需要建立连接。

- 每个数据报的大小都限制在64K（8个字节）以内。
- 面向报文的不可靠协议。（即：发送出去的数据不一定会接收得到）
- 传输速率快，效率高。
- 现实生活实例：邮局寄件、实时在线聊天、视频会议...等。

②、TCP：Transmission Control Protocol传输控制协议

特点：

- 面向连接：传输数据之前需要建立连接。
- 在连接过程中进行大量数据传输。
- 通过"三次握手"的方式完成连接，是安全可靠协议。
- 传输速度慢，效率低。

注意：在TCP/IP协议中，TCP协议通过三次握手建立一个可靠的连接

```
# "我能给你讲个关于tcp的笑话吗?"
#   "行,给我讲个tcp笑话."
#   "好吧那我就给你讲个tcp笑话."
```

网络通讯步骤：

确定对端IP地址→ 确定应用程序端口 → 确定通讯协议

总结：网络通讯的过程其实就是一个（源端）不断封装数据包和（目的端）不断拆数据包的过程。

简单来说就是：发送方利用应用软件将上层应用程序产生的数据前后加上相应的层标识不断的往下层传输（封包过程），最终到达物理层通过看得见摸得着的物理层设备，例如：网线、光纤...等将数据包传输到数据接收方，然后接收方则通过完全相反的操作不断的读取和去除每一层的标识信息（拆包过程），最终将数据传递到最高层的指定的应用程序端口，并进行处理。

SOCKET 编程

要想理解socket,就要先来理解TCP,UDP协议

TCP/IP（Transmission Control Protocol/Internet Protocol）即传输控制协议/网间协议，定义了主机如何连入因特网及数据如何再它们之间传输的标准，

从字面意思来看TCP/IP是TCP和IP协议的合称，但实际上TCP/IP协议是指因特网整个TCP/IP协议族。不同于ISO模型的七个分层，TCP/IP协议参考模型把所有的TCP/IP系列协议归类到四个抽象层中

应用层：TFTP，HTTP，SNMP，FTP，SMTP，DNS，Telnet 等等

传输层：TCP，UDP

网络层：IP，ICMP，OSPF，EIGRP，IGMP

数据链路层：SLIP，CSLIP，PPP，MTU

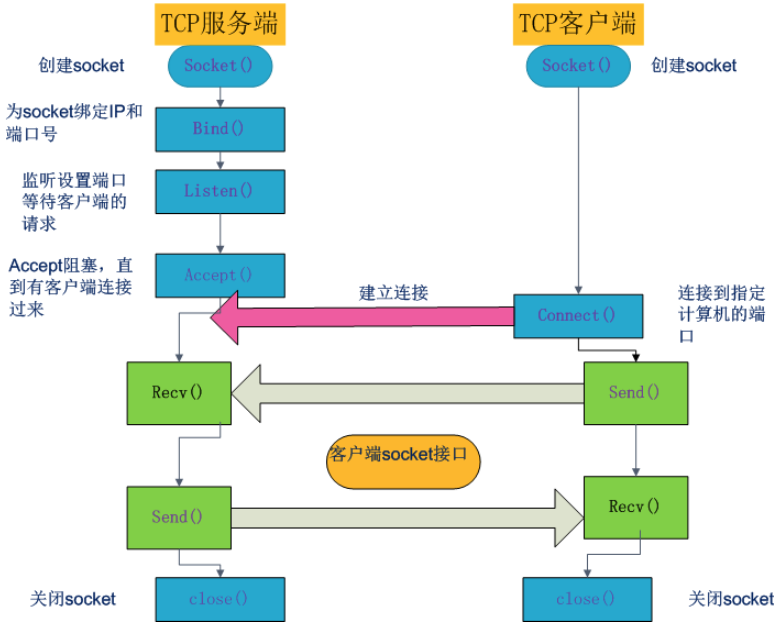
每一抽象层建立在低一层提供的服务上，并且为高一层提供服务，看起来大概是这样子的

我们可以利用ip地址＋协议＋端口号唯一标示网络中的一个进程。能够唯一标示网络中的进程后，它们就可以利用socket进行通信了，我们经常把socket翻译为套接字，socket是在应用层和传输层(TCP/IP协议族通信)之间的一个抽象层，是一组接口，它把TCP/IP层复杂的操作抽象为几个简单的接口供应用层调用已实现进程在网络中通信。

