

# Python 实现 MQTT/HTTP/FTP 服务器和客户端

## 作者:向仔州

MQTT 和我们平时的 tcp 服务端客户端收发逻辑有点区别.....	3
Windows 系统 pip 软件安装, 和 paho-mqtt 库文件安装.....	3
Linux 搭建 MQTT 服务器, 使用 mosquitto 来做 MQTT 服务器.....	4
<b>mosquito 库使用.....</b>	<b>5</b>
python 在 windows 下实现 MQTT 客户端.....	7
mqtt.Client(...)//该函数是给程序分配 1 个 MQTT 客户端对象, 使用这个对象操作 MQTT.....	7
on_connect(...)//on_connect 是自己定义的回调函数, 连接 MQTT 服务器成功的时候会调用这个回调函数.....	7
on_message(...)//收到服务器发来的数据.....	7
publish(...)# 向主题为...发布, 内容为 ‘...’ 的数据, 也就是发数据给服务器了, 类似 socket 的 send.....	9
在 linux 上实现有 MQTT 服务器的客户端程序, 也就是服务器上的客户端来处理其它连接上的客户端.....	9
subscribe(...)//订阅主题.....	10
loop_start(..)//在 connect 连接之后调用一次 loop_start 会在后台运行一个线程来处理 loop.....	11
loop_stop() #来停止后台线程.....	11
多个开发板客户端发数据到服务器上多客户端的案例.....	12
loop_start() 使用问题.....	12
disconnect() 使用.....	15
QOS 重点讲解.....	17
保留消息, 就是订阅主题就会马上收到上一次其它设备发布的指定主题消息.....	19
MQTT 心跳机制.....	20
HTTP 协议 Python 实现.....	21
JSON 概念与使用.....	24
Json.dumps(变量) //将 python 数据类型转成字符串.....	25
Json.loads(字符串变量) #将 json 字符串转换成 python 使用的字典返回字典 ...	25
Python 从头实现 http 服务器.....	26
HTTP 静态服务器开发.....	27
使用 http 库来实现 http 服务器.....	31
对象 = HTTPServer((服务器 IP, 端口号), 被客户端请求后要执行的回调类) #创建 HTTP 服务器.....	31
对象. serve_forever() #监听端口.....	31
在解析 JSON 格式很复杂的 HTTP 文件时遇到一些格式情况.....	33
python 实现 HTTP 文件服务器.....	35
<b>miniserver 使用方式.....</b>	<b>36</b>
用 HTTP 客户端测试软件 Postman 测试下文件服务器下载功能。.....	37
使用 python 的 http 协议来下载 http 文件服务器里面的文件内容.....	38
如果是下载大文件, 如何实现 http 请求, 数据分段获取?.....	38
HEAD 请求使用, HEAD 请求服务器只返回响应头, 不会返回数据本身。.....	39
FTP 文件传输实现.....	39
<b>TCP 黏包问题解决(重点).....</b>	<b>39</b>
TCP 大文件下载实验.....	43
FTP 服务端/客户端实现逻辑.....	45
os.path.dirname(path) #去掉文件名, 返回目录名.....	45

os.path.abspath(__file__)	#__file__是获取当前执行 file 命令的程序文件 绝对路径.....	45
sys.path	#输出当前系统的环境变量.....	46
sys.path.append(path)	#加入新的路径到系统环境变量中.....	46
服务端程序初步框架.....	46	
os.path.join()	#函数用于路径拼接字符，形成新的路径.....	47
用户认证实现.....	50	
对象 = configparser.ConfigParser()	#创建解析. ini 文件的对象.....	50
configparser.read(读取文件)	.....	50
返回列表 = configparser.section()	#获取 ini 文件所有段.....	50
configparser.add_section(填入加入段的字符)	#在 ini 文件中加入.....	51
configparser.write(...)	#加入段或者键值之后，一定要执行 write，才能成功写入.....	51
configparser.set(...)	#在指定段加入新的键值对.....	51
<b>FTP 固定报文头实现.</b>	54	
zfill 函数用法.....	54	
<b>Linux 中 FTP 服务器搭建(使用 vsftpd 软件)</b>	56	
<b>ubuntu 查看系统指定端口是否被开放.</b>	57	
500 OOPS: vsftpd: both local and anonymous access disabled!	本地和匿名访问都被禁用.....	58
500 OOPS: cannot change directory:/FTP_dir.....	58	
<b>登陆 FTP 服务器.</b>	58	
Wireshark 抓包 VMware 虚拟机与本地 PC 主机的通信数据.....	59	
标准 FTP 实现.....	63	
FTP PWD 命令.....	65	
<b>FTP 目录列表/文件列表查询.</b>	66	
FTP SIZE 命令.....	68	
FTP RETR 命令.....	68	
<b>FTP 文件上传.</b>	70	
FTP STOR 命令.....	70	
<b>FTP 实现大文件下载.</b>	72	
FTP QUIT 命令.....	72	
<b>FTP 大文件下载，断点续传问题.</b>	75	
FTP REST 命令.....	75	

## MQTT 和我们平时的 tcp 服务端客户端收发逻辑有点区别

我们常规的TCP/UDP客户端服务器收发模式是下面这样



但是MQTT就不是常规的客户端服务器这种做法



所以现在MQTT服务器不需要你来写服务器了，你只需要下载软件在linux上面运行就行，至于稳不稳定就看你下载的什么软件那么MQTT服务器接收的数据如何处理呢？

所以你要在MQTT服务器的主机上写一个MQTT客户端，这个客户端就是来当做服务器程序处理数据的

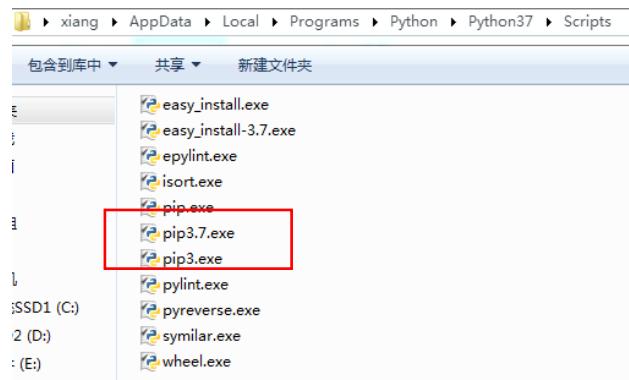
## Windows 系统 pip 软件安装，和 paho-mqtt 库文件安装

下载一个 get-pip.py 文件，用这个 python 文件帮你下载 pip  
执行 `python get-pip.py`

```
C:\Users\xiang\AppData\Local\Programs\Python\Python37\pip>python get-pip.py
Collecting pip
  Downloading https://files.pythonhosted.org/packages/54/0c/d01aa759fdc501a58f431eb594a17495f15b88da142ce14b5845662c13f3/pip-20.0.2-py2.py3-none-any.whl (1.4MB)
    39% [██████████] 563kB 18kB/s eta 0:00:49
    39% [██████████] 573kB 16kB/s eta 0:00:52
    40% [██████████] 583kB 22kB/s eta 0:00:39
    41% [██████████] 593kB 22kB/s eta 0:00:39
Successfully uninstalled pip-19.2.3
The script wheel.exe is installed in 'C:\Users\xiang\AppData\Local\Programs\Python\Python37\Scripts' which is not on PATH.
Consider adding this directory to PATH or, if you prefer to suppress this warning, use --no-warn-script-location.
Successfully installed pip-20.0.2 wheel-0.34.2
```

根据黄色字符提示 pip 已经安装在了

C:/Users/xiang/AppData/Local/Programs/Python/Python37/Scripts 路径下



这就是 pip 软件

为了方便使用我们把 pip 放入环境变量，如果环境变量不成功可以直接去 scripts 目录下执行 pip

在 scripts 目录下用 pip 下载 paho-mqtt 库

pip install paho-mqtt

```
C:\Users\xiang\AppData\Local\Programs\Python\Python37\Scripts>pip install paho-mqtt
Collecting paho-mqtt
  Downloading paho-mqtt-1.5.0.tar.gz (99 kB)
    |████████████████████████████████████████████████████████████████████████████████| 99 kB 39 kB/s eta 0:00:
    |████████████████████████████████████████████████████████████████████████████████| 61 kB 46 kB/s eta 0:
    |████████████████████████████████████████████████████████████████████████████████| 71 kB 40 kB/s eta 0:
    |████████████████████████████████████████████████████████████████████████████████| 81 kB 34 kB/s eta 0:
    |████████████████████████████████████████████████████████████████████████████████| 92 kB 38 kB/s eta 0:
    |████████████████████████████████████████████████████████████████████████████████| 99 kB 40 kB/s eta 0:
kB/s
Building wheels for collected packages: paho-mqtt
  Building wheel for paho-mqtt (setup.py) ... done
    Created wheel for paho-mqtt: filename=paho_mqtt-1.5.0-py3-none-any.whl size=64
```

现在可以在 vscode 使用 python 版本的 mqtt 库了

## Linux 搭建 MQTT 服务器，使用 mosquitto 来做 MQTT 服务器

mosquito 库是用来做 MQTT 的中间服务器的，需要在 linux 下安装

```
root@ubuntu:/home/xiang/BC28serverTest/mosquitto# ls
mosquitto-1.6.8  mosquitto-1.6.8.tar.gz
```

解压 mosquito-1.6.8

修改配置文件关闭与 OPENSSL 相关的宏，这样虽然没有 SSL 加密，但是可以编译通过使用

```
root@ubuntu:/home/xiang/BC28serverTest/mosquitto/mosquitto# ls
about.html      CMakeLists.txt  CONTRIBUTING.md  instal
aclfile.example  compiling.txt   edl-v10          lib
ChangeLog.txt    config.h       epl-v10          libmos
client           config.mk     examples         libmos
root@ubuntu:/home/xiang/BC28serverTest/mosquitto/mosquitto# vim config.mk
```

```
# Comment out to disable SSL/TLS support
# Disabling this will also mean that
# it is strongly recommended that you do
# password authentication at all.
#WITH_TLS:=yes

# Comment out to disable TLS/PSK support
# WITH_TLS=yes.
# This must be disabled if using openSSL
#WITH_TLS_PSK:=yes
```

如果打开 SSL 功能可能就会出现编译报错，所以下面这一段跟 SSL 有关的可以暂时不做

```
/lib/libmosquitto.so.1: undefined reference to `OPENSSL_sk_num'  
/lib/libmosquitto.so.1: undefined reference to `SSL_CTX_up_ref'  
/lib/libmosquitto.so.1: undefined reference to `OPENSSL_sk_pop_free'  
/lib/libmosquitto.so.1: undefined reference to  
`SSL_CTX_set_alpn_protos'  
/lib/libmosquitto.so.1: undefined reference to `OPENSSL_sk_value'
```

下面这节安装 openssl 只是参考，可以不做，因为 mosquitto 编译链接 openssl 问题没有解决

```
root@ubuntu:/home/xiang/BC28serverTest/ssl/openssl-1.0.1q# ls  
ACKNOWLEDGMENTS CHANGES.SSLay doc INSTALL INSTALL.VM  
apps config engines install.com INSTALL.W3  
appveyor.yml Configure e_os2.h INSTALL.DIGPP INSTALL.W6  
.config shared --prefix=/usr/local/ 配置 openssl  
make install 编译安装 openssl  
cp libcrypto.so.1.0.0 /usr/lib  
cp libssl.so.1.0.0 /usr/lib  
rm /usr/bin/openssl #删除以前的 openssl 软连接，不管是 openssl1.1 的还是 1.0.2 的  
ln -s /usr/local/bin/openssl /usr/bin/openssl  
root@ubuntu:/home/xiang/BC28serverTest/ssl/openssl-1.0.1q# openssl version  
OpenSSL 1.0.1q 3 Dec 2015  
root@ubuntu:/home/xiang/BC28serverTest/ssl/openssl-1.0.1q# make install
```

这就是 openssl 安装成功

然后在 mosquitto-1.6.8 目录下 make 。最后 sudo make install 记住一定要 sudo

```
apt-get install uuid-dev  
apt-get install libc-ares-dev  
mosquitto.c:46:18: fatal error: tcpd.h: No such file or directory 该  
问题需要安装 libwrap0-dev  
sudo apt install libwrap0-dev
```

### mosquito 库使用

进入 /etc/mosquitto 目录

touch mosquitto.conf 最好是将 /etc/mosquitto 目录下的 mosquitto.conf.example 拷贝出一份改成 mosquitto.conf

```
root@ubuntu:/etc/mosquitto# ls  
aclfile.example mosquitto.conf mosquitto.conf.example pskfile.example pwfile.example
```

然后修改 mosquitto.conf

```
198 #user mosquitto  
199 user xiang
```

user 是关键字不能变，“xiang”是你当前用户名你自己取，如果不设置这一步，启动 mosquitto -c mosquitto.conf 会报错，**报错信息：Error: Invalid user 'mosquitto'** 这个“xiang”一定要和你 ubuntu 系统自己用户的名称一样。

```
547 # retained_persistence  
548 persistence true # 持久化功能的开关 persistence true
```

这是最基本的修改，修改完成后可以保存了。如果要修改端口在 mosquito.conf 文件中

```
# Port to use for the default listener.  
port 1884
```

这样就改成了 1884 端口，和其它 MQTT 用户不冲突

执行 mosquitto -c mosquitto.conf

```
root@ubuntu:/etc/mosquitto# mosquitto -c mosquitto.conf  
1581833054: mosquitto version 1.6.8 starting  
1581833054: Config loaded from mosquitto.conf.  
1581833054: Opening ipv4 listen socket on port 1883.  
1581833054: Opening ipv6 listen socket on port 1883.
```

你看 mosquitto 中间服务程序已经成功执行，如果你要重新设置该服务程序的 ip 地址和端口，就在 mosquitto.conf 里面去修改。mosquitto.conf 这个文件的细节后面再讲

另开一个终端执行 mosquitto\_sub -t mqtt //这个-t mqtt mqtt 就是你的主题

出现执行错误 mosquitto\_sub: error while loading shared libraries:  
libmosquitto.so.1: cannot open shared object file: No such file or directory

我们修改动态链接

```
sudo ln -s /usr/local/lib/libmosquitto.so.1 /usr/lib/libmosquitto.so.1  
ldconfig
```

重新执行 mosquitto\_sub -t mqtt

```
^Croot@ubuntu:/home/xiang/BC28serverTest/mosquitto/mosquitto-1.6.8# mosquitto_sub -t mqtt
```

现在客户端的主题发布成功，现在等待另外一个客户端订阅该主题然后发数据

```
root@ubuntu:/home/xiang# mosquitto_pub -t mqtt -m "xzzz"
```

这是另外一个客户端订阅主题，然后发送 xzzz 数据

```
^Croot@ubuntu:/home/xiang/BC28serverTest/mosquitto/mosquitto-1.6.8# mosquitto_sub -t mqtt  
"xzzz"
```

我发布主题的客户端收到了另外一个订阅我主题客户端的数据

mosquitto 这个中间库调试成功

## python 在 windows 下实现 MQTT 客户端

```
import paho.mqtt.client as mqtt
import time

def on_connect(client, userdata, flags, rc):
    if rc == 0:
        print("连接成功")
        print("Connected with result code " + str(rc))

def on_message(client, userdata, msg):
    print(msg.topic + " " + str(msg.payload))

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
client.connect(host="192.168.1.12", port = 1883, keepalive=60) # 订阅频道
while True:
    client.publish("mqtt", "xxxxxxxxxx", 0) # 向主题为'mqtt'发布,内容为'xxxxxxxxxx'的信息
    time.sleep(1)
```

`mqtt.Client(client_id="", clean_session=True, userdata=None, protocol=MQTTv311, transport="tcp")`//该函数是给程序分配 1 个 MQTT 客户端对象，使用这个对象操作 MQTT

参数	含义
<code>client_id</code>	连接到代理时使用的唯一客户端 ID 字符串。如果 <code>client_id</code> 长度为零或无，则会随机生成一个。在这种情况下， <code>clean_session</code> 参数必须为 <code>True</code> 。
<code>clean_session</code>	一个决定客户端类型的布尔值。如果为 <code>True</code> ，那么代理将在其断开连接时删除有关此客户端的所有信息。如果为 <code>False</code> ，则客户端是持久客户端，当客户端断开连接时，订阅信息和排队消息将被保留。
<code>userdata</code>	用户定义的任何类型的数据作为 <code>userdata</code> 参数传递给回调函数。它可能会在稍后使用 <code>user_data_set()</code> 函数进行更新。
<code>protocol</code>	用于此客户端的 MQTT 协议的版本。可以是 <code>MQTTv31</code> 或 <code>MQTTv311</code> 。
<code>transport</code>	设置为“ <code>websockets</code> ”通过 WebSockets 发送 MQTT。保留默认的“ <code>tcp</code> ”使用原始 TCP。

`client.on_connect = on_connect(client, userdata, flag, rc)` ←  
//`on_connect` 是自己定义的回调函数，连接 MQTT 服务器成功的时候会调用这个回调函数，

`client`: 填入 `client` 句柄，表示你操作的是自己定义的客户端，当然你也可以用 `mqtt.clinet` 多定义几个客户端，多做几个句柄。

`userdata`: `userdata` 可以是任何类型的数据，可以在创建新客户端实例时设置或者直接使用 `user_data_set(userdata)` 函数设置，就是外部传参数进

`on_connect` 回调函数

`rc`: 决定了连接成功或不成功

rc 值	连接情况
0	连接成功
1	协议版本错误
2	无效的客户端标识
3	服务器无法使用
4	错误的用户名或密码
5	未经授权



```
client.on_message = on_message(client, userdata, message)
```

client: 填入 client 句柄, 表示你操作的是自己定义的客户端

userdata: 同上页一样

message: 收到服务器发来的数据

msg. topic 返回收到数据是用的是哪个主题

msg. payload 返回收到的数据值, 用 str(msg. payload) 转换成字符串



on\_message接受

数据, 数据就放在

msg变量

```
client.connect(host="192.168.1.12", port = 1883, keepalive=60)
//连接 MQTT 服务器, 填上 ip 地址和端口
```

参数	含义
host	远程代理的主机名或 IP 地址
port	要连接的服务器主机的网络端口。 默认为 1883
keepalive	与代理通信之间允许的最长时间段（以秒为单位）。 如果没有其他消息正在交换，则它将控制客户端向代理发送 ping 消息的速率
bind_address	假设存在多个接口, 将绑定此客户端的本地网络接口的 IP 地址

```
client.publish(topic, payload, qos, retain=False) # 向主题为...发布,
内容为'...'的数据, 也就是发数据给服务器了, 类似 socket 的 send
```

参数	含义
topic	该消息发布的主题
payload	要发送的实际消息。如果没有给出, 或设置为无, 则将使用零长度消息。传递 int 或 float 将导致有效负载转换为表示该数字的字符串。如果你想发送一个真正的 int / float, 使用 struct.pack() 来创建你需要的负载
qos	服务的质量级别, 这个 qos 要和 MQTT 服务器客户端质量级别一样, 不然会出问题
retain	如果设置为 True, 则该消息将被设置为该主题的“最后已知良好”/保留的消息