应用程序两端通过“套接字”向网络发出请求或者应答网络请求。可以把socket理解为通信的把手（hand）

socket起源于UNIX，在Unix一切皆文件哲学的思想下，socket是一种"打开—读/写—关闭"模式的实现，服务器和客户端各自维护一个"文件"，在建立连接打开后，可以向自己文件写入内容供对方读取或者读取对方内容，通讯结束时关闭文件。socket的英文原义是“插槽”或“插座”，就像我们家里座机一样，如果没有网线的那个插口，电话是无法通信的。Socket是实现TCP，UDP协议的接口，便于使用TCP,UDP。

socket通信流程




```
# 流程描述:
#
# 1 服务器根据地址类型（ipv4,ipv6）、socket类型、协议创建socket
#
# 2 服务器为socket绑定ip地址和端口号
#
# 3 服务器socket监听端口号请求，随时准备接收客户端发来的连接，这时候服务器的socket并没有被打开
#
# 4 客户端创建socket
#
# 5 客户端打开socket，根据服务器ip地址和端口号试图连接服务器socket
#
# 6 服务器socket接收到客户端socket请求，被动打开，开始接收客户端请求，直到客户端返回连接信息。这时候socket进入阻
#
# 7 客户端连接成功，向服务器发送连接状态信息
#
# 8 服务器accept方法返回，连接成功
#
# 9 客户端向socket写入信息(或服务端向socket写入信息)
#
# 10 服务器读取信息(客户端读取信息)
#
# 11 客户端关闭
#
# 12 服务器端关闭
```

相关方法及参数介绍

```
sk.bind(address)

    #s.bind(address) 将套接字绑定到地址。address地址的格式取决于地址族。在AF_INET下，以元组（host,port）的形

sk.listen(backlog)

    #开始监听传入连接。backlog指定在拒绝连接之前，可以挂起的最大连接数量。

        #backlog等于5，表示内核已经接到了连接请求，但服务器还没有调用accept进行处理的连接个数最大为5
        #这个值不能无限大，因为要在内核中维护连接队列

sk.setblocking(bool)

    #是否阻塞（默认True），如果设置False，那么accept和recv时一旦无数据，则报错。

sk.accept()
```

```
    #接受连接并返回（conn,address），其中conn是新的套接字对象，可以用来接收和发送数据。address是连接客户端的地址
    #接收TCP 客户的连接（阻塞式）等待连接的到来
sk.connect(address)
    #连接到address处的套接字。一般，address的格式为元组（hostname,port），如果连接出错，返回socket.error错误。
sk.connect_ex(address)
    #同上，只不过会有返回值，连接成功时返回 0 ，连接失败时候返回编码，例如：10061
sk.close()
    #关闭套接字
sk.recv(bufsize[,flag])
    #接受套接字的数据。数据以字符串形式返回，bufsize指定最多可以接收的数量。flag提供有关消息的其他信息，通常可以忽
sk.recvfrom(bufsize[.flag])
    #与recv()类似，但返回值是（data,address）。其中data是包含接收数据的字符串，address是发送数据的套接字地址。
sk.send(string[,flag])
    #将string中的数据发送到连接的套接字。返回值是要发送的字节数量，该数量可能小于string的字节大小。即：可能未将指
sk.sendall(string[,flag])
    #将string中的数据发送到连接的套接字，但在返回之前会尝试发送所有数据。成功返回None，失败则抛出异常。
       #内部通过递归调用send，将所有内容发送出去。
sk.sendto(string[,flag],address)
    #将数据发送到套接字，address是形式为（ipaddr，port）的元组，指定远程地址。返回值是发送的字节数。该函数主要用
sk.settimeout(timeout)
    #设置套接字操作的超时期，timeout是一个浮点数，单位是秒。值为None表示没有超时期。一般，超时期应该在刚创建套接字
sk.getpeername()
    #返回连接套接字的远程地址。返回值通常是元组（ipaddr,port）。
sk.getsockname()
    #返回套接字自己的地址。通常是一个元组(ipaddr,port)
sk.fileno()
    #套接字的文件描述符
```