你看服务端 qos 和客户端 qos 等级一样, 这个 publish 可以循环发送数据给服务端

## 在 linux 上实现有 MQTT 服务器的客户端程序, 也就是服务器上的客户端来处理其它连接上的客户端

使用 paho 之前, 先要在 ubuntu 下安装 paho

```
apt install python-pip
pip install paho-mqtt
  100% |██████████| 102kB 17kB/s
Building wheels for collected packages: paho-mqtt
  Running setup.py bdist_wheel for paho-mqtt ... done
  Stored in directory: /root/.cache/pip/wheels/75/e2/f5/
Successfully built paho-mqtt
Installing collected packages: paho-mqtt
Successfully installed paho-mqtt-1.5.1
```

安装好的 paho-mqtt-1.5.1 很有可能默认在 python2 路径下, 导致使用 python3 解释器启动 MQTT 程序报错 **ImportError: No module named 'paho'**

最简单的方法, 先安装 python3 的 pip 下载器, apt install python3-pip

用 pip3 install paho-mqtt 再安装一次 paho

```
root@ubuntu:/home/xzz/pyserver# pip3 install paho-mqtt
Collecting paho-mqtt
  Using cached https://files.pythonhosted.org/packages/32/d3/6dcb8fd14746fc372c0e7/paho-mqtt-1.5.1.tar.gz
Building wheels for collected packages: paho-mqtt
  Running setup.py bdist_wheel for paho-mqtt ... done
  Stored in directory: /root/.cache/pip/wheels/75/e2/f5/78942b19b4d135605e5
Successfully built paho-mqtt
Installing collected packages: paho-mqtt
Successfully installed paho-mqtt-1.5.1
You are using pip version 8.1.1, however version 21.0.1 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
```

然后先启动 mosquito -c mosquitto.conf 中间程序

再跑下面自己写的 MQTT 服务器程序, 运行 OK。

```

import paho.mqtt.client as mqtt
import time

def on_connect(client, userdata, flags, rc):
    if rc == 0:
        print("连接成功")
        print("Connected with result code " + str(rc))

def on_message(client, userdata, msg):
    print(msg.topic + " " + str(msg.payload))

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
client.connect(host="192.168.1.12", port = 1883, keepalive=60) # 订阅频道

client.subscribe('xzzmqtt', qos=0)
client.loop_start()
while True:
    pass

```

Client.on\_connect 连接本服务器上面的 MQTT，Client.on\_message 接受客户端发到服务器来的数据，Client.connect 连接上 MQTT 服务器，这些过程和上一节客户端是一样的。



client.subscribe 这里不一样，这里就是自定义主题名称，所以叫做订阅设置，因为这里的设置的主题，客户端会订阅，客户端一旦定义该主题，那么客户端连接上服务器就知道给哪个主题发数据了。

client.subscribe(topic, qos)

第 1 种写法

subscribe("my/topic", 2)

参数	值
topic	一个字符串，指定要订阅的订阅主题
qos	期望的服务质量等级。默认为 0。

第 2 种写法

subscribe(("my/topic", 1))

参数	值
topic	(topic, qos) 的元组。主题和 qos 都必须存在于元组中。
qos	没有使用

第 3 种写法

这允许在单个 SUBSCRIPTION 命令中使用多个主题订阅，这比使用多个订阅 subscribe() 更有效。

subscribe([( "my/topic", 0), ("another/topic", 2)])

参数	值
topic	格式元组列表 (topic, qos)。topic 和 qos 都必须出现在所有的元组中。
qos	没有使用

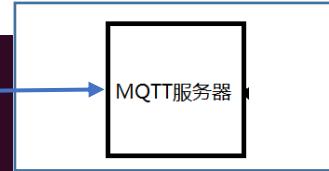
client.loop\_start() //在 connect 连接之后调用一次 loop\_start 会在后台运行一个线程来处理 loop, 你才可以使用 while 循环让服务器客户端不被执行结束。你才能接受客户端发来的信息。

后面的 loop\_start() 使用问题会详细介绍，一定要看。

loop\_stop() # 来停止后台线程。

下面实验执行流程

```
root@ubuntu:/etc/mosquitto# mosquitto -c mosquitto.conf
1581854576: mosquitto version 1.6.8 starting
1581854576: Config loaded from mosquitto.conf.
1581854576: Opening ipv4 listen socket on port 1883.
1581854576: Opening ipv6 listen socket on port 1883.
```



mosquito 启动之后，启动服务端的客户端程序

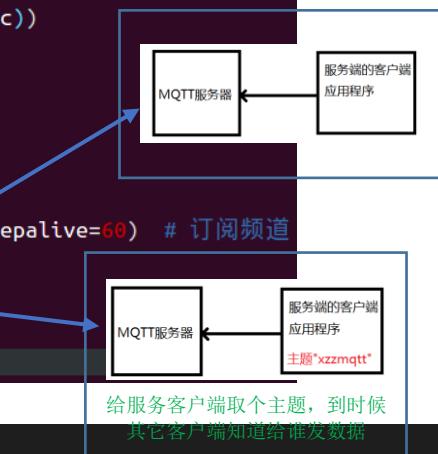
```
import paho.mqtt.client as mqtt
import time

def on_connect(client, userdata, flags, rc):
    if rc == 0:
        print("连接成功")
        print("Connected with result code " + str(rc))

def on_message(client, userdata, msg):
    print(msg.topic + " " + str(msg.payload))

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
client.connect(host="192.168.1.12", port = 1883, keepalive=60) # 订阅频道

client.subscribe('xzzmqtt',qos=0)
client.loop_start()
while True:
    pass
```



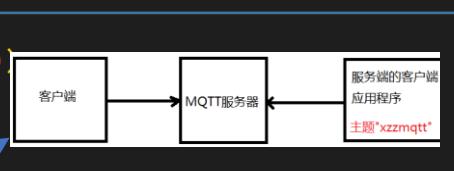
```
import paho.mqtt.client as mqtt
import time

def on_connect(client, userdata, flags, rc):
    if rc == 0:
        print("连接成功")
        print("Connected with result code " + str(rc))

def on_message(client, userdata, msg):
    print(msg.topic + " " + str(msg.payload))

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
client.connect(host="192.168.1.12", port = 1883, keepalive=60) # 订阅频道

while True:
    client.publish("xzzmqtt", "xxxxzzzz", 0) # 向主题为'mqtt'发布,内容为'xxxxzzzz'
    time.sleep(1)
```

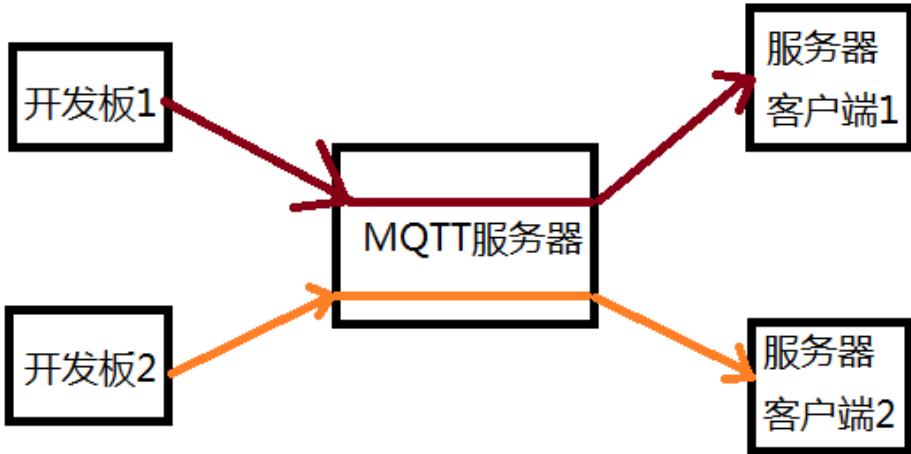


这样客户端的数据就发送到服务端的客户端了。

11 / 75 就订阅 xzzmqtt，然后发布数据

## 多个开发板客户端发数据到服务器上多客户端的案例

loop\_start() 使用问题



服务器客户端 1 代码

```
import paho.mqtt.client as mqtt
import time

def on_connect(client, userdata, flags, rc):
    if rc == 0:
        print("连接成功")
        print("Connected with result code " + str(rc))

def on_message(client, userdata, msg):
    print(msg.topic + " " + str(msg.payload))

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
client.connect(host="192.168.1.12", port = 1883, keepalive=60) # 订阅频道

client.subscribe("xzzmqtt", qos=1)
client.loop_start()
while True:
    pass
```

服务器客户端 2 代码

```
import paho.mqtt.client as mqtt
import time

def on_connect(client, userdata, flags, rc):
    if rc == 0:
        print("连接成功")
        print("Connected with result code " + str(rc))

def on_message(client, userdata, msg):
    print(msg.topic + " " + str(msg.payload))

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
client.connect(host="192.168.1.12", port = 1883, keepalive=60) # 订阅频道

client.subscribe("xiangmqtt", qos=1)
client.loop_start()
while True:
    pass
```

区分客户端发送给哪一个服务器客户端程序，就是用黄色框着的订阅主题编号

## 开发板客户端 1

```
def on_connect(client, userdata, flags, rc):
    if rc == 0:
        print("连接成功")
        print("Connected with result code " + str(rc))

def on_message(client, userdata, msg):
    print(msg.topic + " " + str(msg.payload))

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
client.on_disconnect = on_disconnect
client.connect(host="192.168.1.12", port = 1883, keepalive=60) # 连接

#client.loop_start()
while True:
    client.publish("xzzmqtt","xxxxxxxx",1) # 向主题为'xzzmqtt'发布,消息内容为xxxxxxxx,消息QoS为1
    time.sleep(1)
```

发送给主题为 xzzmqtt 的服务器客户端

## 开发板客户端 2

```
def on_connect(client, userdata, flags, rc):
    if rc == 0:
        print("连接成功")
        print("Connected with result code " + str(rc))

def on_message(client, userdata, msg):
    print(msg.topic + " " + str(msg.payload))

client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
client.on_disconnect = on_disconnect
client.connect(host="192.168.1.12", port = 1883, keepalive=60) # 连接

#client.loop_start()
while True:
    client.publish("xiangmqtt","12345678",1)
    time.sleep(1)
```

发送给主题 xiangmqtt 的服务器客户端

下面先启动服务器的两个服务客户端程序

### 服务客户端程序 1

```
root@ubuntu:/home/xiang/BC28serverTest# python3 BC28MQTTserver.py
连接成功
Connected with result code 0
```

### 服务客户端程序 2

```
root@ubuntu:/home/xiang/BC28serverTest# python3 BC28MQTTserver2.py
连接成功
```

因为 vscode 只能执行一个 py 文件的进程，所以我们用 windows 终端来模拟开发板执行两个开发板的程序

```
E:\Desktop\BC28_Project\python_client>python MQTTclient.py
```

开发板 1

```
E:\Desktop\BC28_Project\python_client>python MQTTclient2.py
```

开发板 2

现在我们发现开发板并没有打印连接 MQTT 服务器成功的字符，但是我上一张开发板也没有打印连接 MQTT 服务器成功的字符啊，所以这里先埋个伏笔后面再讲。

```
root@ubuntu:/home/xiang/BC28serverTest# python3 BC28MQTTserver.py
连接成功
Connected with result code 0
xzzmqtt b'xxxxxxxx'
```

```
root@ubuntu:/home/xiang/BC28serverTest# python3 BC28MQTTserver2.py
连接成功
Connected with result code 0
xiangmqtt b'12345678'
```

我发现两个开发板发送给 MQTT 服务器的，两个(服务客户端的程序)执行一半卡住了？

`loop_start() //函数使用`

```
client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
client.on_disconnect = on_disconnect
client.connect(host="192.168.1.12", port = 1883, keepalive=60) # 连接

client.loop_start() # 在循环发送前加入 loop_start
while True:
    client.publish("xzzmqtt","xxxxxxxx",1) # 向主题为"xzzmqtt"发布消息
    time.sleep(1)
```

开发板 1 加 `loop_start`

```
client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
client.on_disconnect = on_disconnect
client.connect(host="192.168.1.12", port = 1883, keepalive=60) # 连接

client.loop_start() # 在循环发送前加入 loop_start
while True:
    client.publish("xiangmqtt","12345678",1)
    time.sleep(1)
```

开发板 2 加 `loop_start`

```
E:\Desktop\BC28_Project\python_client>python MQTTclient.py
连接成功
```

```
Connected with result code 0
```

两个开发板也执行了连接成功回调函数。这是上一张单客户端和(服务的客户端)测试没遇到过的。

```
xzzmqtt b'xxxxxxxx'
```

```
xiangmqtt b'12345678'
```

你看两边服务客户端接受程序就不会卡住了，正常运行。

## loop\_start() 的意义

在 connect\* () 之前或之后调用 loop\_start () 一次，会在后台运行一个线程来自动调用 loop ()。  
这释放了可能阻塞的其他工作的主线程(所以刚才可能主线程被 connect 后阻塞了)。这个调用也处理重新连接到代理

loop\_start () 对应的还有 loop\_stop

调用 loop\_stop () 来停止后台线程

## disconnect() 使用

```
client = mqtt.Client()  
client.on_connect = on_connect  
client.on_message = on_message  
client.on_disconnect = on_disconnect  
client.connect(host="192.168.1.12", port = 1883, keepalive=60) # 订阅频道  
client.loop_start()  
while True:  
    client.publish("xiangmqtt", "12345678", 1)  
    time.sleep(1)  
  
client.loop_stop()  
client.disconnect()
```

如果直接关闭主程序是不用执行  
loop\_stop 和 disconnect 的，因为主程序  
关闭，链接和后台 loop 自动关闭断开  
但是如果该程序是用在 python 主程序的某  
个线程里面，线程结束了就一定要执行这  
两句，因为主程序没有结束，说不定还有  
再次 connect MQTT 服务器的可能

disconnect() 回调函数详解，可以用来处理 MQTT 服务器异常断开提示

```
client.on_disconnect = on_disconnect(client, userdata, rc)
```

当我执行 disconnect() 断开与 MQTT 服务器链接时，会执行 on\_disconnect  
回调函数

Client: 填入你创建客户端的句柄

userdata: 前面介绍过

rc: 返回 0 表示你执行了 disconnect 函数断开与 MQTT 服务器连接

返回 1 表示 MQTT 服务器异常自动断开，我客户端其实没有断开，但是因为  
服务器问题客户端被迫断开

```
def on_disconnect(client, userdata, rc):
    print("rc = %d" %(rc))
```

这是连接断开回调函数

```
client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
client.on_disconnect = on_disconnect
client.connect(host="192.168.1.12", port = 1883, keepalive=60)
client.loop_start()

while True:
    client.publish("xzzmqtt","xxxxxxxx",1) # 向主题为'xxxxxxxx'
    time.sleep(1)
    client.disconnect()
```

在我连接成功后

执行断开程序主动断开一次看看现象

```
E:\Desktop\BC28_Project>python -u "e:\Desktop\BC28_Project\python_client\MQTTclient.py"
连接成功
Connected with result code 0
rc = 0
```

你看我主动断开，on\_disconnect 回调函数 rc 返回的是

```
client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message
client.on_disconnect = on_disconnect
client.connect(host="192.168.1.12", port = 1883,
client.loop_start()

while True:
    client.publish("xzzmqtt","xxxxxxxx",1) # 向
```

我不主动断开，让服务器来断开

```
1581936756: Saving in-memory database to mosquitto.db.
1581936756: Error saving in-memory database, unable to open mosquitto.db.new for writing.
1581936756: Error: Permission denied.
root@ubuntu:/etc/mosquitto#
```

MQTT 服务器断开

```
E:\Desktop\BC28_Project>python -u "e:\Desktop\BC28_Project\python_client\MQTTclient.py"
连接成功
Connected with result code 0
rc = 1
```

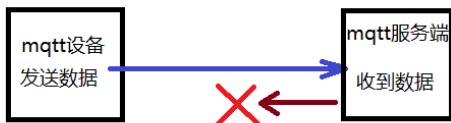
你看客户端 disconnect 的 rc 返回 1

我们可以用这个方法来判断服务器的 MQTT 是否断开

## QOS 重点讲解

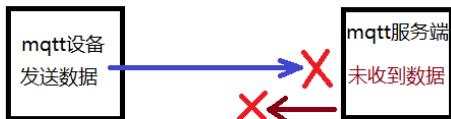
Qos = 0 最多发一次

QOS = 0



MQTT客户端向服务端发数据，服务端收到，但是服务端不回发数据

QOS = 0 还可以如下理解



如果mqtt客户端向服务端发数据，服务端未收到数据

服务端也不回数据给mqtt客户端

那么MQTT客户端也不知道数据发送成功没有

所以QOS = 0 就是针对一些数据要求不高的场合设置的。比如

客户端实时发数据给服务端，服务端也不在乎中间有数据丢包

的情况。这种场合就适合用QOS = 0

实际操作过程:



QOS = 1 最少发一次

QOS = 1



如果客户端发数据，服务端没有回呢？



如果服务端没有回信息，那么客户端  
会再重复发一次数据给服务端



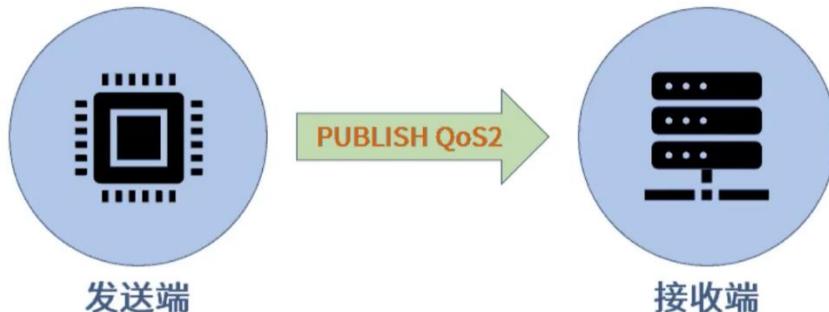
如果服务端还是没有回



客户端会继续重发，发到服务端回复为止，才停止发送

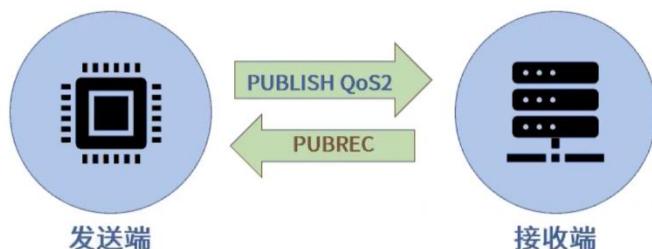
QOS = 1 就适合用在要求通信质量比较高的场景，要求客户端和服务端数据不丢包的应用下使用 QOS = 1。