四 实例(屌丝追女神的故事)

故事1

背景：从前，有个屌丝阿武，想追女神阿思，闷骚型，心想：老子就给她一次机会，不把握就算了...

```
#################server
import socket
ip_port = ('127.0.0.1',9997)
sk = socket.socket()
sk.bind(ip_port)
sk.listen(5)

print ('server waiting...')

conn,addr = sk.accept()
client_data = conn.recv(1024)
print (str(client_data,"utf8"))
conn.sendall(bytes('滚蛋!',encoding="utf-8"))

sk.close()

################client
import socket
ip_port = ('127.0.0.1',9997)

sk = socket.socket()
sk.connect(ip_port)


sk.sendall(bytes('俺喜欢你',encoding="utf8"))

server_reply = sk.recv(1024)
print (str(server_reply,"utf8"))
```

故事2

背景：又有个屌丝，名叫武二，想追女神阿思，内心强大，臭不要脸，于是...

```
#-----------------------------------------------server.py
#-------------------------------------------------
import socket
ip_port = ('127.0.0.1',8888)
sk = socket.socket()
sk.bind(ip_port)
sk.listen(2)
print ("服务端启动...")
conn,address = sk.accept()
while True:
    client_data=conn.recv(1024)
    if str(client_data,"utf8")=='exit':
        break
    print (str(client_data,"utf8"))
```

```
    server_response=input(">>>")
    conn.sendall(bytes(server_response,"utf8"))

conn.close()

#-----------------------------------------------client.py
#-----------------------------------------------

import socket
ip_port = ('127.0.0.1',8888)
sk = socket.socket()
sk.connect(ip_port)
print ("客户端启动:")

while True:
    inp = input('>>>')
    sk.sendall(bytes(inp,"utf8"))
    if inp == 'exit':
        break
    server_response=sk.recv(1024)
    print (str(server_response,"utf8"))
sk.close()
```

注意：固然武二脸皮够厚，但是落花有意流水无情，女神不给机会和你聊也白搭（server端也需要有while）。

故事3

背景：某天，女神孤独难耐，决定调整状态想和多人聊一聊找个合适的(虽然不是并发，但是可以在不重新开启server的前提下与多人聊天)，于是52抓住了机会，俩人一顿深聊后.....52放弃了。

代码同上

win:

```
import socket

ip_port = ('127.0.0.1',8870)
sk = socket.socket()
sk.bind(ip_port)
sk.listen(2)
print ("服务端启动...")

while True:
    conn,address = sk.accept()
    print(address)
    while True:
        try:
            client_data=conn.recv(1024)
        except:
            print("意外中断")
            break
        print (str(client_data,"utf8"))

        server_response=input(">>>")
        conn.sendall(bytes(server_response,"utf8"))

    conn.close()

##################
import socket
ip_port = ('127.0.0.1',8870)
sk = socket.socket()
sk.connect(ip_port)
print ("客户端启动:")

while True:
    inp = input('>>>')
    if inp == 'exit':
        break
    sk.sendall(bytes(inp,"utf8"))
    server_response=sk.recv(1024)
    print (str(server_response,"utf8"))
sk.close()
```

linux:

```
client_data=conn.recv(1024)
print (str(client_data,"utf8"))
if len(client_data)==0:
      print("意外中断")
      break

server_response=input(">>>")
conn.sendall(bytes(server_response,"utf8"))
```

关于linux这里，有同学可能会问

```
client_data=conn.recv(1024)
print('3333',str(client_data,"utf8"))
if len(client_data)==0:
     print("意外中断")
     break
```

这样如果客户端发的数据就是空数据的话岂不是也意外退出，那不就bug啦?