$QoS = 2$  保证收一次，就更严格了



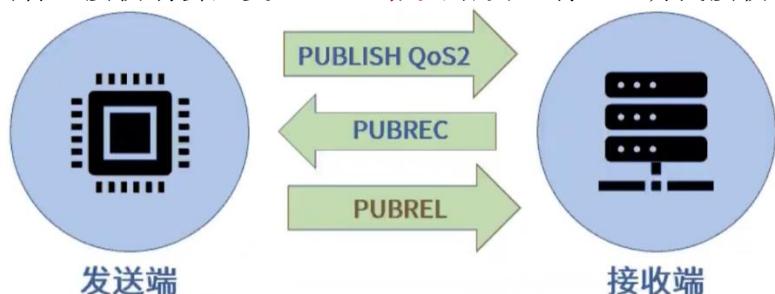
MQTT QoS2 PUBLISH

发送端或客户端，发数据给服务端或者接收端



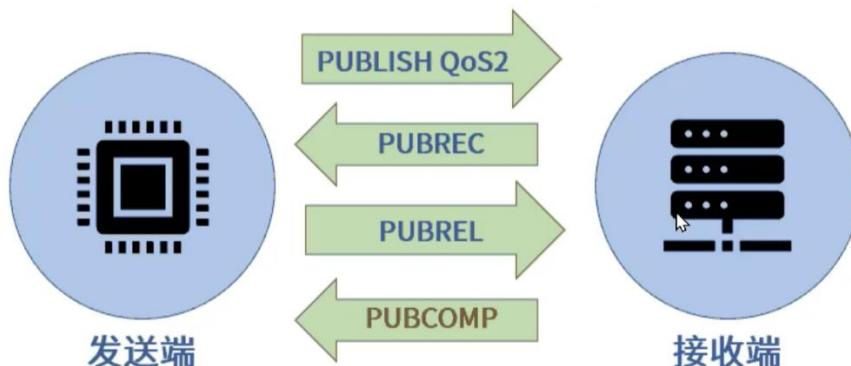
MQTT QoS2 PUBREC

接着，接收端会回复 PUBREC 报文给发送端。证明我接收端接收到数据了。



MQTT QoS2 PUBREL

然后发送端还会发送 PUBREL 给接收端，证明我发送端收到了接收端的回复。

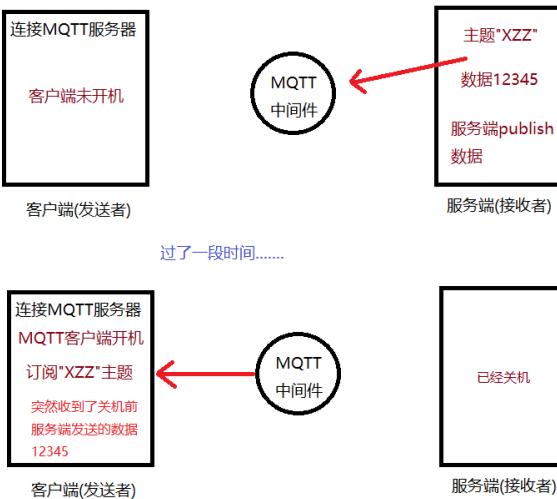


MQTT QoS2 PUBCOMP

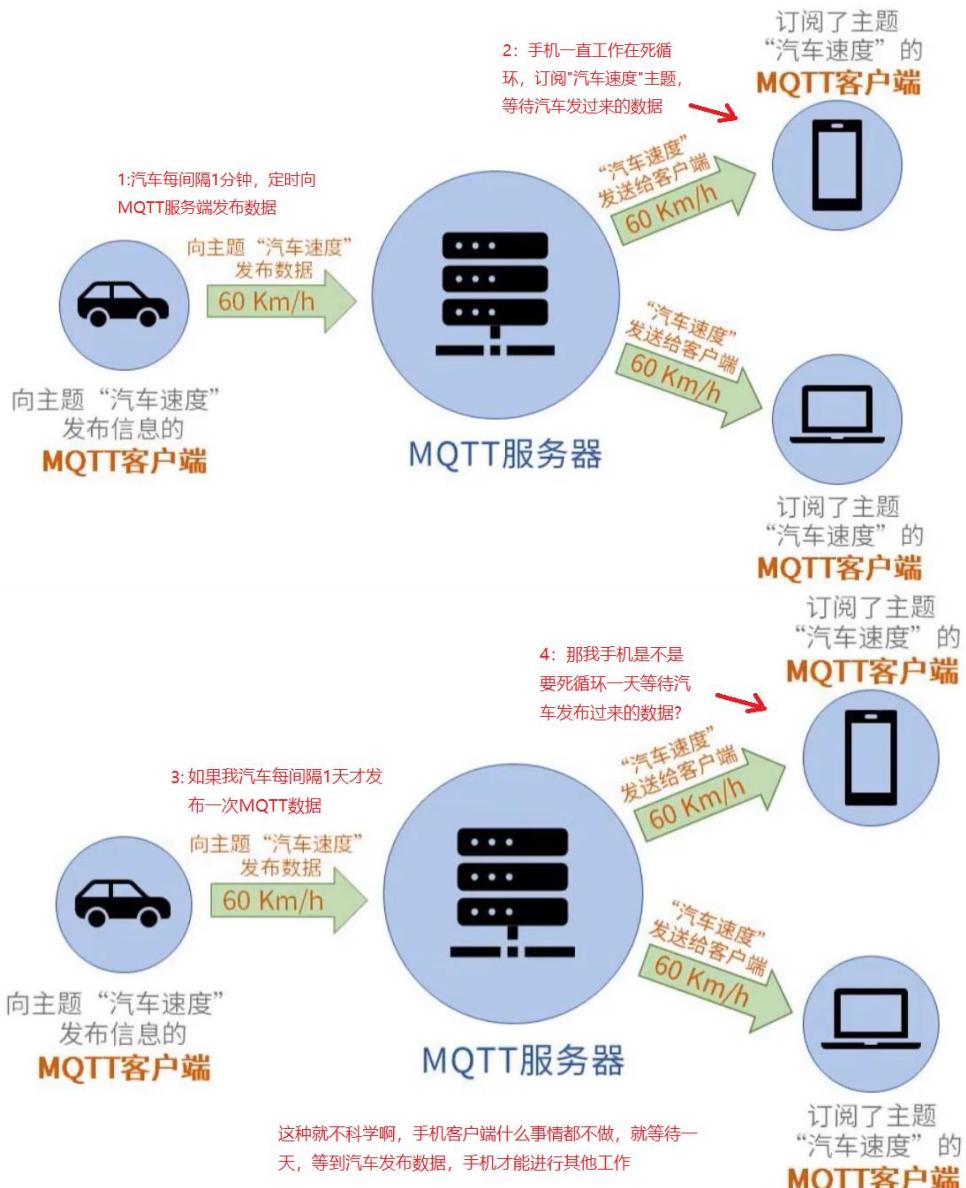
服务端最后会再回复一次 PUBCOMP 报文给接收端，证明我收到接收端确认信息。至此一帧 MQTT 信息传输完成。

这就是要求极高的通信场合下使用的 QOS 等级。一般都用  $QoS = 1$ 。有些用  $QoS = 0$ ，这样可以提升传输速度。

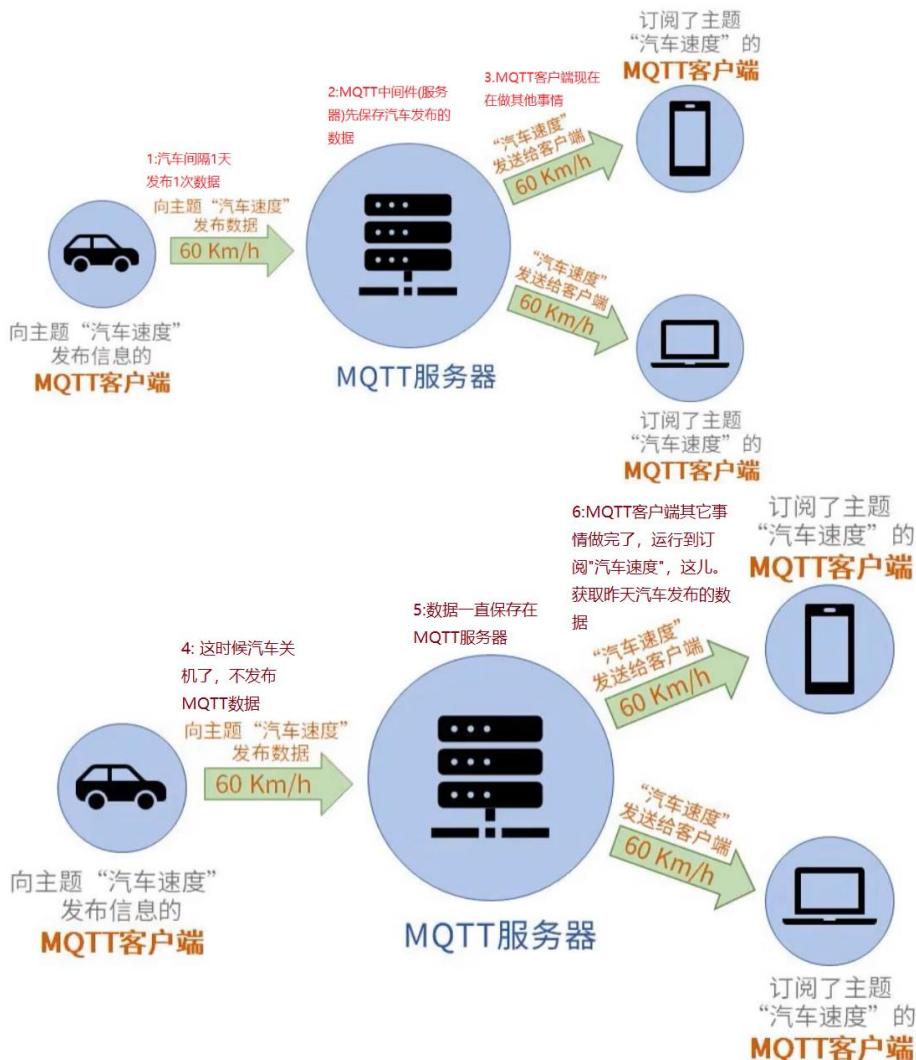
## 保留消息，就是订阅主题就会马上收到上一次其它设备发布的指定主题消息



这种保留消息有什么实际的应用场景呢？

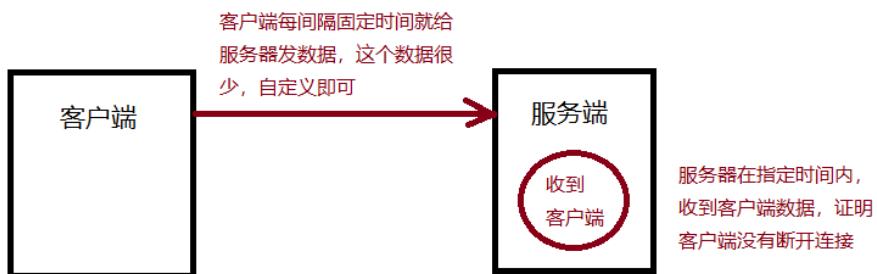


所以我们可以使用保留消息这种机制



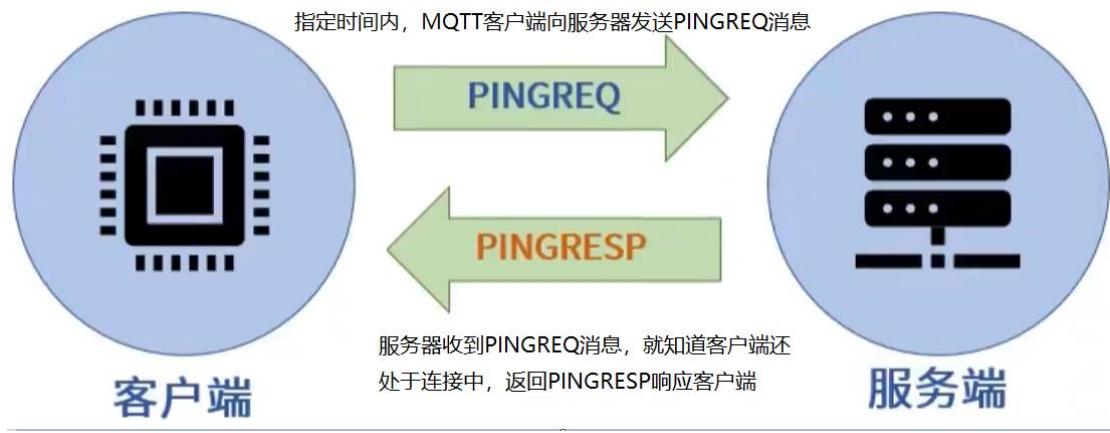
## MQTT 心跳机制

常规心跳方式



但是有些应用场景客户端只收数据，不向服务器发送数据，那么服务器如何认为客户端与服务端还保持连接呢？

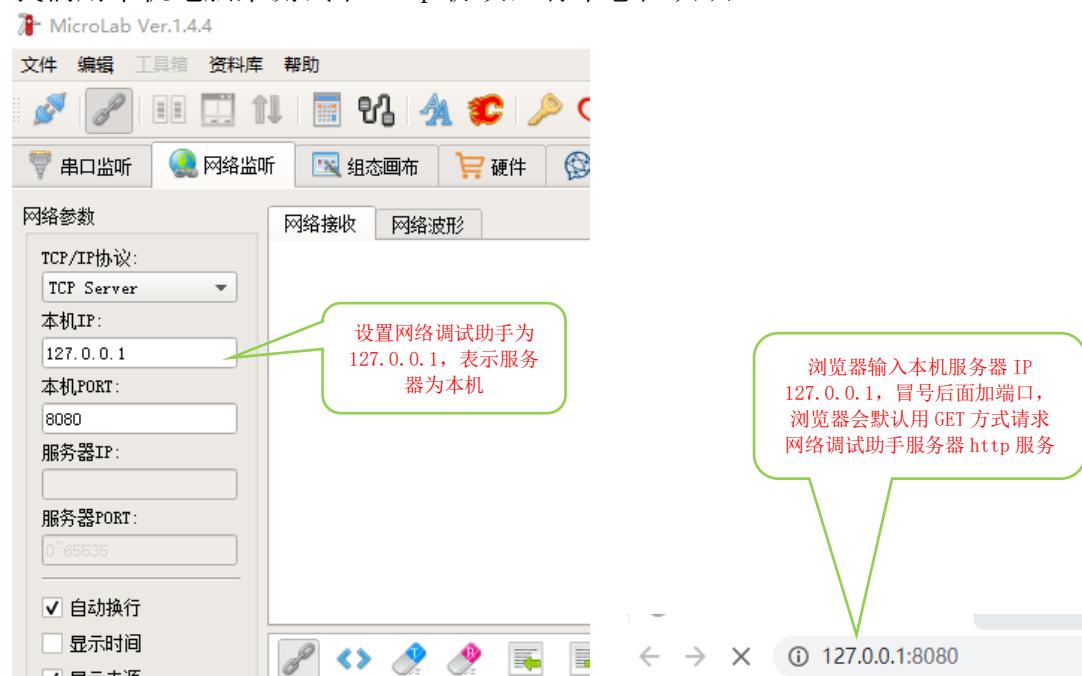
所以MQTT有个专用的心跳包，专门用来给服务器发数据，让服务器知道，客户端还没有断开



## HTTP 协议 Python 实现

http 协议必须基于 socket TCP 来实现

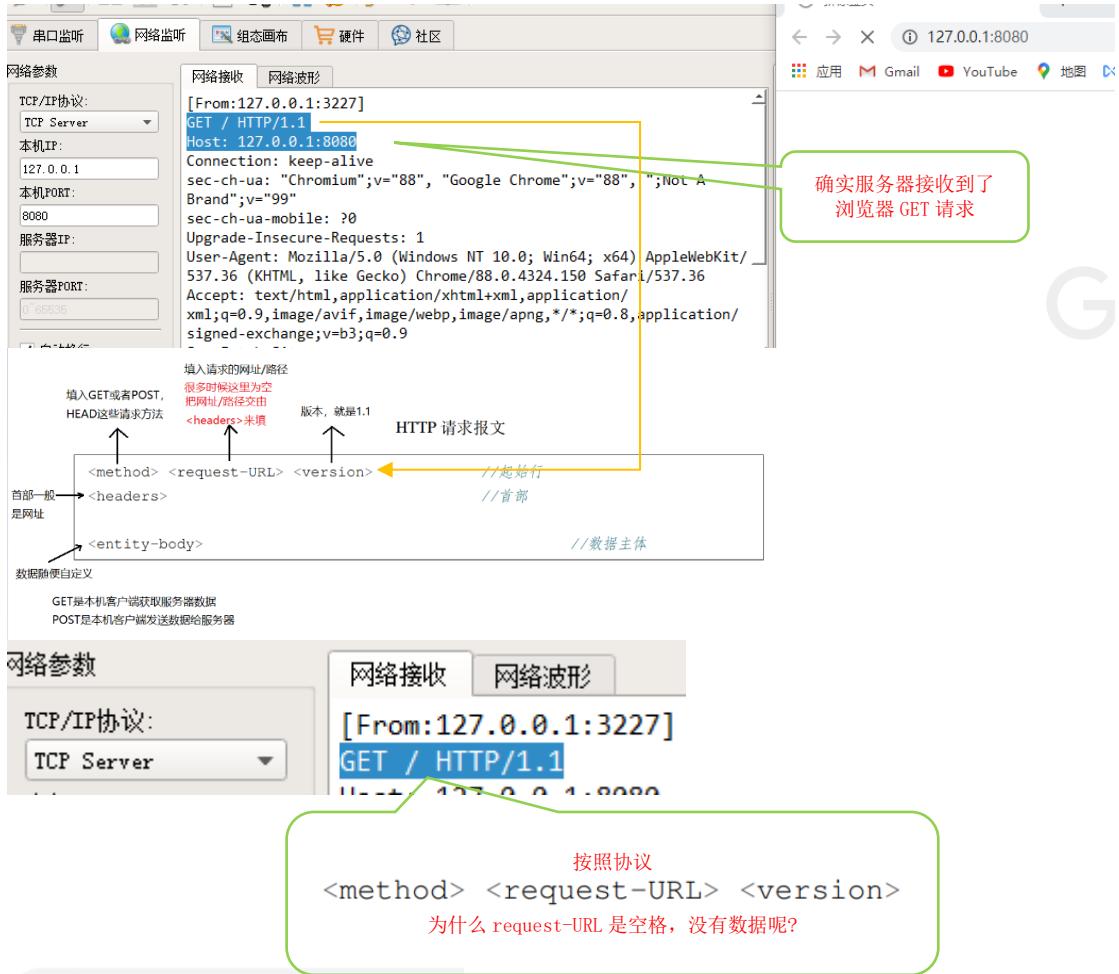
我们用本机电脑来测试下 http 协议，有个感性认识



GET 请求流程如下：



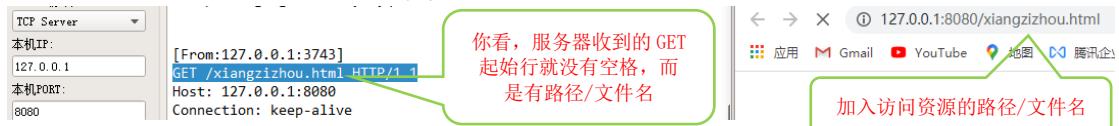
浏览器点击回车，看看是不是这个流程



如果浏览器访问服务器的同时，不在后面追加你要访问的内容，比如 xxxx.html 文件，图片，音频，视频文件。那么你服务器得到的就是

GET / HTTP/1.1 这种内容

我在浏览器加入访问资源的路径



所以 GET /xiangzizhou.html HTTP/1.1 表示浏览器需要服务器提供 xiangzizhou.html 文件

GET /xiangzizhou.html HTTP/1.1

Host: 127.0.0.1:8080

Host 表示服务器 IP 地址和端口

Connection: keep-alive

keep-alive 表示长连接

Accept: text/html,application/xhtml+xml,application/

Accept 表示浏览器能接受什么样格式的数据。这里能接受 text/html, application/xhtml+xml...