其实不用担心，如果客户端send了一个空数据后客户端继续向下执行，而server端的recv方法会继续阻塞，直到接收到一个非空数据才会继续向下执行。

而stop_button的退出会使client_data成为一个空数据继续向下执行，所以可以依据代码意外退出。

ok,关键的问题来了：如何让女神可以同时和多个屌丝聊天呢？(如何实现并发)

简单并发实例

```
#-----------------------------------------------------server.py
#-----------------------------------------------------
import socketserver

class MyServer(socketserver.BaseRequestHandler):

    def handle(self):
        print ("服务端启动...")
        while True:
            conn = self.request
            print (self.client_address)
            while True:
                client_data=conn.recv(1024)
                print (str(client_data,"utf8"))
                print ("waiting...")
                conn.sendall(client_data)
            conn.close()

if __name__ == '__main__':
    server = socketserver.ThreadingTCPServer(('127.0.0.1',8091),MyServer)
    server.serve_forever()

#-----------------------------------------------------client.py
#-----------------------------------------------------
import socket

ip_port = ('127.0.0.1',8091)
sk = socket.socket()
sk.connect(ip_port)
print ("客户端启动: ")
while True:
    inp = input('>>>')
    sk.sendall(bytes(inp,"utf8"))
    if inp == 'exit':
        break
    server_response=sk.recv(1024)
    print (str(server_response,"utf8"))
sk.close()
```

聊天并发实例

```
import socketserver

class MyServer(socketserver.BaseRequestHandler):

    def handle(self):
        print ("服务端启动...")
        while True:
            conn = self.request
            print (self.client_address)
            while True:

                client_data=conn.recv(1024)

                print (str(client_data,"utf8"))
                print ("waiting...")
                server_response=input(">>>")
                conn.sendall(bytes(server_response,"utf8"))
                # conn.sendall(client_data)

            conn.close()
            # print self.request,self.client_address,self.server


if __name__ == '__main__':
    server = socketserver.ThreadingTCPServer(('127.0.0.1',8098),MyServer)
    server.serve_forever()


##########################################
import socket


ip_port = ('127.0.0.1',8098)
sk = socket.socket()
sk.connect(ip_port)
print ("客户端启动: ")
while True:
    inp = input('>>>')
    sk.sendall(bytes(inp,"utf8"))
    server_response=sk.recv(1024)
    print (str(server_response,"utf8"))
    if inp == 'exit':
        break
sk.close()
```

五 其它应用

命令传送1:

```
#-------------------------------------------------server
#-------------------------------------------------
import socket
import subprocess
ip_port = ('127.0.0.1',8879)
sk = socket.socket()
sk.bind(ip_port)
sk.listen(5)
print ("服务端启动...")
while True:
    conn,address = sk.accept()
    while True:
        try:

            client_data=conn.recv(1024)
        except Exception:
            break
        print (str(client_data,"utf8"))
        print ("waiting...")
```

```
        # server_response=input(">>>")
        # conn.sendall(bytes(server_response,"utf8"))
        cmd=str(client_data,"utf8").strip()
        cmd_call=subprocess.Popen(cmd,shell=True,stdout=subprocess.PIPE)
        cmd_result=cmd_call.stdout.read()
        if len(cmd_result)==0:
            cmd_result=b"no output!"
        conn.sendall(cmd_result)
        print('send data size',len(cmd_result))
        print('******************')
        print('******************')
        print('******************')

    conn.close()

#-----------------------------------------------client
#-----------------------------------------------
import socket
ip_port = ('127.0.0.1',8879)
sk = socket.socket()
sk.connect(ip_port)
print ("客户端启动: ")
while True:
    inp = input('cdm:>>>').strip( )
    if len(inp)==0:
        continue
    if inp=="q":
        break
    sk.sendall(bytes(inp,"utf8"))
    server_response=sk.recv(1024)
    print (str(server_response,"gbk"))
    print('receive data size',len(server_response))
    if inp == 'exit':
        break
sk.close()
```

```
#试一试
netstat -an
```