下面实现浏览器 GET 请求 http，服务器响应浏览器 http 功能代码

```
import socket
from socket import *

def http_client_response(new_socket):
    request = new_socket.recv(1024)
    print(request)

    response = "HTTP/1.1 200 OK\r\n"
    response += "\r\n"
    response += "<h1>xxxxxxxx</h1>"
    new_socket.send(response.encode('utf-8'))

    new_socket.close()

def main():
    httpserver = socket(AF_INET, SOCK_STREAM)
    httpserver.bind(('', 8888))
    httpserver.listen(128)

    while True:
        newsocket, addr = httpserver.accept()
        http_client_response(newsocket)

    httpserver.close()

if __name__ == "__main__":
    main()

response = "HTTP/1.1 200 OK\r\n"
response += "\r\n"
response += "<h1>xxxxxxxx</h1>"
new_socket.send(response.encode('utf-8'))
```

1. http 服务器接收到浏览器的 TCP 连接

2. 进入响应函数，响应浏览器

3. 打印浏览器发过来的数据，理论上这里该解析字符串含义，我为了简便就不解析了

4. 我按照浏览器和 html 标准，返回字符串给浏览器显示，数据格式如下

版本 状态码 失败原因 HTTP 应答报文

<version> <status> <reason-phrase>

<headers>

<entity-body>

两次换行之后，写入 html 文件内容

//起始行 //首部 //数据主体

根据格式这里需要换行，因为浏览器只识别 \r\n 表示换行，所以我用 \r\n 换行

启动 http 服务器

inet addr:192.168.0.104

服务器 IP 地址

▲ 不安全 | 192.168.0.104:8888 浏览器请求 IP 和端口，标准 http 是 80 端口，我这里为了测试，自定义 8888 端口

```
root@ubuntu:/home/xzz/sleep_box/sleep_httpserver# python3 httpserver.py
b'GET / HTTP/1.1\r\nHost: 192.168.0.104:8888\r\nConnection: keep-alive\r\nUpgrade-Insecure-Request: 1\r\nUser-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/88.0.4324.150 Safari/537.36\r\nAccept: */*\r\nAccept-Encoding: gzip, deflate\r\nAccept-Language: zh-CN,zh;q=0.9\r\n\r\n'
b'GET /favicon.ico HTTP/1.1\r\nHost: 192.168.0.104:8888\r\nConnection: keep-alive\r\nUpgrade-Insecure-Request: 1\r\nUser-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/88.0.4324.150 Safari/537.36\r\nAccept: */*\r\nAccept-Encoding: gzip, deflate\r\nAccept-Language: zh-CN,zh;q=0.9\r\n\r\n'
```

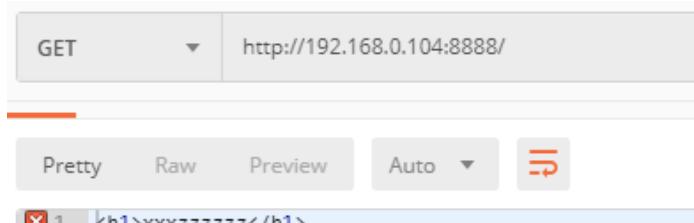
服务器得到浏览器发来的数据

# XXXZZZZZZ

```
response = "HTTP/1.1 200 OK\r\n"
response += "\r\n"
response += "<h1>xxxzzzzz</h1>"
new_socket.send(response.encode('utf-8'))
```

服务器返回数据给浏览器

我用 postman 测试也一样的效果



这就是 http 客户端和 http 服务器整个通信流程。

## JSON 概念与使用

JSON基本的样子

{ "fiestName" : "xzz" , "lastName" : "xiangzizhou" }    这就是JSON基本格式  
    变量名    对应的数据    变量名    对应的数据

其实和key value思想很接近，就是一个变量名，对应一个数据，需要获取某个数据，先传入该数据的变量名，然后获取变量名对应的数据

json基本格式就是大括号开始，

然后大括号结尾

逗号间隔表示下一个数据

{ "fiestName" : "xzz" , "lastName" : "xiangzizhou" }

用冒号进行key value分割

JSON数组写法

```
{
  "array" : [
    { "fiestName" : "xzz" , "lastName" : "xiangzizhou" },
    { "number" : 500, "str" : 1.254},           JSON支持十进制数字和小数, (不支持16进制)
    { "id" : 0,   "bool" : true}                JSON支持bool值
  ]
}
```

获取数据方法 数组名[ ].变量

array[1].number 得到500

也可以 array[1].number = 100 修改JSON值

将数据从Python格式转换成json格式的时候，数据会发生变化，变化如下：

python类型  JSON类型

dict(字典)	object(对象)
list(列表), tuple(元组)	array(数组)
str(字符串)	string(字符串)
int, float	number(数字)
True	true
False	false
None	null

Json.dumps(变量) //将 python 数据类型转成字符串

返回字符串

```
import json

person = {"name": "Sniper", "age": 30, "tel": ["1234567", "87654321"], "isonly": True}
#用python数据类型按照JSON格式定义
print(person)

jsonStr = json.dumps(person) #将python数据类似转成json字符串

print(jsonStr)
```

{'isonly': True, 'age': 30, 'tel': ['1234567', '87654321'], 'name': 'Sniper'}

python 数据类型

{"isonly": true, "age": 30, "tel": ["1234567", "87654321"], "name": "Sniper"}

转成了 json 字符串

jsonStr = json.dumps(person, indent=4) #indent 将 json 数据按列格式化输出

```
import json

person = {"name": "Sniper", "age": 30, "tel": ["1234567", "87654321"], "isonly": True}
print(person)

jsonStr = json.dumps(person, indent=4)

print(jsonStr)
{
    "isonly": true,
    "age": 30,
    "tel": [
        "1234567",
        "87654321"
    ],
    "name": "Sniper"
}
```

这就是格式化之后的输出，方便眼睛看。

Json.loads(字符串变量) #将 json 字符串转换成 python 使用的字典

返回字典

# Python 从头实现 http 服务器

http 协议前面章节有讲解，这里只是大概介绍下 URL 地址解析

URL就是网站 <https://news.163.com/18/1122/10/E178.html>

URL组成部分: https://, http://, ftp://

http是明文传输，就是原始的网页字符，直接转换成16进制传输

域名部分：如news.163.com

资源路径部分: /18/1122/10/E178.html

注意: 如果是https, 那么默认端口是443

如果是http，那么默认端口号是80

比如我想请求服务器发给我E178.html文件

<https://news.163.com/18/1122/10/E178.html>

我就去news.163.com服务器下，目录18/112/10路径下去找



▼ General  
Request URL: http://www.baidu.com/  
Request Method: GET  
Status Code: 302 Found ← 302表示在服务器找到数据  
Remote Address: 14.215.177.38:80  
Referer Policy: no-referrer-when-downgrade

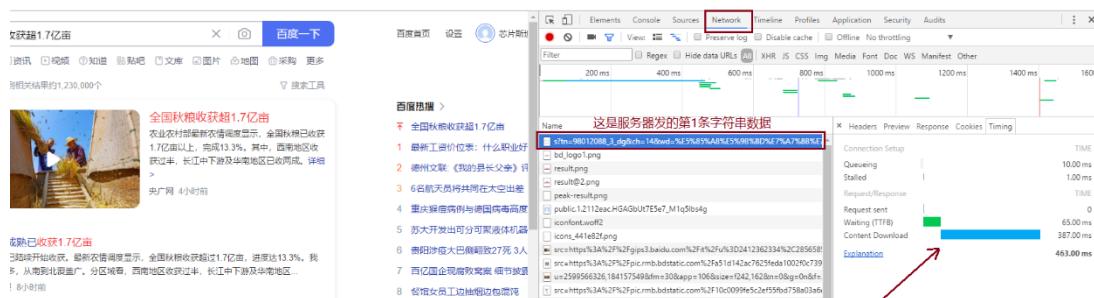
▼ Response Headers [view source](#)

Connection: keep-alive  
Content-Length: 154  
Content-Type: text/html  
Date: Mon, 19 Sep 2022 12:30:24 GMT  
Location: https://www.baidu.com/  
Server: BWS/1.1  
Set-Cookie: BD\_LAST\_QID=10923581600285825604; path=/; Max-Age=3600  
Traceid: 16635090624027187252110923581600285825604  
X-Frame-Options: sameorigin  
X-UA-Compatible: IE=Edge,chrome=1

2. 服务器回发给我设备的数据

1.这是我浏览器(也就是设备)TCP发送的字符串

POST 可以获取服务器数据到设备，POST 也可以设备提交数据给服务器



GET命令之后，得到的每一条网页数据都在这里

设备请求数据就会到服务器的 Response

设备请求就会发 Request 数据

Request Headers

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,\*/\*;q=0.8

你看后面的数据都是服务器发的了，设备端没有 Request

你看后面的数据都是服务器发的了，设备端没有 Request

这就是设备请求 http 服务器的过程。

### 状态码介绍

- 200 : 就是获取服务器数据成功
- 307 : 重定向，改变网页请求地址，比如我输入的是http://XXXXXX，服务器自动请求到https://XXXXXX
- 400 : 错误的请求，看到400，要检查请求网页地址写错误没有，或者浏览器里面文件路径参数写对没有
- 404 : 请求的服务器成功，但是服务器指定的目录下没有你要的文件
- 500 : 500就是服务器自己的错误，找后端解决

## HTTP 静态服务器开发

采用 TCP 底层开发 HTTP 服务器，所以我们用 socket 来开发。

```
root@ubuntu:/home/xzz/pyserver# ls
httpserver.py      mosquitto      pyMQTTserver.py   tcpclient.py    tcpserver.py  xzz
httpsockserver.py  paho.mqtt.c    pytest.py       tcpserver2.py  udpserver.py
```

浏览器获取服务器当前 xzz 目录下的 index.html 文件

http 静态服务器程序采用 httpsockserver.py 编写

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <title> Welcome xzz </title>
5 <h1> welcome to xzz html </h1>
6 </head>
7 </html>
8
```

index.html 内容

```
1 import socket
2
3 if __name__ == '__main__':
4     tcp_server = socket.socket(socket.AF_INET,socket.SOCK_STREAM) #创建TCP服务器SOCKET套接字
5     tcp_server.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,True) #绑定端口号
6     tcp_server.bind(("0.0.0.0",8888)) #默认使用本机IP, http协议端口官方要求80, 但是我可以自定义为8888, 只要浏览器指定8888就行
7     tcp_server.listen(128) #设置监听
8
9     while True:
10         new_socket, ip_port = tcp_server.accept() #等待接收客户端的连接请求,记住这种写法, 客户端连接一个, 后面的客户端只有等待前一个客户端请求完, 才能再连接下一个客户端
11         recv_data = new_socket.recv(4096) #如果网页采用的GET请求, 那么网页客户端发的数据最多不会超过4K
12         print(recv_data) #将接收的网页数据打印出来
13
14         with open("xzz/index.html","r") as file: #with open的好处就是打开指定文件, 不需要手动close关闭文件
15             file_data = file.read() #读取指定index.html文件内容
16
17         #服务器不是随随便便将html文件发给浏览器就可以了, 必须有响应体来做帧头
18
19         response_line = "HTTP/1.1 200 OK\r\n" #响应行
20         response_header = "Server: PWS/1.0\r\n" #响应头
21
22         response_null = "\r\n" #空行
23         response_body = file_data #响应体(响应体就是发的html文件, 也可以是txt, 或者其它数据)
24
25         response = response_line + response_header + response_null + response_body #经过响应头+空行+数据的组合, 现在数据被封装成了http报文格式
26
27         new_socket.send(response.encode("utf-8")) #将封装的数据发出去
28
29         new_socket.close() #关闭本次为客户端创建的socket
30
```

▲ 不安全 | 192.168.0.112:8888

浏览器发送请求

```
b'GET / HTTP/1.1\r\nHost: 192.168.0.112:8888\r\nCookie: la/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4453.113 Safari/537.36\r\nAccept: */*\r\nAccept-Encoding: gzip, deflate\r\nAccept-Language: zh-CN,zh;q=0.9\r\n\r\nReferer: \r\n\r\n'
```

服务器收到浏览器发来的数据

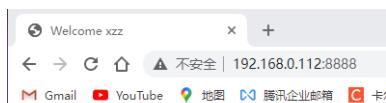
# welcome to xzz html

服务器返回给浏览器的

index.html 内容

以上服务器程序有个问题，就是不管浏览器发送任何字段的请求，服务器都只返回 index.html。

静态服务器浏览器请求什么页面，服务器就返回指定的页面



# welcome to xzz html

浏览器一旦发送了 Get 请求

```
root@ubuntu:/home/xzz/pyserver# python3 httpsockserver.py
```

```
b'GET / HTTP/1.1\r\nHost: 192.168.0.112:8888\r\nConnection: keep-alive\r\nUpgrade-Insecure-Requests: 1\r\nUser-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/105.0.0.0 Safari/537.36\r\nAccept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9\r\nAccept-Encoding: gzip, deflate\r\nAccept-Language: zh-CN,zh;q=0.9\r\n\r\nb'GET /favicon.ico HTTP/1.1\r\nHost: 192.168.0.112:8888\r\nConnection: keep-alive\r\nUser-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/105.0.0.0 Safari/537.36\r\nAccept: image/avif,image/webp,image/apng,image/svg+xml,image/*,*/*;q=0.8\r\nReferer: http://192.168.0.112:8888\r\nAccept-Encoding: gzip, deflate\r\nAccept-Language: zh-CN,zh;q=0.9\r\n\r\nb''
```

这就是服务器收到浏览器的原始报文数据。我们现在要利用原始报文的数据字段，来判断服务器发送什么样的内容给浏览器。

空格1个 空格1个  
b'GET /favicon.ico HTTP/1.1\r\nHost: 192.168.0.112:8888\r\nConnection: keep-alive\r\nUser-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/105.0.0.0 Safari/537.36\r\nAccept: image/avif,image/webp,image/apng,image/svg+xml,image/\*,\*;q=0.8\r\nReferer: http://192.168.0.112:8888/\r\nAccept-Encoding: gzip, deflate\r\nAccept-Language: zh-CN,zh;q=0.9\r\n\r\n' 浏览器发送过来的数据

我们以空格为间隔，将每一段字符截取出来

形成以下这种方式：

```
/favicon.ico  
HTTP/1.1\r\nHost:  
192.168.0.112:8888\r\nConnection:
```

```
9     while True:  
10    new_socket, ip_port = tcp_server.accept() #等待接收客户端的连接请求  
11    recv_data = new_socket.recv(4096)          #如果网页采用的GET请求，那么网页客户端发的数据最多不会超过4K  
12    #      print(recv_data)                  #接收的浏览器数据就不打印了，免得混乱  
13  
14    RecvTxt = recv_data.decode("utf-8")        #将接收浏览器的二进制数据进行解码，解码成utf-8的字符串  
15  
16    yuanzulist = RecvTxt.split(" ",maxsplit=2) #将遇到单个空格进行分割，分割2次，意思就是字符串前的空格进行分割，  
17                                #字符串后的空格进行分割，两个空格之间的字符串放到列表。  
18    xpath1 = yuanzulist[1] #取出前2个空格分割的字符串  
19    print(xpath1)  
20  
21    with open("xzz/index.html","r") as file:  #with open的好处就是打开指定文件，不需要手动close关闭文件  
22        file_data = file.read()                #读取指定index.html文件内容  
23  
24
```

你看，列表返回了两个‘ ’ / ‘ ’ 这个是存在问题的，我后面会讲

/favicon.ico 把第1个空格端的字符分割出来了。

```
['GET', '/', 'HTTP/1.1\r\nHost: 192.168.0.112:8888\r\nConnection: keep-alive\r\nCache-Control: max-age=0\r\nUpgrade-Insecure-Requests: 1\r\nUser-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/105.0.0.0 Safari/537.36\r\nAccept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/apng,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9\r\nAccept-Encoding: gzip, deflate\r\nAccept-Language: zh-CN,zh;q=0.9\r\n\r\n']  
['GET', '/favicon.ico', 'HTTP/1.1\r\nHost: 192.168.0.112:8888\r\nConnection: keep-alive\r\nUser-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/105.0.0.0 Safari/537.36\r\nAccept: image/avif,image/webp,image/apng,image/svg+xml,image/*,*;q=0.8\r\nReferer: http://192.168.0.112:8888/\r\nAccept-Encoding: gzip, deflate\r\nAccept-Language: zh-CN,zh;q=0.9\r\n\r\n']  
/favicon.ico
```

Split(“ ”, ...)分割出来的是元组。我发现元组有两个好长的列表。

```
16    yuanzulist = RecvTxt.split(" ",maxsplit=2) #将遇到单个空格进行分割，分割2次，意思就是字符串前的空格进行分割，  
17                                #字符串后的空格进行分割，两个空格之间的字符串放到列表。  
18  
19    xpath = yuanzulist[0] #第1个列表前两个空格数据  
20    print(xpath)  
21    xpath = yuanzulist[1] #第2个列表前两个空格数据  
22    print(xpath)
```

```
GET  
/  
GET  
/favicon.ico 获取了两个列表的数据。
```

下面我们根据浏览器访问时发送的字符，来判断返回给浏览器什么类型的文件或者是数据

```

while True:
    new_socket, ip_port = tcp_server.accept() #等待接收客户端的连接请求
    recv_data = new_socket.recv(4096)          #如果网页采用的GET请求，那么网页客户端发的数据最多不会超过4K
    print(recv_data)                         #接收的浏览器数据就不打印了，免得混乱

    RecvTxt = recv_data.decode("utf-8")        #将接收浏览器的二进制数据进行解码，解码成utf-8的字符串

    yuanzulist = RecvTxt.split(" ",maxsplit=10) #将遇到单个空格进行分割，分割2次，意思就是字符串前的空格进行分割，#字符串后的空格进行分割，两个空格之间的字符串放到列表.

    xpath = yuanzulist[1] #第2个列表前两个空格数据
    print(xpath)

    with open("xzz" + xpath , "rb") as file:   #with open 的好处就是打开指定文件，不需要手动close关闭文件
        #默认在当前xzz目录下，寻找xpath变量字符串指定的文件
        #比如浏览器传入的favicon.ico字符，经过解析成字符串传递给xpath变量
        #xzz + xpath变量 = xzz/favicon.ico，所以OPENxzz/favicon.ico
        #后面加入rb，是因为打开的ico文件是二级制文件，所以必须是rb形式打开
        #读取指定文件内容

        file_data = file.read()

```

```

response_line = "HTTP/1.1 200 OK\r\n"      #响应行
response_header = "Server: PWS/1.0\r\n"       #响应头
response_null = "\r\n"                      #空行
response_body = file_data                  #响应体（响应体就是发的html文件，也可以是txt，或者其它数据）

# response = response_line + response_header + response_null + response_body
# 经过响应头+行+空行+数据的组合，现在数据被封装成了http报文格式
# response = (response_line + response_header + response_null).encode("utf-8") + response_body
# 因为图片数据是二级制，所以不能把图片数据变成字符，只有将前面的报文头等数据变成二级制
#     new_socket.send(response.encode("utf-8")) #将封装的数据发出去
new_socket.send(response) #直接二级制数据发送
new_socket.close() #关闭本次为客户端创建的socket

```