conclusion:

sendall会把数据直接全部发送到客户端，客户端将所有的数据都放到缓冲区,每次recv多少字节取决于recv内的参数，理论不应该超过8k。

所以，并不能一次recv()无限大数据，所以这里我们应该通过循环去接收。

```
sk.recv(4096)
```

命令传送2：解决大数据传送和粘包问题

```
import socketserver
import subprocess

class Myserver(socketserver.BaseRequestHandler):
    def handle(self):
        while True:
            conn=self.request
            conn.sendall(bytes("欢迎登录","utf8"))
            while True:
                client_bytes=conn.recv(1024)
                if not client_bytes:break
                client_str=str(client_bytes,"utf8")
                print(client_str)
                command=client_str

                result_str=subprocess.getoutput(command)
                result_bytes = bytes(result_str,encoding='utf8')
                info_str="info|%d"%len(result_bytes)
                conn.sendall(bytes(info_str,"utf8"))
                # conn.recv(1024)
```

```
                conn.sendall(result_bytes)
            conn.close()

if __name__=="__main__":
    server=socketserver.ThreadingTCPServer(("127.0.0.1",9998),Myserver)
    server.serve_forever()

####################################client


import socket
ip_port=("127.0.0.1",9998)

sk=socket.socket()
sk.connect(ip_port)
print("客户端启动...")

print(str(sk.recv(1024),"utf8"))

while True:
    inp=input("please input:").strip()


    sk.sendall(bytes(inp,"utf8"))
    basic_info_bytes=sk.recv(1024)
    print(str(basic_info_bytes,"utf8"))
    # sk.send(bytes('ok','utf8'))
    result_length=int(str(basic_info_bytes,"utf8").split("|")[1])

    print(result_length)
    has_received=0
    content_bytes=bytes()
    while has_received<result_length:
        fetch_bytes=sk.recv(1024)
        has_received+=len(fetch_bytes)
        content_bytes+=fetch_bytes
    cmd_result=str(content_bytes,"utf8")
    print(cmd_result)

sk.close()
```

文件上传

```
import socket,os
ip_port=("127.0.0.1",8898)
sk=socket.socket()
sk.bind(ip_port)
sk.listen(5)
BASE_DIR=os.path.dirname(os.path.abspath(__file__))

while True:
    print("waiting connect")
    conn,addr=sk.accept()
    flag = True
    while flag:

            client_bytes=conn.recv(1024)
            client_str=str(client_bytes,"utf8")
            func,file_byte_size,filename=client_str.split("|",2)

            path=os.path.join(BASE_DIR,'yuan',filename)
            has_received=0
            file_byte_size=int(file_byte_size)

            f=open(path,"wb")
            while has_received<file_byte_size:
                data=conn.recv(1024)
                f.write(data)
                has_received+=len(data)
            print("ending")
            f.close()
```

```
#-------------------------------------------client
#-------------------------------------------
import socket
import re,os,sys
ip_port=("127.0.0.1",8898)
sk=socket.socket()
sk.connect(ip_port)
BASE_DIR=os.path.dirname(os.path.abspath(__file__))
print("客户端启动....")

while True:
    inp=input("please input:")

    if inp.startswith("post"):
        method,local_path=inp.split("|",1)
        local_path=os.path.join(BASE_DIR,local_path)
        file_byte_size=os.stat(local_path).st_size
        file_name=os.path.basename(local_path)
        post_info="post|%s|%s"%(file_byte_size,file_name)
        sk.sendall(bytes(post_info,"utf8"))
        has_sent=0
        file_obj=open(local_path,"rb")
        while has_sent<file_byte_size:
            data=file_obj.read(1024)
            sk.sendall(data)
            has_sent+=len(data)
        file_obj.close()
        print("上传成功")
```

注意：

1 纸条就是conn

2 一收一发

3 client_data=conn.recv(1024)

if 那边send一个空数据 这边recv为空，则recv继续阻塞，等待其他的数据。所以聊天的时候好好聊，别发空数据。

socketserver

虽说用Python编写简单的网络程序很方便，但复杂一点的网络程序还是用现成的框架比较好。这样就可以专心事务逻辑，而不是套接字的各种细节。SocketServer模块简化了编写网络服务程序的任务。同时SocketServer模块也是Python标准库中很多服务器框架的基础。

socketserver模块可以简化网络服务器的编写，Python把网络服务抽象成两个主要的类，一个是Server类，用于处理连接相关的网络操作，另外一个则是RequestHandler类，用于处理数据相关的操作。并且提供两个MixIn类，用于扩展 Server，实现多进程或多线程。

Server类

它包含了种五种server类，BaseServer(不直接对外服务)。TCPServer使用TCP协议，UDPServer使用UDP协议，还有两个不常使用的，即UnixStreamServer和UnixDatagramServer,这两个类仅仅在unix环境下有用(AF_unix)。

Base class for server classes.

```
class BaseServer
```

This uses the Internet TCP protocol, which provides for continuous streams of data between the client and server.

```
class socketserver.TCPServer(server_address, RequestHandlerClass, bind_and_activate=True)
```

This uses datagrams, which are discrete packets of information that may arrive out of order or be lost while in transit. The parameters are the same as for TCPServer

```
class socketserver.UDPServer(server_address, RequestHandlerClass, bind_and_activate=True)
```

These more infrequently used classes are similar to the TCP and UDP classes, but use Unix domain sockets; they’ re not available on non-Unix platforms. The parameters are the same as for TCPServer.

```
class socketserver.UnixStreamServer(server_address, RequestHandlerClass, bind_and_activate=True)
class socketserver.UnixDatagramServer(server_address, RequestHandlerClass,bind_and_activate=True)
```

```
class UnixStreamServer(TCPServer):
    address_family = socket.AF_UNIX

class UnixDatagramServer(UDPServer):
    address_family = socket.AF_UNIX
```