```

root@ubuntu:/home/xzz/pyserver# python3 httpsockserver.py
/
Traceback (most recent call last):
  File "httpsockserver.py", line 22, in <module>
    with open("xzz" + xpath , "rb") as file:   #with open 的好处就是打开指定文件，不需要手动close关闭文件
IsADirectoryError: [Errno 21] Is a directory: 'xzz'
root@ubuntu:/home/xzz/pyserver#

```

浏览器发送数据之后，出现错误？，这是怎么回事呢？

前面 yuanzulist 变量解析出来的是有两行数据，第1行是“/”，第2行才是你想要的文件名字符，所以我们要修改代码。

```

xpath = yuanzulist[1] #第2个列表前两个空格数据
print(xpath)

if xpath == "/":
    xpath = "/index.html" #第2次收到/favicon.ico字符，发送ico图片给浏览器

    with open("xzz" + xpath , "rb") as file:   #with open 的好处就是打开指定文件，不需要手动close关闭文件
        #默认在当前xzz目录下，寻找xpath变量字符串指定的文件
        #比如浏览器传入的favicon.ico字符，经过解析成字符串传递给xpath变量
        #xzz + xpath变量 = xzz/favicon.ico，所以OPENxzz/favicon.ico
        #后面加入rb，是因为打开的ico文件是二级制文件，所以必须是rb形式打开
        #读取指定文件内容

        file_data = file.read()

    response_line = "HTTP/1.1 200 OK\r\n"      #响应行
    response_header = "Server: PWS/1.0\r\n"       #响应头
    response_null = "\r\n"                      #空行
    response_body = file_data                  #响应体（响应体就是发的html文件，也可以是txt，或者其它数据）

#     response = response_line + response_header + response_null + response_body
#     经过响应头+行+空行+数据的组合，现在数据被封装成了http报文格式
#     response = (response_line + response_header + response_null).encode("utf-8") + response_body
#     因为图片数据是二级制，所以不能把图片数据变成字符，只有将前面的报文头等数据变成二级制
#     new_socket.send(response.encode("utf-8")) #将封装的数据发出去
new_socket.send(response) #直接二级制数据发送
new_socket.close() #关闭本次为客户端创建的socket

```

welcome to xzz html 浏览器访问成功，也得到了左上角的 favicon 图标

## 使用 http 库来实现 http 服务器

http 服务器库为 `http.server`, 该 `http.server` 是 python3 内置的库。不需要另外安装

对象 = `HTTPServer((服务器 IP, 端口号), 被客户端请求后要执行的回调类)` #  
创建 HTTP 服务器

服务器 IP: 可以是'192.168.x.x'这种形式, 也可以是' '什么都不写, 就表示是本机 IP 做服务器

端口号: 服务器端口号, 类型 8080, 80, 8888 这种

对象: 就是 `HTTPServer` 函数执行后返回的对象, 我们后面的 http 服务器主要操作这个对象

被客户端请求后要执行的回调类: 填入自定义的类, 客户端 GET 本服务器之后, 服务器执行类里面的内容

### 对象. `serve_forever()` #监听端口

`serve_forever()`方法使用 `select.select()`循环监听请求, 当接收到请求后调用。当监听到请求时, 取出请求对象给回调类  
回调类中常用的函数

<code>BaseHTTPRequestHandler.path</code>	#包含的请求路径和 GET 请求的数据
<code>BaseHTTPRequestHandler.command</code>	#请求类型 GET、POST...
<code>BaseHTTPRequestHandler.request_version</code>	#请求的协议类型 HTTP/1.0、HTTP/1.1
<code>BaseHTTPRequestHandler.headers</code>	#请求的头
<code>BaseHTTPRequestHandler.responses</code>	#HTTP 错误代码及对应错误信息的字典
<code>BaseHTTPRequestHandler.handle()</code>	#用于处理某一连接对象的请求, 调用
<code>handle_one_request</code> 方法处理	
<code>BaseHTTPRequestHandler.handle_one_request()</code>	#根据请求类型调用 <code>do_XXX()</code> 方法, XXX 为请求类型
<code>BaseHTTPRequestHandler.do_XXX()</code>	#处理请求
<code>BaseHTTPRequestHandler.send_error()</code>	#发送并记录一个完整的错误回复到客户端, 内部调用 <code>send_response()</code> 方法实现
<code>BaseHTTPRequestHandler.send_response()</code>	#发送一个响应头并记录已接收的请求
<code>BaseHTTPRequestHandler.send_header()</code>	#发送一个指定的 HTTP 头到输出流。 keyword
应该指定头关键字, value 指定它的值	
<code>BaseHTTPRequestHandler.end_headers()</code>	#发送一个空白行, 标识发送 HTTP 头部结束

代码例程:

```
from http.server import HTTPServer, BaseHTTPRequestHandler
import json

data = {'result': 'this is a test'} #要发送给客户端浏览器的数据

class Resquest(BaseHTTPRequestHandler): #创建响应类
    def do_GET(self): #如果是客户端/浏览器请求的 GET, 就指向 do_GET
        self.send_response(200) #服务器发送成功码给客户端/浏览器
        self.send_header('Content-type', 'application/json') #发送一个指定的 http 头到输出流
        self.end_headers() #发送一个空白行, 表示发送 HTTP 头结束
        self.wfile.write(json.dumps(data).encode()) #发送页面数据, 就是我 data 自定义的数据

if __name__ == '__main__':
    server = HTTPServer(('', 8888), Resquest) #创建 HTTP 服务器
```

```
    print("Starting server ")
    server.serve_forever() #死循环监听 HTTP 服务器是否收到客户端
GET
```

```
root@ubuntu:/home/xzz/pyserver# python3 httpserver.py
Starting server
```

启动 HTTP 服务器

服务器 IP 为 192.168.0.112，所以填入 <http://192.168.0.112:8888/>



```
{"result": "this is a test"}
```

浏览器访问 192.168.0.112 服务器，得到数据

```
root@ubuntu:/home/xzz/pyserver# python3 httpserver.py
Starting server
192.168.0.103 - - [01/May/2021 05:23:51] "GET / HTTP/1.1" 200 -
192.168.0.103 - - [01/May/2021 05:23:51] "GET /favicon.ico HTTP/1.1" 200 -
```

服务器得到客户端/浏览器 GET 请求的地址

有时候页面无法获取 http 服务器

```
root@ubuntu:/home/xzz/pyserver# python3 httpserver.py
Starting server
```

```
fd_event_list = self._poll.poll(timeout)
KeyboardInterrupt
root@ubuntu:/home/xzz/pyserver#
```

```
root@ubuntu:/usr/share/vim# netstat -ap | grep 8888
tcp        0      0 *:8888          *:*          LISTEN      3489/python3
tcp        1      0 192.168.0.112:8888    192.168.0.103:5735  CLOSE_WAIT  -
tcp      463      0 192.168.0.112:8888    192.168.0.103:5806  CLOSE_WAIT  -
tcp        0      0 192.168.0.112:8888    192.168.0.103:5428  ESTABLISHED 3489/python3
tcp      265      0 192.168.0.112:8888    192.168.0.103:5429  ESTABLISHED  -
tcp        1      0 192.168.0.112:8888    192.168.0.103:5807  CLOSE_WAIT  -
tcp      463      0 192.168.0.112:8888    192.168.0.103:5734  CLOSE_WAIT  -
tcp      437      0 192.168.0.112:8888    192.168.0.103:5845  CLOSE_WAIT  -
```

## 在解析 JSON 格式很复杂的 HTTP 文件时遇到一些格式情况

```
from http.server import HTTPServer,BaseHTTPRequestHandler
import json

respondedata = {'id': 'a537dc8d-6756-4dd1-9dea-543f441a1d31',
                 "sn": "0A0USZ095",
                 "note": None,
                 "created_at": "2019-10-03T21:23:31.294+08:00",
                 "updated_at": "2020-12-18T21:30:25.407+08:00",
                 "uuid": "ASM9260T11412101972700115",
                 "mac": None,
                 "stage": "active",
                 "service_group_id": "6562de64-6322-48db-b3d9-412fdc394736",
                 "iccid": None,
                 "imei": None,
                 "bunks": [
                     {
                         "id": "0f071eae-57cc-47a4-9248-820aa44c502d",
                         "name": "1210_xiangzizhou",
                         "created_at": "2020-01-14T16:23:30.669+08:00",
                         "gravity_sensors": [
                             {
                                 "idx": 1,
                                 "id": "f55e9ed7-0126-4bcd-afad-c000eaff5295",
                                 "sn": "0A0USC142",
                                 "uuid": None,
                                 "can_id": 522131,
                                 "k": 5.7865,
                                 "stage": "stock",
                                 "created_at": "2019-10-14T09:09:50.265+08:00",
                                 "main_board_id": "a537dc8d-6756-4dd1-9dea-543f441a1d31"
                             },
                         ],
                         "created_at": "2019-10-03T21:23:31.294+08:00",
                         "name": "1210_xiangzizhou"
                     }
                 ],
                 "note": null,
                 "stage": "active",
                 "sn": "0A0USZ095"
             }
```

我定义很复杂的 json 格式的 http  
文件

```
jsondata = json.dumps(respondedata, indent=4)
print(jsondata)
```

为了按行输出，我用 indent 进行格式化

## 我发现我打印的内容是完整的，但是内容的排列顺序不对

```
root@ubuntu:/home/xzz/sleep_box/sleep_httpserver# python3 sleep_httpserver.py
{
    "id": "a537dc8d-6756-4dd1-9dea-543f441a1d31",
    "service_group_id": "6562de64-6322-48db-b3d9-412fdc394736",
    "uuid": "ASM9260T11412101972700115",
    "updated_at": "2020-12-18T21:30:25.407+08:00",
    "created_at": "2019-10-03T21:23:31.294+08:00",
    "imei": null,
    "mac": null,
    "iccid": null,
    "bunks": [
        {
            "id": "0f071eae-57cc-47a4-9248-820aa44c502d",
            "gravity_sensors": [
                {
                    "id": "f55e9ed7-0126-4bcd-afad-c000eaff5295",
                    "uuid": null,
                    "idx": 1,
                    "k": 5.7865,
                    "created_at": "2019-10-14T09:09:50.265+08:00",
                    "stage": "stock",
                    "can_id": 522131,
                    "main_board_id": "a537dc8d-6756-4dd1-9dea-543f441a1d31",
                    "sn": "0A0USC142"
                },
            ],
            "created_at": "2019-10-03T21:23:31.294+08:00",
            "name": "1210_xiangzizhou"
        }
    ],
    "note": null,
    "stage": "active",
    "sn": "0A0USZ095"
}
```

我发现  
" sn"，" stage"，" note" 没有出现在前面几行，  
但是我写的 JSON 文件里  
面，明明  
将" sn"，" stage"，" n

" sn"，" stage"，" note" 被格式化在  
最后几行了，当然这不影响 json 解析，但  
是始终感觉 jumps 格式化之后的顺序不合  
理

我们使用 http 协议传输给浏览器看看现象

```
jsondata = json.dumps(responsedata, indent=4)
print(jsondata)

class Resquest(BaseHTTPRequestHandler):
    def do_GET(self):
        self.send_response(200)
        self.send_header('Content-type', 'application/json')
        self.end_headers()
        self.wfile.write(json.dumps(jsondata).encode())

if __name__ == '__main__':
    server = HTTPServer(('', 8888), Resquest)
    print("Starting server")
    server.serve_forever()
```

你看，浏览器出现了\n换行符，和\"转义字符，我必须取消掉这两个字符

```
jsondata = json.dumps(responsedata, indent=4).replace('\n', '')
print(jsondata)
```

取消，替换\n的符号就是，。也可以不使用 indent=4，这样就不用按照行格式化输出，\n就自然没有了。

```
jsondata = json.dumps(responsedata)
print(jsondata)
```

```
root@ubuntu:/home/xzz/sleep_box/sleep httpserver# python3 sleep httpserver.py
{"id": "a537dc8d-6756-4d19-9da4-543f441a1d31", "mac": null, "sn": "0A0USZ095", "imei": null, "stage": "active", "iccid": null, "note": null, "service_group_id": "6562de46-6322-48db-b3d9-412fdc394736", "updated_at": "2020-12-18T21:30:25.407+08:00", "bunks": [{"gravity_sensors": [{"k": 5.7865, "id": 1, "can_id": 522131, "id": "f55e9ed7-0126-4bcd-afad-c000eaaff5295", "stage": "stock", "sn": "0A0USC142", "uuid": null, "created_at": "2019-10-14T09:09:50.265+08:00", "main_board_id": "a537dc8d-6756-4d19-9da4-543f441a1d31"}, {"k": 5.7865, "id": 2, "can_id": 522131, "id": "f55e9ed7-0126-4bcd-afad-c000eaaff5295", "stage": "stock", "sn": "0A0USC142", "uuid": null, "created_at": "2019-10-14T09:09:50.265+08:00", "main_board_id": "a537dc8d-6756-4d19-9da4-543f441a1d31"}, {"k": 5.7865, "id": 3, "can_id": 522131, "id": "f55e9ed7-0126-4bcd-afad-c000eaaff5295", "stage": "stock", "sn": "0A0USC142", "uuid": null, "created_at": "2019-10-14T09:09:50.265+08:00", "main_board_id": "a537dc8d-6756-4d19-9da4-543f441a1d31"}, {"k": 5.7865, "id": 4, "can_id": 522131, "id": "f55e9ed7-0126-4bcd-afad-c000eaaff5295", "stage": "stock", "sn": "0A0USC142", "uuid": null, "created_at": "2019-10-14T09:09:50.265+08:00", "main_board_id": "a537dc8d-6756-4d19-9da4-543f441a1d31"}]}, "print 打印结果，看起来是有点不舒服
```

你毛) 就没有了) 日利于\)"岐以宣管

```
    jsondata = json.dumps(responsedata).replace("\\"", "")  
    print(jsondata)
```

```
直接用 -l 替换 \"转义字符或 OK 了
```

直接用 replace 替换所有以全文就 OK 了

```
[{"uid": "A5B9A8F7B111D201757200115", "service_group_id": "6552ed4-632c-4640-9269-41EFD394739", "sn": "040125399", "node": "null", "impl": "null", "created_at": "2019-10-14T07:13:31.294+08:00", "updated_at": "2019-12-18T02:13:25.407+08:00", "stage": "active", "version": "1.0", "status": "normal", "parent": "11", "parent_id": "6552ed4-632c-4640-9269-41EFD394739", "granularity": "by_node", [{"created_at": "2019-10-14T07:13:31.294+08:00", "updated_at": "2019-10-14T07:13:31.294+08:00", "v": 1, "t": "M_7865", "sn": "04051C42", "can_id": "521211", "uid": "null", "impl": "null"}, {"uid": "555e97-116c-40cd-afad-0000e800f259", "stage": "stock", "stock_idx": 1, "main_board_id": "a537de-6376-40d1-90e8-5434f41a1d31", {"created_at": "2019-10-14T09:09:50.265+08:00", "updated_at": "2019-10-14T09:09:50.265+08:00", "v": 1, "t": "M_7865", "sn": "04051C42", "can_id": "521211", "uid": "null", "impl": "null"}, {"uid": "555e97-116c-40cd-afad-0000e800f259", "stage": "stock", "stock_idx": 2, "main_board_id": "a537de-6376-40d1-90e8-5434f41a1d31", {"created_at": "2019-10-14T09:09:50.265+08:00", "updated_at": "2019-10-14T09:09:50.265+08:00", "v": 1, "t": "M_7865", "sn": "04051C42", "can_id": "521211", "uid": "null", "impl": "null"}, {"uid": "555e97-116c-40cd-afad-0000e800f259", "stage": "stock", "stock_idx": 3, "main_board_id": "a537de-6376-40d1-90e8-5434f41a1d31", {"created_at": "2019-10-14T09:09:50.265+08:00", "updated_at": "2019-10-14T09:09:50.265+08:00", "v": 1, "t": "M_7865", "sn": "04051C42", "can_id": "521211", "uid": "null", "impl": "null"}]
```

你看\"也没有了，越来越像 JSON 了。但是怎么浏览器还是解析不出 JSON 格式呢？

```
{uuid: ASM9260T11412101972700115, service_group_id: 6562de6  
null, created_at: 2019-10-03T21:23:31.294+08:00, updated_a  
756-4dd1-9dea-543f441a1d31, iccid: null, mac: null, bunks:
```

因为数据之间有空格

```
jsondata = (json.dumps(responsedata).replace("\\"",'')).replace(" ", '')  
print(jsondata)
```

我在 replace 取消\"转义字符之后又加入了 replace 取消掉空格

```
"{id:a537dc8d-6756-4dd1-9dea-543f441a1d31,sn:0A0USZ095,note:null,imei:null,-412fdc394736,iccid:null,bunks:[{id:0f071eae-57cc-47a4-9248-820aa44c502,stage:stock,main_board_id:a537dc8d-6756-4dd1-9dea-543f441a1d31},{idx:1:id:a537dc8d-6756-4dd1-9dea-543f441a1d31},{idx:2:id:a537dc8d-6756-4dd1-9dea-543f441a1d31}],id:f55e9ed7-0126-4bcd-af2
```

数据更紧凑了，但是怎么还是没有像 JSON 那样显示。

```
"{id:a537dc8d-6756-4dd1-9dea-543f441a1d31,sn:0A0USZ095,note:null,imei:null,-412fdc394736,iccid:null,bunks:[{id:0f071eae-57cc-47a4-9248-820aa44c502,stage:stock,main_board_id:a537dc8d-6756-4dd1-9dea-543f441a1d31},{idx:1:id:a537dc8d-6756-4dd1-9dea-543f441a1d31},{idx:2:id:a537dc8d-6756-4dd1-9dea-543f441a1d31}],id:f55e9ed7-0126-4bcd-af2
```

我发现数据最外层大括号前后有冒号 ”

```
09:09:50.265+08:00}],id:0f071eae-57cc-47a4-924  
_id:6562de64-6322-48db-b3d9-412fdc394736}"|
```

处理了以上方式之后，还需要注意一个低级错误，方法如下：

```
jsondata = (json.dumps(responsedata).replace("\\"",'')).replace(" ", '')  
print(jsondata)
```

```
class Request(BaseHTTPRequestHandler):  
    def do_GET(self):  
        self.send_response(200)  
        self.send_header('Content-type', 'application/json')  
        self.end_headers()  
        self.wfile.write(json.dumps(jsondata).encode())
```

这儿我已经处理过  
JSON 数据了

我在发送的时候怎么  
还要处理 JSON 数据，  
明显是错误的

实际修改如下

```
jsondata = json.dumps(responsedata).replace(" ", '')  
print(jsondata)  
  
class Request(BaseHTTPRequestHandler):  
    def do_GET(self):  
        self.send_response(200)  
        self.send_header('Content-type', 'application/json')  
        self.end_headers()  
        self.wfile.write(jsondata.encode())
```

直接发送处理好的  
JSON 变量

```
[{"bunks": [  
    {"gravity_sensors": [  
        {"created_at": "2019-10-14T09:09:50.265+08:00",  
        "main_board_id": "a537dc8d-6756-4dd1-9dea-543f441a1d31",  
        "uuid": null,  
        "stage": "stock",  
        "sn": "0A0USC142",  
        "idx": 1,  
        "kt": 5.7865,  
        "cam_id": 521231,  
        "id": "f55e9ed7-0126-4bcd-afad-c000eaff5295"  
    },  
    {"created_at": "2019-10-14T09:09:50.265+08:00",  
    "main_board_id": "a537dc8d-6756-4dd1-9dea-543f441a1d31",  
    "uuid": null,  
    "stage": "stock",  
    "sn": "0A0USC142",  
    "idx": 2,  
    "kt": 5.7865,  
    "cam_id": 521231,  
    "id": "f55e9ed7-0126-4bcd-afad-c000eaff5295"}],  
    "note": null,  
    "service_group_id": "6562de64-6322-48db-b3d9-412fdc394736",  
    "imei": null,  
    "iccid": null,  
    "id": "a537dc8d-6756-4dd1-9dea-543f441a1d31"}]
```

你看，JSON 格式的数据发送成功。

## python 实现 HTTP 文件服务器

1. 下载 miniserver 程序 地址

sudo curl -L

[https://github.com/svenstaro/miniserve/releases/download/v0.4.1/miniserve-linux-x86\\_64](https://github.com/svenstaro/miniserve/releases/download/v0.4.1/miniserve-linux-x86_64) -o /usr/local/bin/miniserve

```
root@ubuntu:/home/xzz/pyserver/miniserver# sudo curl -L https://github.com/svenstaro/miniserve
iserve-linux-x86_64 -o /usr/local/bin/miniserve
  % Total    % Received % Xferd  Average Speed   Time   Time     Time  Current
               Dload  Upload   Total   Spent   Left  Speed
  0      0      0      0      0       0      0 --:--:-- --:--:-- --:--:--   0
100 1379k  100 1379k      0       0  24966      0  0:00:56  0:00:56 --:--:-- 29635
```

这是下载过程

2. 加权限 sudo chmod +x /usr/local/bin/miniserve

3. 查询是否安装成功 **miniserve --help**

```
root@ubuntu:/home/xzz/pyserver/miniserver# miniserve --help
miniserve 0.4.1
Sven-Hendrik Haase <svenstaro@gmail.com>, Boastful Squirrel <boas...
For when you really just want to serve some files over HTTP right...

USAGE:
  miniserve [FLAGS] [OPTIONS] [--] [PATH]

FLAGS:
  -u, --upload-files      Enable file uploading
```

安装成功

miniserver 使用方式

**miniserver** 创建个下载目录

```
root@ubuntu:/home/xzz/pyserver/miniserver# ls
222dir  test111.txt
```

在目录里面创建需要下载的文件

执行 **miniserve ./miniserver/**

这个 ./miniserver 就是在当前目录下，加载要下载的目录。要下载的目录就是 miniserver

```
root@ubuntu:/home/xzz/pyserver# miniserve ./miniserver/
miniserve v0.4.1
Serving path /home/xzz/pyserver/miniserver at http://[:]:8080, http://localhost:8080
Quit by pressing CTRL-C
```

执行之后，http://localhost:8080 就是你外面浏览器需要输入的地址。但是注意 localhost 要改成你当前运行 miniserver 服务器的 IP 地址。

外部浏览器运行测试



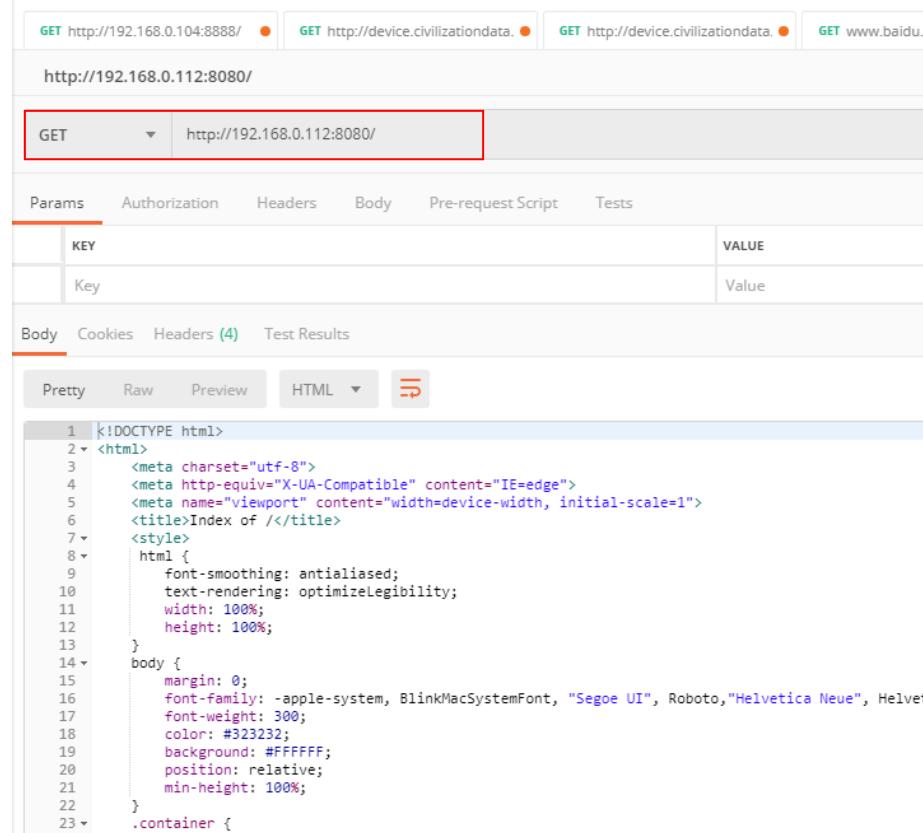
Index of /

[Download .tar.gz](#)

▲ Name
222dir/
test111.txt

你看输入对应的 ip 和端口文件就出来了，可以右键选择下载。注意可以直接下载 txt 文件，但是直接下载目录不行，因为目录是 ubuntu 格式的目录，在 windows 打不开。

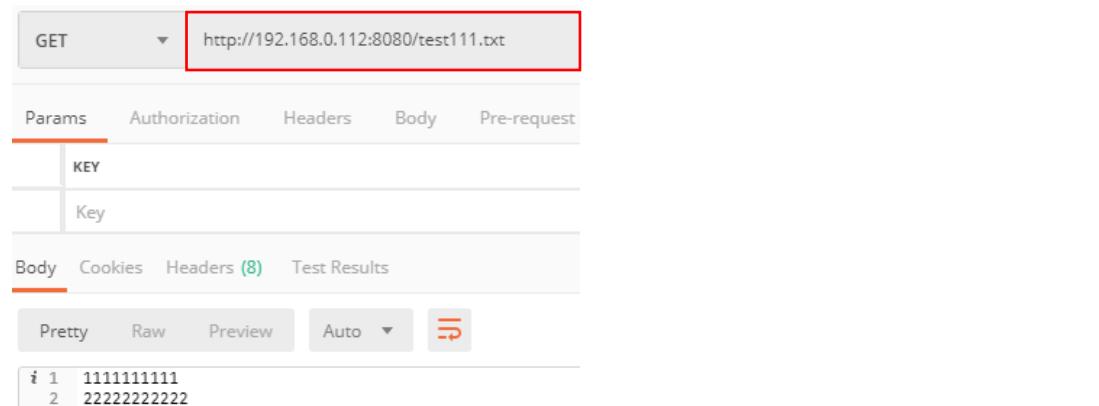
用 HTTP 客户端测试软件 Postman 测试下文件服务器下载功能。



The screenshot shows the Postman interface with a red box highlighting the URL input field. The URL is set to `http://192.168.0.112:8080/`. The response body is displayed as a pretty-printed HTML document:

```
1 <!DOCTYPE html>
2 <html>
3   <meta charset="utf-8">
4   <meta http-equiv="X-UA-Compatible" content="IE=edge">
5   <meta name="viewport" content="width=device-width, initial-scale=1">
6   <title>Index of /</title>
7   <style>
8     html {
9       font-smoothing: antialiased;
10      text-rendering: optimizeLegibility;
11      width: 100%;
12      height: 100%;
13   }
14   body {
15     margin: 0;
16     font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto, "Helvetica Neue", Helvetica
17     font-weight: 300;
18     color: #323232;
19     background: #FFFFFF;
20     position: relative;
21     min-height: 100%;
22   }
23   .container {
```

我直接用 GET 方法，把页面的 html 数据全部抓取下来了。这不是我想要的，我想要的是我要下载某个 txt 文件的真实数据。



The screenshot shows the Postman interface with a red box highlighting the URL input field. The URL is set to `http://192.168.0.112:8080/test111.txt`. The response body is a plain text file containing the following content:

```
1 1111111111
2 222222222222
3 333333333333
4 |
```

直接输入 txt 文件具体路径，就能直接下载 txt 文件内容。

使用 python 的 http 协议来下载 http 文件服务器里面的文件内容

先使用 python3 内置的 urllib3 http 网络库实现 http 客户端

```
import urllib3
import requests
requests.packages.urllib3.disable_warnings() # 忽略警告：InsecureRequestWarning
http = urllib3.PoolManager() ## 一个PoolManager实例来生成请求，由该实例对象处理与线程池的连接以及线程安全的所有细节
r = http.request('GET', 'http://192.168.0.112:8080/test111.txt')# 通过request()方法创建一个请求：
print("Response = %d" %r.status) # 200
print("#"*30)
print(r.data.decode())
```

这是在 ubuntu 服务器上面模拟的 http 客户端

```
root@ubuntu:/home/xzz/pyserver# python3 httpclient.py
Response = 200
#####
1111111111
2222222222
3333333333
```

简短的文件数据可以这样请求下来。

http 响应头查看和数据本身长度查看

```
import urllib3
import requests
requests.packages.urllib3.disable_warnings() # 忽略警告 : InsecureRequestWarning

http = urllib3.PoolManager() ## 一个PoolManager实例来生成请求,
r = http.request('GET', 'http://192.168.0.112:8080/test111.txt')
print("Response = %d" %r.status) # 200
print("Headers = %s" %r.headers) #查看http响应头内容
print("#"*30)
print(r.data.decode())#真实数据查看
```

修改 txt 文本数据位 31 个字符，但是不知道怎么计算出来是 32C，也就是 32 个字符，应该是多了个结束符。

```
root@ubuntu:/home/xzz/pyserver# python3 httpclient.py
Response = 200
Headers = HTTPEndPointDict([('content-length': '32', 'accept-ranges': 'bytes', 'etag': '"1141ae6:20:637f0cbd:1024e8a0"', 'last-modified': 'Thu, 24 Nov 2022 06:18:37 GMT', 'content-disposition': 'inline; filename="test111.txt"', 'date': 'Thu, 24 Nov 2022 06:22:05 GMT', 'content-type': 'text/plain'})
#####
1111111111222222223333333333
```

执行请求之后，服务器返回的响应头包含了，`content-length: 32` 数据长度，`accept-ranges:bytes` 数据类型。`filename = ...`读取的文件名。

如果是下载大文件，如何实现 http 请求，数据分段获取？

在请求头中加入 { ‘Range’ :’ bytes=0-10’ } 这样就可以获取文本的前 0 到 10 个字节数据。

```
import urllib3
import requests
requests.packages.urllib3.disable_warnings() # 忽略警告 : InsecureRequestWarning

header = {'Range': 'bytes=0-10'}

http = urllib3.PoolManager() ## 一个PoolManager实例来生成请求, 由该实例对象处理与线程池的连接以及线程安全的所有细节
r = http.request('GET', 'http://192.168.0.112:8080/test111.txt', headers=header, fields={})# 通过request()方法创建一个请求:
print("Response = %d" %r.status) # 200
print("Headers = %s" %r.headers) #查看http响应头内容
print("#"*30)
print(r.data.decode())#真实数据查看
```

```

root@ubuntu:/home/xzz/pyserver# python3 httpclient.py
Response = 206
Headers = HTTPHeaderDict({'etag': '"1141ae6:20:637f0cbd:1024e8a0"', 'accept-ranges': 'bytes', 'content-length': '11', 'content-type': 'text/plain', 'last-modified': 'Thu, 24 Nov 2022 06:18:37 GMT', 'content-disposition': 'inline; filename="test111.txt"', 'date': 'Thu, 24 Nov 2022 06:41:36 GMT', 'content-range': 'bytes 0-10/32'})
#####
11111111112

```

现在发现个问题，因为 0 也算 1 个字节，所以得到了 11 个字节的数据，而且 content-length 返回的是这次发送的数据长度，而不是整个文件的长度。

HEAD 请求使用，HEAD 请求服务器只返回响应头，不会返回数据本身。

```

import urllib3
import requests
requests.packages.urllib3.disable_warnings() # 忽略警告 : InsecureRequestWarning

http = urllib3.PoolManager() ## 一个PoolManager实例来生成请求,
r = http.request('HEAD', 'http://192.168.0.112:8080/test111.txt')
print("Response = %d" %r.status) # 200
print("Headers = %s" %r.headers) #查看http响应头内容
print("#"*30)
print(r.data.decode())#真实数据查看

```

```

root@Ubuntu:/home/xzz/pyserver# python3 httpclient.py
Response = 200
Headers = HTTPHeaderDict({'last-modified': 'Thu, 24 Nov 2022 06:18:37 GMT', 'content-disposition': 'inline; filename="test111.txt"', 'date': 'Thu, 24 Nov 2022 06:54:03 GMT', 'content-type': 'text/plain', 'etag': '"1141ae6:20:637f0cbd:1024e8a0"', 'content-length': '32', 'accept-ranges': 'bytes'})
#####

```

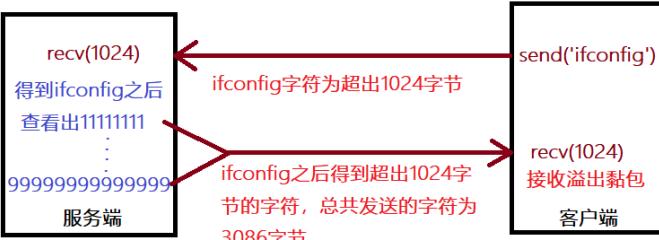
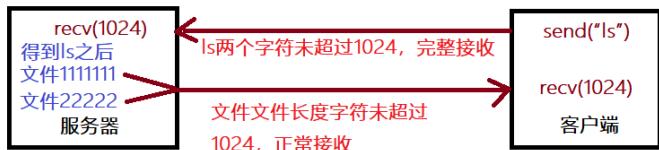
你看，HEAD 请求只返回头数据，不会返回 txt 文本实际内容。也就是不返回数据本身。

这样就可以读取 content-length:32 文件总长度 32 字节。

## FTP 文件传输实现

### TCP 黏包问题解决(重点)





## TCP 黏包问题测试

服务端代码

```
import socket

server = socket.socket(socket.AF_INET,socket.SOCK_STREAM)

server.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1) #socket.socket端口占用问题解决方式

server.bind(('',8888))

server.listen(5)

conn,addr = server.accept()

res1 = conn.recv(1024) #阻塞接收客户端第1次 send的数据
print('1=',res1)
res2 = conn.recv(1024) #阻塞接收客户端第2次 send的数据
print('2=',res2)
```

客户端代码

```
import socket
import time

client = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
client.connect(('192.168.0.112',8888))
client.send('hello'.encode('utf-8'))
time.sleep(5) #延时5秒发送第2帧数据
client.send('world'.encode('utf-8'))
```

```
root@ubuntu:/home/xzz/pyserver/sshserver# python3 sshserver.py
1= b'hello'
2= b'world'
```

你看服务器两个 recv 是接收两次 send 的数据，而不是一个 recv 同时接收两次 send。

所以 recv 这个函数是和客户端 send 一一对应的，发一次 send，不管数据多少字节，就消耗一次 recv。超出 recv 接收长度，剩余的字节就在网络中残留，等待下一次 send 发送，残留数据和新的 send 数据一起发送到 recv 造成黏包。

```

import socket
server = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
server.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1) #socket.socket端口占用问题解决方式
server.bind(('',8888))
server.listen(5)
conn,addr = server.accept()
res1 = conn.recv(2) #阻塞接收客户端第1次 send的数据
print('1=',res1)
res2 = conn.recv(1024) #阻塞接收客户端第2次 send的数据
print('2=',res2)

```

我们第一次 recv 只接受 2 字节数据

```

import socket
import time

client = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
client.connect(('192.168.0.112',8888))
client.send('hello'.encode('utf-8'))
time.sleep(1)#DELAY 1
client.send('world'.encode('utf-8'))

```

我客户端延时改成 1 秒

```

root@ubuntu:/home/xzz/pyserver/sshserver# python3 sshserver.py
1= b'he'
2= b'llo'

```

测试结果，我们发现第 1 个 send 的数据 hello，因为服务端 recv 只接受 2 字节，导致第一次 recv 只能接受 'he'，第 2 次 recv 才能接收残留在网络的 'llo'。但是因为客户端延时发送第 2 帧 send 的问题，导致服务端 2 次 recv 在 1 秒中之内执行完。接收不到第 2 个 send 数据。

```

import socket
import time

client = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
client.connect(('192.168.0.112',8888))
client.send('hello'.encode('utf-8'))
#time.sleep(1)#DELAY 1
client.send('world'.encode('utf-8'))

```

客户端端取消掉延时发送。

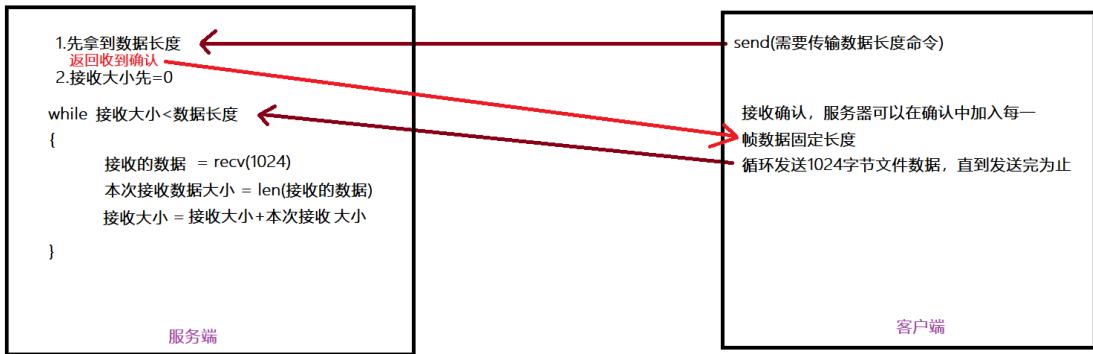
```

root@ubuntu:/home/xzz/py
1= b'he'
2= b'lloworld'

```

服务端接收了两次 send 数据，但是第一次 recv 没接收完的数据放在了第 2 次 recv 接收。这个接收残留数据的顺序和我前面框图描述的一致，残留的数据在第 2 次 recv 接收的时候放在前面，黏包形成。

## 解决黏包问题方法



## 服务端程序

```
import socket  
import struct  
  
server = socket.socket(socket.AF_INET,socket.SOCK_STREAM)  
server.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1) #socket.socket端口占用问题解决方式  
server.bind(('',8888))  
server.listen(5)  
  
conn,addr = server.accept()  
  
head = conn.recv(4) #先收struct封包的4个字节报头  
total = struct.unpack('i',head)[0] #把报头数据取出来1024  
  
print('head = ', total)  
  
recv_size = 0  
recvdata = b'';  
while recv_size < total:  
    recv1 = conn.recv(256);  
    recvdata = recvdata+recv1  
    recv_size = recv_size + len(recv1)  
  
print(recvdata)  
print(recv_size)  
  