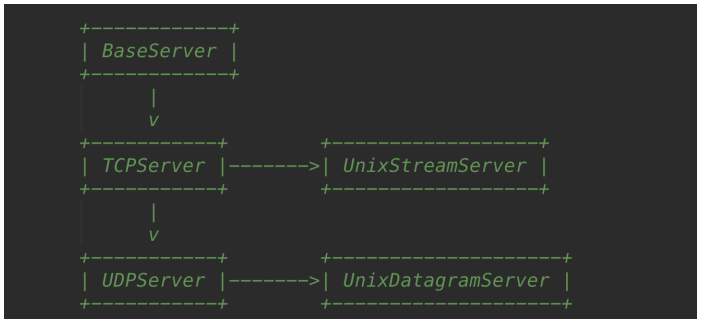
BaseServer的源码：

```
class BaseServer:

    """Base class for server classes.

    Methods for the caller:

    - __init__(server_address, RequestHandlerClass)
    - serve_forever(poll_interval=0.5)
    - shutdown()
    - handle_request()  # if you do not use serve_forever()
    - fileno() -> int   # for select()

    Methods that may be overridden:

    - server_bind()
    - server_activate()
    - get_request() -> request, client_address
    - handle_timeout()
    - verify_request(request, client_address)
    - server_close()
    - process_request(request, client_address)
    - shutdown_request(request)
    - close_request(request)
    - service_actions()
    - handle_error()

    Methods for derived classes:

    - finish_request(request, client_address)

    Class variables that may be overridden by derived classes or
    instances:

    - timeout
    - address_family
    - socket_type
    - allow_reuse_address

    Instance variables:

    - RequestHandlerClass
    - socket

    """

    timeout = None

    def __init__(self, server_address, RequestHandlerClass):
        """Constructor.  May be extended, do not override."""
        self.server_address = server_address
        self.RequestHandlerClass = RequestHandlerClass
        self.__is_shut_down = threading.Event()
        self.__shutdown_request = False

    def server_activate(self):
        """Called by constructor to activate the server.

        May be overridden.

        """
        pass

    def serve_forever(self, poll_interval=0.5):
        """Handle one request at a time until shutdown.
```

```
        Polls for shutdown every poll_interval seconds. Ignores
        self.timeout. If you need to do periodic tasks, do them in
        another thread.
        """
        self.__is_shut_down.clear()
        try:
            while not self.__shutdown_request:
                # XXX: Consider using another file descriptor or
                # connecting to the socket to wake this up instead of
                # polling. Polling reduces our responsiveness to a
                # shutdown request and wastes cpu at all other times.
                r, w, e = _eintr_retry(select.select, [self], [], [],
                                       poll_interval)
                if self in r:
                    self._handle_request_noblock()

                self.service_actions()
        finally:
            self.__shutdown_request = False
            self.__is_shut_down.set()

    def shutdown(self):
        """Stops the serve_forever loop.

        Blocks until the loop has finished. This must be called while
        serve_forever() is running in another thread, or it will
        deadlock.
        """
        self.__shutdown_request = True
        self.__is_shut_down.wait()

    def service_actions(self):
        """Called by the serve_forever() loop.

        May be overridden by a subclass / Mixin to implement any code that
        needs to be run during the loop.
        """
        pass

    # The distinction between handling, getting, processing and
    # finishing a request is fairly arbitrary.  Remember:
    #
    # - handle_request() is the top-level call.  It calls
    #   select, get_request(), verify_request() and process_request()
    # - get_request() is different for stream or datagram sockets
    # - process_request() is the place that may fork a new process
    #   or create a new thread to finish the request
    # - finish_request() instantiates the request handler class;
    #   this constructor will handle the request all by itself

    def handle_request(self):
        """Handle one request, possibly blocking.

        Respects self.timeout.
        """
        # Support people who used socket.settimeout() to escape
        # handle_request before self.timeout was available.
        timeout = self.socket.gettimeout()
        if timeout is None:
            timeout = self.timeout
        elif self.timeout is not None:
            timeout = min(timeout, self.timeout)
        fd_sets = _eintr_retry(select.select, [self], [], [], timeout)
        if not fd_sets[0]:
            self.handle_timeout()
            return
        self._handle_request_noblock()

    def _handle_request_noblock(self):
        """Handle one request, without blocking.

        I assume that select.select has returned that the socket is
        readable before this function was called, so there should be
        no risk of blocking in get_request().
        """
        try:
```

```
            request, client_address = self.get_request()
        except OSError:
            return
        if self.verify_request(request, client_address):
            try:
                self.process_request(request, client_address)
            except:
                self.handle_error(request, client_address)
                self.shutdown_request(request)

    def handle_timeout(self):
        """Called if no new request arrives within self.timeout.

        Overridden by ForkingMixIn.
        """
        pass

    def verify_request(self, request, client_address):
        """Verify the request.  May be overridden.

        Return True if we should proceed with this request.

        """
        return True

    def process_request(self, request, client_address):
        """Call finish_request.

        Overridden by ForkingMixIn and ThreadingMixIn.

        """
        self.finish_request(request, client_address)
        self.shutdown_request(request)

    def server_close(self):
        """Called to clean-up the server.

        May be overridden.

        """
        pass

    def finish_request(self, request, client_address):
        """Finish one request by instantiating RequestHandlerClass."""
        self.RequestHandlerClass(request, client_address, self)

    def shutdown_request(self, request):
        """Called to shutdown and close an individual request."""
        self.close_request(request)

    def close_request(self, request):
        """Called to clean up an individual request."""
        pass

    def handle_error(self, request, client_address):
        """Handle an error gracefully.  May be overridden.

        The default is to print a traceback and continue.

        """
        print('-'*40)
        print('Exception happened during processing of request from', end=' ')
        print(client_address)
        import traceback
        traceback.print_exc() # XXX But this goes to stderr!
        print('-'*40)
```

There are five classes in an inheritance diagram, four of which represent synchronous servers of four types:



RequestHandler类

所有requestHandler都继承BaseRequestHandler基类。

```
class BaseRequestHandler:

    """Base class for request handler classes.

    This class is instantiated for each request to be handled.  The
    constructor sets the instance variables request, client_address
    and server, and then calls the handle() method.  To implement a
    specific service, all you need to do is to derive a class which
    defines a handle() method.

    The handle() method can find the request as self.request, the
    client address as self.client_address, and the server (in case it
    needs access to per-server information) as self.server.  Since a
    separate instance is created for each request, the handle() method
    can define arbitrary other instance variariables.

    """

    def __init__(self, request, client_address, server):
        self.request = request
        self.client_address = client_address
        self.server = server
        self.setup()
        try:
            self.handle()
        finally:
            self.finish()

    def setup(self):
        pass

    def handle(self):
        pass

    def finish(self):
        pass
```

创建一个socketserver 至少分以下几步

1. First, you must create a request handler class by subclassing the BaseRequestHandlerclass and overriding its handle() method; this method will process incoming requests.
2. Second, you must instantiate one of the server classes, passing it the server's address and the request handler class.
3. Then call the handle_request() orserve_forever() method of the server object to process one or many requests.
4. Finally, call server_close() to close the socket.

```
import socketserver

class MyTCPHandler(socketserver.BaseRequestHandler):
```

```
    """
    The request handler class for our server.

    It is instantiated once per connection to the server, and must
    override the handle() method to implement communication to the
    client.
    """

    def handle(self):
        # self.request is the TCP socket connected to the client
        self.data = self.request.recv(1024).strip()
        print("{} wrote:".format(self.client_address[0]))
        print(self.data)
        # just send back the same data, but upper-cased
        self.request.sendall(self.data.upper())

if __name__ == "__main__":
    HOST, PORT = "localhost", 9999

    # Create the server, binding to localhost on port 9999
    server = socketserver.TCPServer((HOST, PORT), MyTCPHandler)

    # Activate the server; this will keep running until you
    # interrupt the program with Ctrl-C
    server.serve_forever()
```

让你的socketserver并发起来， 必须选择使用以下一个多并发的类

```
class socketserver.ForkingTCPServer

class socketserver.ForkingUDPServer

class socketserver.ThreadingTCPServer

class socketserver.ThreadingUDPServer
```

所以：

```
server = socketserver.TCPServer((HOST, PORT), MyTCPHandler)
#替换为
server = socketserver.ThreadingTCPServer((HOST, PORT), MyTCPHandler)
```

思考：

1 tcp与udp的区别（三次握手后，建立连接，双向通道，一个收，一个发，tcp每次接到数据后都会有一个应答，有了应答，新的数据就会被覆盖掉）

2 粘包

Yuan先生
关注 - 1
粉丝 - 3941
我在关注他 取消关注

13 0

posted @ 2016-07-21 18:10 Yuan先生 阅读(8012) 评论(7) 编辑 收藏

Post Comment

#1楼 2017-07-24 10:05 | LiChaoAI 回复 引用

打卡!

支持(0) 反对(0)

#2楼 2017-08-16 09:12 | guojigang 回复 引用