#1.制作固定长度报头  
#res = struct.pack('i',1024) #i就是转成32位整形, 1024就是整形数字转换成0x00000400, 但是排列方式是大端模式00040000, 0\x04\x00\x00 .另外一端使用value = struct.unpack('i', res) 解析, 得到元组(1024, ), 我直接取value[0]得到1024
```

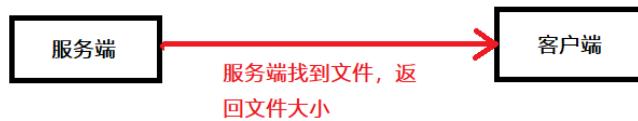
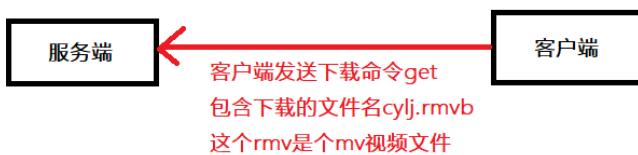
## 客户端程序

```
import socket  
import time  
import struct  
  
client = socket.socket(socket.AF_INET,socket.SOCK_STREAM)  
client.connect(('192.168.0.112',8888))  
  
array = [] #write byte data package . byte data in(0~256)  
for i in range(255):  
    array.append(i)  
  
array.append(0xff) #add [256] = 0xff  
print(array)  
  
arraysend =bytearray(array) #list to byte send  
print(arraysend)  
  
res = struct.pack('i',1024)  
client.send(res)  
client.send(arraysend)
```

客户端发送的数据包一次是 1024 字节

服务器虽然每次 `recv` 是 256 字节，最后还是接受到了 4 个 ff。证明黏包问题解决了。

## TCP 大文件下载实验



这种模式，服务端并不知道客户端接收了多少，反正把视频文件数据发完就是

客户端反正接收就是了，如果中间出现断网，客户端这时候没有接收完，就死在循环里面了  
下一次重连客户端也不会受到新的数据

这种模式在网比较好的情况下可以这样做，网络不好的环境，要考虑改进

服务端程序

```
import socket
import struct
import json
import os

server = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
server.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1) #socket.socket端口占用问题解决方式
server.bind(('',8888))
server.listen(5)

conn,addr = server.accept()

#1.服务端先收到客户端数据
head = conn.recv(1024) #服务端先收到客户端的get请求

#2.解析命令，提取参数

cmds = json.loads(head.decode('utf-8')) #解析字典
```

客户端发送下载命令get  
包含下载的文件名cyjl.rmvb  
这个rmvb是个mv视频文件

```

print(cmds)
filename=cmds['file'] #提取 'mv.txt'

print("file = ",filename)

#3.返回给客户端的，找到的文件名，和文件大小头
header_msg = {
    'filename' : filename,
    'file_size' : os.path.getsize(filename) #os..getsize 获得指定路径下文件的大小，我直接获取当前路径下文件大小
}

header_json = json.dumps(header_msg) #因为字典无法进行TCP字节流传输，我将字典转成json字符串

header_bytes = header_json.encode('utf-8') #json字符串转码成utf-8的字节流

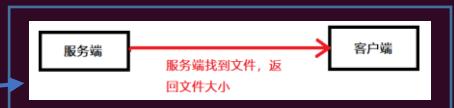
conn.send(struct.pack('i',len(header_bytes)))#先给客户端回报头长度
conn.send(header_bytes) #然后再回报头数据

#3.读取客户端要求下载的文件

with open(filename,'rb') as f: #注意，先读取的这个文件是我在虚拟机读取的，没有和windows客户端处于同一个系统
#如果服务端和客户端处于同一个系统，这时候服务端去读同名文件，客户端也去写同名文件，系统崩溃
    #conn.send(f.read()) #不能将整个文件读完再发送，如果文件大小是几个T，直接读完导致内存崩溃
    for line in f:
        conn.send(line)

conn.close()

```



## 客户端程序

```

import socket
import struct
import json

client = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
client.connect(('192.168.0.112',8888))

send_str = {
    'cmd' : 'get',
    'file': 'cylj.rmvb' #Download file name
}

send_json = json.dumps(send_str).encode('utf-8') #send Download cmd
client.send(send_json)
#1.recv head length
obj = client.recv(4)
header_size = struct.unpack('i',obj)[0] #get to head byte length

#. recv head
header_bytes = client.recv(header_size)
#3.parser head data
header_json = header_bytes.decode('utf-8') # decode is bin transition string
head_dic = json.loads(header_json) # string transition json format
print(head_dic)
total_size = head_dic['file_size'] #get Download file length
#4.recv data
filename = head_dic['filename']#get recv file name

with open(filename,'wb') as f:
    recv_size = 0
    while recv_size < total_size:
        data = client.recv(1024)
        f.write(data)
        recv_size+=len(data)
        #select add progress bar

    f.close()
print("file Download Over..")

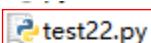
```

```

cylj.rmvb sshserver.py
root@ubuntu:/home/xzz/pyserver/sshserver# python3 sshserver.py
{'cmd': 'get', 'file': 'cylj.rmvb'}
file = cylj.rmvb

```

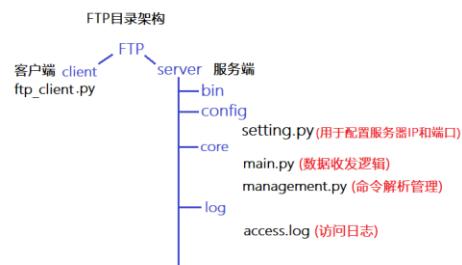
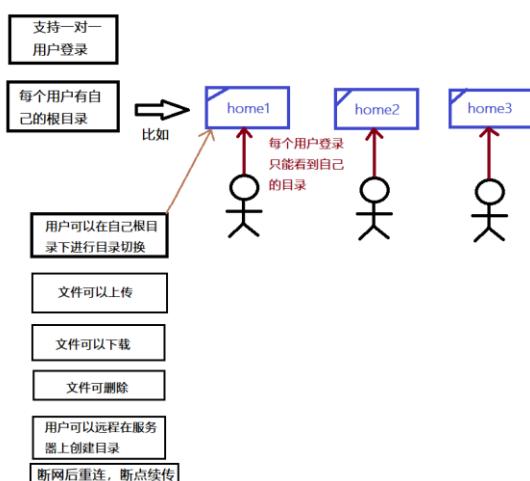
服务器成功运行下载视频文件



客户端成功收到视频文件，播放正常

## FTP 服务端/客户端实现逻辑

FTP软件开发需求



`os.path.dirname(path) #去掉文件名, 返回目录名。 (要 import os)`

`path:` 填入文件绝对路径

例如:

```
print(os.path.dirname("E:/Read_File/read_yaml.py"))
#结果:
E:/Read_File
```

```
print(os.path.dirname("E:/Read_File"))
#结果:
E:/
```

`os.path.abspath(__file__)` #`__file__`是获取当前执行 file 命令的程序文件绝对路径

例如:

```
import os,sys
print(os.path.abspath(__file__))
```

```
root@ubuntu:/home/xzz/pyserver/ftp/bin# python3 ftp_server.py
/home/xzz/pyserver/ftp/bin/ftp_server.py
```

得到当前执行`__file__`的 py 文件所在系统的绝对路径

```
import os,sys
print(os.path.dirname(os.path.abspath(__file__)))
```

增加 1 级 `os.path.dirname`, 得到当前 py 文件的上级目录绝对路径

```
root@ubuntu:/home/xzz/pyserver/ftp/bin# python3 ftp_server.py
/home/xzz/pyserver/ftp/bin
```

你看 py 文件上级目录是 bin

如果我想获取上上级目录呢?, 多包含一层 `os.path.dirname` 不就行了。

```
import os,sys  
print(os.path.dirname(os.path.dirname(os.path.abspath(__file__))))  
root@ubuntu:/home/xzz/pyserver/ftp/bin# python3 ftp_server.py  
/home/xzz/pyserver/ftp
```

你看，上上及目录。反正依次类推。

sys.path #输出当前系统的环境变量

```
import os,sys  
print(sys.path)  
/home/xzz/pyserver/ftp  
['/home/xzz/pyserver/ftp/bin', '/usr/lib/python35.zip', '/usr/lib/python3.5', '/usr/lib/python3.5/plat-x86_64-  
/usr/lib/python3.5/lib-dynload', '/usr/local/lib/python3.5/dist-packages', '/usr/lib/python3/dist-packages']
```

以列表的方式输出当前系统的环境变量有哪些。

sys.path.append(path) #加入新的路径到系统环境变量中

path: 写入路径

```
import os,sys  
  
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__))) #得到 ftp根目录路径  
print(BASE_DIR)  
  
sys.path.append(BASE_DIR) #加入新的路径到系统环境变量中  
  
print(sys.path)
```

```
/home/xzz/pyserver/ftp  
['/home/xzz/pyserver/ftp/bin', '/usr/lib/python35.zip', '/usr/lib/python3.5', '/usr/lib/python3.5/plat-x86_64-  
/usr/lib/python3.5/lib-dynload', '/usr/local/lib/python3.5/dist-packages', '/usr/lib/python3/dist-packages']
```

新的路径进来了

## 服务端程序初步框架

ftp\_server.py

```
import os,sys  
  
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__))) #得到 ftp根目录路径  
print(BASE_DIR)  
  
sys.path.append(BASE_DIR) #加入新的路径到系统环境变量中  
  
print(sys.path)  
  
if __name__ == "__main__":  
    from core import management #程序初始化运行的时候，导入管理模块，执行管理模块里面的初始类函数  
    argv_parser = management.ManagementTool(sys.argv[1]) #创建management.py文件中ManagementTool类的对象  
    print('out = ', sys.argv) #得到运行程序的py文件名，和命令行输入的参数  
    argv_parser.execute()
```

提取命令  
↓  
执行服务端程序ftp\_server.py start

management.py

```

import os,sys

class ManagementTool(object):
    """负责对用户输入的指令进行解析，并调用响应的模块函数进行处理"""
    def __init__(self,sys_argv):
        print("==INIT==",sys.argv)
        self.sys_argv = sys.argv
        self.verify_argv()

    def verify_argv(self):
        #验证指令是否合法
        if len(self.sys_argv) < 2: #输入参数的列表必须>1，意思就是最少都要有一个文件名
            self.help_msg()
        cmd = self.sys_argv
        if cmd in ['start','stop']: #如果得到的参数不是start或stop，输出错误
            self.help_msg()
        else:
            print("invalid")

    def help_msg(self):
        msg = '''
        start FTP
        stop FTP
        restart FTP
        creat username
        ...
        print(msg)
    def execute(self):
        #解析并执行指令
        print("exec")

```

root@ubuntu:/home/xzz/pyserver/ftp/bin# python3 ftp\_server.py sss  
/home/xzz/pyserver/ftp  
['/home/xzz/pyserver/ftp/bin', '/usr/lib/python35.zip', '/usr/lib/python3.5', '/usr/lib/python3.5/plat-x86\_64-linux-gnu', '/usr/lib/python3.5/lib-dynload', '/usr/local/lib/python3.5/dist-packages', '/usr/lib/python3/dist-packages', '/home/xzz/pyserver/ftp']  
==INIT== sss  
invalid  
out = ['ftp\_server.py', 'sss']  
exec

传入 sss，命令参数错误

执行服务端程序ftp\_server.py start  
↑  
执行的时候解析该命令  
start 启动服务器，  
stop停止服务器

```

root@ubuntu:/home/xzz/pyserver/ftp/bin# python3 ftp_server.py stop
/home/xzz/pyserver/ftp
['/home/xzz/pyserver/ftp/bin', '/usr/lib/python35.zip', '/usr/lib/python3.5', '/usr/lib/python3.5/plat-x86_64-linux-gnu', '/usr/lib/python3.5/lib-dynload', '/usr/local/lib/python3.5/dist-packages', '/usr/lib/python3/dist-packages', '/home/xzz/pyserver/ftp']  

==INIT== stop  

start FTP
stop FTP
restart FTP
creat username

out = ['ftp_server.py', 'stop']
exec
```

传入 stop 命令参数正确

**os.path.join() #函数用于路径拼接字符，形成新的路径**

- 从后往前看，会从第一个以"/"开头的参数开始拼接，之前的参数全部丢弃；

```
import os
```

```
print("1:",os.path.join('aaaa','/bbbb','cccc.txt'))
```

1: /bbbb\cccc.txt

```
print("2:",os.path.join('/aaaa','/bbbb','/cccc.txt'))
```

2: /cccc.txt

以上一种情况为先。在上一种情况确保情况下，若出现"./"开头的参数，会从"./"开头的参数的前面参数全部保留

```
print("3:",os.path.join('aaaa','ddd','./bbb','cccc.txt'))
```

3: aaaa\ddd\./bbb\cccc.txt

## 例子

```
import os  
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__))) #得到 ftp根目录路径  
BASE_DIR = os.path.join(BASE_DIR, 'home')  
print(BASE_DIR)
```

/home/xzz/pyserver/ftp 没加入 join 的原始路径

/home/xzz/pyserver/ftp/home 加入 join 的新路径

## ftp\_server.py 改进 1

```
import os,sys  
  
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__))) #得到 ftp根目录路径  
print(BASE_DIR)  
  
sys.path.append(BASE_DIR) #加入新的路径到系统环境变量中  
  
print(sys.path)  
  
if __name__ == "__main__":  
    from core import management #程序初始化运行的时候，导入 core目录下管理模块.py，执行管理模块里面的初始类函数  
  
    argv_parser = management.ManagementTool(sys.argv[1]) #创建management.py文件中ManagementTool类的对象  
    print('out ', sys.argv) #得到运行程序的py文件名，和命令行输入的参数  
    argv_parser.execute() #启动socket 接收事件循环
```

## management.py 改进 1

```
import os,sys  
from core import main  
  
class ManagementTool(object):  
    """负责对用户输入的指令进行解析，并调用响应的模块函数进行处理"""  
    def __init__(self,sys_argv):  
        print("==INIT==",sys_argv)  
        self.sys_argv = sys_argv  
        self.verify_argv()  
  
    def verify_argv(self):  
        #验证指令是否合法  
        if len(self.sys_argv) < 2: #输入参数的列表必须>1，意思就是最少都要有一个文件名  
            self.help_msg()  
        cmd = self.sys_argv  
        if cmd in ['start','stop']: #如果得到的参数不是 start或stop，输出错误  
  
            self.help_msg()  
        else:  
            print("invalid")  
  
    def help_msg(self):  
        msg = ''  
        start FTP  
        stop FTP  
        restart FTP  
        creat username  
        ...  
        print(msg)
```

```

def execute(self):
    #解析并执行指令
    cmd = self.sys_argv

    if cmd == 'start': #如果传入字符是start
        self.start() #执行类内部的函数用self指定
    else:
        pass

def start(self):
    server = main.FTPServer(self)
    server.run_forever()

def creteuser(self):
    pass

```

### main.py

```

import socket
from config import settings #导入config目录下的settings.py

class FTPServer(object):
    '''处理与客户端所有的交互，socket'''
    def __init__(self,management_input):
        self.management_input = management_input #得到class ManagementTool类里面的self.sysargv这些self定义的参数
        self.sock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        self.sock.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1) #socket.socket端口占用问题解决方式
        self.sock.bind((settings.HOST,settings.PORT)) #绑定IP和端口，调用setting.py的变量
        self.sock.listen(settings.MAX_SOCKET_LISTEN) #监听数量，调用setting.py的变量

    def run_forever(self):
        '''启动socket server'''
        print("start FTP server on = %s:%s".center(50,'-') %(settings.HOST,settings.PORT))
        self.conn,self.addr = self.sock.accept()
        printf("connect from %s..." %(addr,))
        self.handle()

    def handle(self):
        '''接收的数据进行处理'''
        data = self.conn.recv(1024) #接收数据长度
        if not data: #如果没有数据
            pass

```

### settings.py

```

import os
HOST = "0.0.0.0"
PORT = 9999
BASE_DIR = os.path.dirname(os.path.dirname(os.path.abspath(__file__))) #得到ftp根目录路径

USER_HOME_DIR = os.path.join(BASE_DIR,'home') #在上上级目录下创建新的home目录路径，这个home就是用户的目录
print(BASE_DIR)

MAX_SOCKET_LISTEN = 5

```

```

root@ubuntu:/home/xzz/pyserver/ftp/bin# python3 ftp_server.py start
/home/xzz/pyserver/ftp
['/home/xzz/pyserver/ftp/bin', '/usr/lib/python35.zip', '/usr/lib/python3.5/lib-dynload', '/usr/local/lib/python3.5/dist-packages/ftp']
/home/xzz/pyserver/ftp
==INIT== start

    start FTP
    stop  FTP
    restart FTP
    creat username

out = ['ftp_server.py', 'start']
-----start FTP server on = 0.0.0.0:9999-----

```

FTP 服务器启动

## 用户认证实现

先来介绍 ini 文件使用

xxxx.ini 这种ini格式的配置文件，格式规则如下

1.section 段 用[]表示

就是某一类数据所属的一段位置

例子如下:

```
[user1]    这就表示user1段  
.....    该段下面的字符串数  
.....    据都属于user1段
```

2.keys和values 表示段下面的数据

例如:

```
[user1]  
name = 1111    这些键值对属于  
pass = 2222    user1段  
键值      value
```

```
[user1]  
name = xzz  
password = 123456  
expore = 2022-01-01  
  
[user2]  
name = username
```

这就是 accounts.ini 文件

```
import configparser 解析. ini 配置文件头文件
```

```
对象 = configparser.ConfigParser() #创建解析. ini 文件的对象
```

```
configparser.read(读取文件)
```

读取文件：传入文件名/路径

```
返回列表 = configparser.sections() #获取 ini 文件所有段
```

```
[user1]  
name = xzz  
password = 123456  
expore = 2022-01-01  
  
[user2]  
name = username
```

```
import configparser  
obj = configparser.ConfigParser()  
obj.read('accounts.ini') #读取当前目录的.ini配置文件  
print(obj.sections())#对象.sections只是读取配置文件的配置名，也就是[ ]包裹的字符
```

```
root@ubuntu:/home/xzz/pyserver/ftp/config# python3 test.py  
['user1', 'user2']
```

得到文件两个段的列表字符

返回键值对应的 value = 对象[填入段字符][填入键值字符] #获取 ini 文件里面某一段键值对应的数据

```
[user1]  
name = xzz  
password = 123456  
expore = 2022-01-01  
  
[user2]  
name = username
```

```
import configparser  
obj = configparser.ConfigParser()  
obj.read('accounts.ini') #读取当前目录的.ini配置文件  
print(obj['user1']['name']) #指定段， 获取段里面键值name对应的数据
```

```
root@ubuntu:/home/xzz/pyserver/ftp/config# python3 test.py  
xzz
```

获取 user1 段，键值 name 对应的数据 xzz。

```
configparser.add_section(填入加入段的字符) #在 ini 文件中加入新的段
```

填入加入段的字符：字符串类型

```
configparser.write(open(ini 文件字符, 'w')) #加入段或者键值之后，一定要执行 write，才能成功写入。
```

```
import configparser  
obj = configparser.ConfigParser()  
obj.read('accounts.ini') #读取当前目录的 .ini 配置文件  
obj.add_section('section_1') #加入新的段  
obj.write(open('accounts.ini', 'w')) #向 ini 文件写入数据后一定要执行 write 保存  
[user1]  
name = xzz  
password = 123456  
expore = 2022-01-01  
  
[user2]  
name = username  
  
[section_1]
```

ini 文件写入后形成新的段，[section\_1]

```
configparser.set(新加的键值对属于的段名, 键, 值) #在指定段加入新的键值对
```

```
import configparser  
obj = configparser.ConfigParser()  
obj.read('accounts.ini') #读取当前目录的 .ini 配置文件  
#obj.add_section('section_1') #如果新段加入了，如果再执行程序写同样的段名会出错，所以这里以后需要做判断  
obj.set('section_1', 'key_1', 'value_1') #用 set 方法向指定的段写键值和对应的 value  
obj.set('section_1', 'key_2', 'value_2') #用 set 方法向指定的段写键值和对应的 value  
obj.set('section_1', 'key_3', 'value_3') #用 set 方法向指定的段写键值和对应的 value  
obj.write(open('accounts.ini', 'w')) #向 ini 文件写入数据后一定要执行 write 保存
```

```
[user1]  
name = xzz  
password = 123456  
expore = 2022-01-01  
  
[user2]  
name = username  
  
[section_1]  
key_1 = value_1  
key_2 = value_2  
key_3 = value_3
```

这就是新加入的键值对

python 基础文档(字典章节做形参将漏的部分)，字典传入函数必须带\*\*

```
def func1(**keyword):  
    print(keyword['xzz'])  
    print(keyword['xiang'])  
  
tinydict={'xzz':12345,'xiang':717171}  
  
func1(tinydict)#这是我们常规认为字典传参方式
```

```
Traceback (most recent call last):
  File "test.py", line 9, in <module>
    func1(tinydict)#这是我们常规认为字典传参方式
TypeError: func1() takes 0 positional arguments but 1 was given
```

运行报错: takes 0 positional arguments but 1 was given

字典传参方式我们理解错误了, 其实应该是下面这样

```
def func1(**keyword):
    print(keyword['xzz'])
    print(keyword['xiang'])

tinydict={'xzz':12345,'xiang':717171}

func1(**tinydict) #字典传参加入 **才对
root@ubuntu:/home/xzz/pyserver/ftp/core# python3 test.py
12345
717171
```

python 基础文档(类中的 self 不是随便用来给函数传参的)遗漏部分

```
class A(object):
    def __init__(self):
        pass

    def func1(self):
        data = 20
        self.func2(data)

    def func2(self,data):
        print(data)

    def run(self):
        self.func1();#类内部中的函数自己自己类的其它函数, 都需要加self

obj = A()
obj.run()
root@ubuntu:/home/xzz/pyserver/ftp/core# python3 test.py
20
```

这样测试没有问题

```
class A(object):
    def __init__(self):
        pass

    def func1(self):
        data = 20
        self.func2(self,data);#如果我好奇, 将func2的self也看成是形参, 类似C语言的形参思维, 传入self形参

    def func2(self,data):
        print(data)

    def run(self):
        self.func1();#类内部中的函数自己自己类的其它函数, 都需要加self

obj = A()
obj.run()
```

报错 takes

2 positional, 这是因为 self 虽然是函数形参, 但是在类中是不需要传参的。类中 self 有特殊意义

下面回到 FTP 项目来, 实现用户认证阶段

## 主要修改 main.py

```
import socket
from config import settings #导入config目录下的settings.py
import json
import configparser ##解析ini文件

class FTPServer(object):
    '''处理与客户端所有的交互，socket'''
    def __init__(self,management_input):
        self.management_input = management_input #得到 class ManagementTool类里面的self.sysargv这些self定义的参数
        self.sock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        self.sock.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1) #socket.socket端口占用问题解决方式
        self.sock.bind((settings.HOST,settings.PORT)) #绑定IP和端口，调用setting.py的变量
        self.sock.listen(settings.MAX_SOCKET_LISTEN) #监听数量，调用setting.py的变量
        self.accounts = self.load_accounts() #将ini文件内容赋值给类全局变量 ['user1', 'user2']

    def run_forever(self):
        '''启动socket server'''
        print("start FTP server on = %s:%s".center(50,'-') %(settings.HOST,settings.PORT))
        self.conn,self.addr = self.sock.accept()
        #printf("connect from %s..." %(addr,))
        self.handle()

    def handle(self):
        '''接收的数据进行处理'''
        data = self.conn.recv(1024) #接收数据长度
        raw_data = json.loads(data.decode('utf-8')) #客户端发来的数据是json格式，这儿用json加载数据流
        action_type = raw_data.get('action_type') #获取json字典中action_type的value
        if action_type:
            if action_type == 'auth':
                print(raw_data)
                self.authenticate(**raw_data) #因为解码之后是字典，所以形参定义要是字典，不然会报错，字典传参必须是**
            else:
                pass
        else:
            print("action type none")

    def load_accounts(self): #获取ini文件信息
        self.config_obj = configparser.ConfigParser() #创建解析ini文件的对象
        self.config_obj.read('../config/accounts.ini') #读取上级目录config/下的ini文件
        return self.config_obj.sections() #获取ini文件下的段，返回列表 ['user1', 'user2']

    def authenticate(self,user,password):
        print("user = ",user)
        if 'user1' in self.accounts: #['user1', 'user2']只要包含user1，就为真
            _password = self.config_obj[self.accounts[0]]['password'] #获取ini文件用户下user1的密码
            if password == _password: #socket传入的密码和ini文件的密码相等，为真
                print("user success")
                return True
            else:
                print("password error")
                return False

    def auth(self,**xdict): #因为解码之后是字典，所以形参定义要是字典，不然会报错
        print(xdict["username"])
        print(xdict['password'])
        self.authenticate(xdict["username"],xdict['password']) #将socket得到的用户和密码传入
```

客户端程序如下：

```
import socket
import json
#FTP client
class ftpClient(object):
    def __init__(self):
        self.connection()

    def connection(self): #connect server
        self.sock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        self.sock.connect(('192.168.0.112',9999)) #connect ftp server

    def auth(self): #set user password
        username = 'xzz'
        password = '123456'
        cmd = {
            'action_type':'auth',
            'username': username,
            'password': password,
        }

        self.sock.send(json.dumps(cmd).encode('utf-8')) #send user password
        self.sock.recv(1024)

    def interactive(self):
        if self.auth():
            pass

if __name__ == "__main__":
    client = ftpClient()
    client.interactive()
```

## FTP 固定报文头实现

zfill 函数用法，为字符串，或者数组，列表定义长度，如果不满足长度的空间用 0 填充

### zfill的用法

- 用法: `newstr = string.zfill(width)`
- 参数: `width` 新字符串希望的宽度

```
1 | In [14]: name = 'insane'
2 | In [15]: new_name = name.zfill(10)
3 | In [16]: print(new_name)
4 | 0000000000insane
```

## 服务端程序

```
import socket
from config import settings #导入config目录下的settings.py
import json
import configparser ##解析ini文件

class FTPServer(object):
    '''处理与客户端所有的交互，socket'''
    def __init__(self,management_input):
        self.management_input = management_input #得到class ManagementTool类里面的self.sysargv这些self定义的参数
        self.sock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        self.sock.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1) #socket.socket端口占用问题解决方式
        self.sock.bind((settings.HOST,settings.PORT)) #绑定IP和端口，调用setting.py的变量
        self.sock.listen(settings.MAX_SOCKET_LISTEN) #监听数量，调用setting.py的变量
        self.accounts = self.load_accounts() #将ini文件内容赋值给类全局变量 ['user1', 'user2']

    def run_forever(self):
        '''启动socket server'''
        print("start FTP server on = %s:%s".center(50,'-') %(settings.HOST,settings.PORT))
        self.conn,self.addr = self.sock.accept()
        #printf("connect from %s..." %(addr,))
        self.handle()

    def handle(self):
        '''接收的数据进行处理'''
        data = self.conn.recv(1024) #接收数据长度
        raw_data = json.loads(data.decode('utf-8')) #客户端发来的数据是json格式，这儿用json加载数据流
        action_type = raw_data.get('action_type') #获取json字典中action_type的value
        if action_type:
            if action_type == 'auth':
                print(raw_data)
                self.auth(**raw_data) #因为解码之后是字典，所以形参定义要是字典，不然会报错，字典传参必须是**
            else:
                pass
        else:
            print("action_type none")

    def load_accounts(self): #获取ini文件信息
        self.config_obj = configparser.ConfigParser() #创建解析ini文件的对象
        self.config_obj.read('../config/accounts.ini') #读取上级目录config/下的ini文件
        return self.config_obj.sections() #获取ini文件下的段，返回列表 ['user1', 'user2']
```

```

def authenticate(self, user, password):
    print("user = ", user)
    if 'user1' in self.accounts: #['user1', 'user2']只要包含user1, 就为真
        _password = self.config_obj[self.accounts[0]]['password'] #获取ini文件用户下user1的密码
        if password == _password: #socket传入的密码和ini文件的密码相等, 为真
            print("user success") #密码正确返回True
            return True
        else:
            print("password error")
            return False

def send_response(self, status_code, args): #打包发送消息给客户端
    STATUS_CODE = {
        200:"passed",
        201:"worng user or password",
        300:"file does not exist !"
    }
    #1. 打包数据包
    data = {}
    data['status_code'] = status_code #消息码给字典
    data['status_msg'] = STATUS_CODE[status_code] #消息码对应的字符串给字典
    data['fill'] = '' #报文头第1次编码
    bytes_data = json.dumps(data).encode('utf-8') #报文头第一次编码

    MSG_SIZE = 1024 #固定服务器返回给客户端报头的长度
    if len(bytes_data) < MSG_SIZE:
        data['fill'] = data['fill'].zfill(MSG_SIZE - len(bytes_data)) #报文头如果不满足客户端的1024字节长度接收用zfill填0补充到1024字节
        bytes_data = json.dumps(data).encode('utf-8')

    print(bytes_data)
    self.conn.send(bytes_data)

def auth(self, **xdict): #因为解码之后是字典, 所以形参定义要是字典, 不然会报错
    print(xdict['username'])
    print(xdict['password'])
    flag = self.authenticate(xdict['username'], xdict['password']) #将socket得到的用户名和密码传入
    if flag == True:
        self.send_response(200,4444) #发送响应码200证明密码用户验证成功
    else:
        self.send_response(300,4444) #发送响应码300证明密码用户验证失败

```

用户密码验证之后, 要返回  
给客户端的固定报文

因为是报头, 所以报文  
头做的固定最大长度报  
文, 也就是最大不超过  
1024字节

用户密码验证后, 决定返  
回200还是300响应码

## 客户端程序

```

import socket
import json

#FTP client
class ftpClient(object):
    def __init__(self):
        self.connection()

    def connection(self): #connect server
        self.sock = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        self.sock.connect(('192.168.0.112',9999)) #connect ftp server

    def get_response(self):
        MSG_SIZE = 1024 #recv data max length 1024
        data = self.sock.recv(MSG_SIZE)
        return json.loads(data.decode('utf-8'))

    def auth(self): #set user password
        username = 'xzz'
        password = '123456'

        cmd = {
            'action_type':'auth',
            'username': username,
            'password': password,
        }

        self.sock.send(json.dumps(cmd).encode('utf-8')) #send user password
        response = self.get_response() #server response data
        print("response = ",response)

    def interactive(self):
        if self.auth():
            pass

if __name__ == "__main__":
    client = ftpClient()
    client.interactive()

```

客户端主要是实现接收服务器返回的数据

这是服务器发送的数据，不足 1024 字节的用 0 补齐

客户端接收数据，不足 1024 字节的，也是收到 0 补齐现象。

以上都是 FTP 简化的实现方法，后面讲解标准的 FTP 客户端实现方式

## Linux 中 FTP 服务器搭建(使用 vsftpd 软件)

vsftpd 并不是 sftp 协议，而是 FTP 协议的轻量级软件。

```
root@ubuntu:/home/xzz# vsftpd -v  
The program 'vsftpd' is currently not installed. You can install it by typing:  
apt install vsftpd
```

使用 vsftpd -v 查看版本的方式查看系统是否安装 vsftpd 软件

1. 如果没有安装 vsftpd，执行 sudo apt-get install vsftpd 安装

## 2. 修改 etc 目录下 vsftpd.conf 配置文件

```
/home/xzz# vim /etc/vsftpd.conf  
27 # Uncomment this to allow local users to log in.  
28 local_enable=YES 本地使能一定要打开  
# Allow anonymous FTP? (Disabled by default).  
anonymous_enable=NO 静止用户匿名访问本机 ftp
```

```
122 chroot_local_user=YES  
123 chroot_list_enable=YES  
124 # (default follows)  
125 chroot_list_file=/etc/vsftpd.chroot_list
```

注意：chroot list file= /etc/vsftpd.chroot list 这个

`vsftpd_chroot_list` 里面就是你支持多少个用户和密码的配置文件。

vsftpd\_chroot\_list 这个文件要自己创建

```
30 # Uncomment this to enable any form of FTP write command.  
31 write_enable=YES
```

```
14 listen=NO
15 #
16 # This directive enables listening
17 # on the IPv6 "any" address
18 # and IPv4 clients. It is
19 # sockets. If you want the
20 # addresses) then you must
21 # files.
22 listen ipv6=YES
```

listen\_ipv6。但是我发现打开 ipv6 后，winscp 软件连接的时候会报错（由于目标计算机积极拒绝，无法连接）。

22 #listen\_ipv6=YES 所以我屏蔽 ipv6，我本来地址就是 ipv4 的，所以开 ipv4。

14 listen=YES 打开 ipv4

```
local_root=/home/xzz/FTP  
#winSCP登陆的时候需要跳转到的默认目录，如果不设置默认目录路径，那么winSCP登陆就会直接跳转到系统用户根目录下  
allow_writeable_chroot=YES  
pasv_enable=YES  
#pasv_address=192.168.0.112 #内网不存在，如果是公网服务器，要考虑下设置  
pasv_min_port=40000  
pasv_max_port=45000
```

添加以上内容

3. 修改 sudo vim /etc/pam.d/vsftpd

```
2 auth required pam_listfile.so item=user sense=deny file=/etc/ftpusers onerr=succeed  
3 #auth required pam_nologin.so
```

确保 pam\_listfile.so 是正常打开

4. 启动 ftp 服务，执行 sudo /etc/init.d/vsftpd restart 或者是 service vsftpd restart

```
root@ubuntu:/home/xiang/test# service vsftpd restart  
stop: Unknown job: vsftpd  
start: Unknown job: vsftpd  
root@ubuntu:/home/xiang/test# sudo service vsftpd restart  
vsftpd stop/waiting  
vsftpd start/pre-start, process 5068
```

如果出现 Unknown job：那么就加入 sudo 执行

Ubuntu14.04 以上版本使用命令 sudo restart vsftpd 代替 service vsftpd restart

service vsftpd status 查看服务状态

service vsftpd stop 停止

service vsftpd restart 重启

```
root@ubuntu:/home/xiang# sudo service vsftpd status  
vsftpd stop/waiting
```

发现 vsftpd 还是未运行，卸载 vsftpd，然后重新 apt-get 安装就可以了。

ubuntu 查看系统指定端口是否被开放

```
root@ubuntu:/home/xzz# lsof -i:80 lsof 执行后未返回证明80端口未开放  
root@ubuntu:/home/xzz# lsof -i:22 lsof 指定22端口后，有返回，证明22端口开放  
COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME  
sshd 982 root 3u IPv4 29107 0t0 TCP *:ssh (LISTEN)  
sshd 982 root 4u IPv6 29109 0t0 TCP *:ssh (LISTEN)  
root@ubuntu:/home/xzz# lsof -i:21 lsof查看21端口，居然开放在IPV6网段  
COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME  
vsftpd 3889 root 3u IPv6 55794 0t0 TCP *:ftp (LISTEN)
```

也可以使用 `sudo ufw status` 查看端口是否被开启。

```
sudo ufw allow 80 //打开 80 端口
```

```
sudo ufw enable //防火墙开启
```

```
sudo ufw reload //防火墙重启
```

```
netstat -aptn #查看所有开放端口
```

500 OOPS: vsftpd: both local and anonymous access disabled! 本地和匿名访问都被禁用

```
24 # Allow anonymous FTP? (Disabled by default).  
25 anonymous enable=YES
```

使能 `vsftpd.conf` 匿名访问配置，其实可以不用。

```
500 OOPS: cannot change directory:/FTP_dir
```

```
root@ubuntu:/home/xzz# setsebool -P ftpd_disable_trans 1  
setsebool: SELinux is disabled.
```

执行成功之后，返回 SELinux 成功关闭

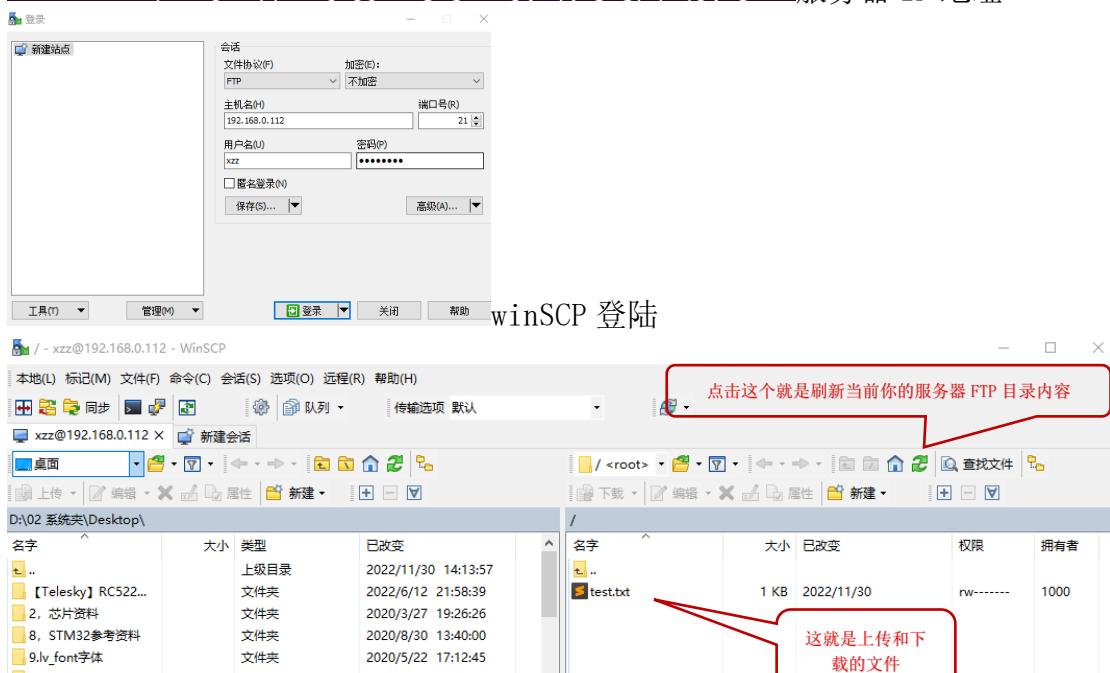
## 登陆 FTP 服务器

1. 确认自己的系统用户

```
root@ubuntu:/home/xzz/FTP# ls  
test.txt
```

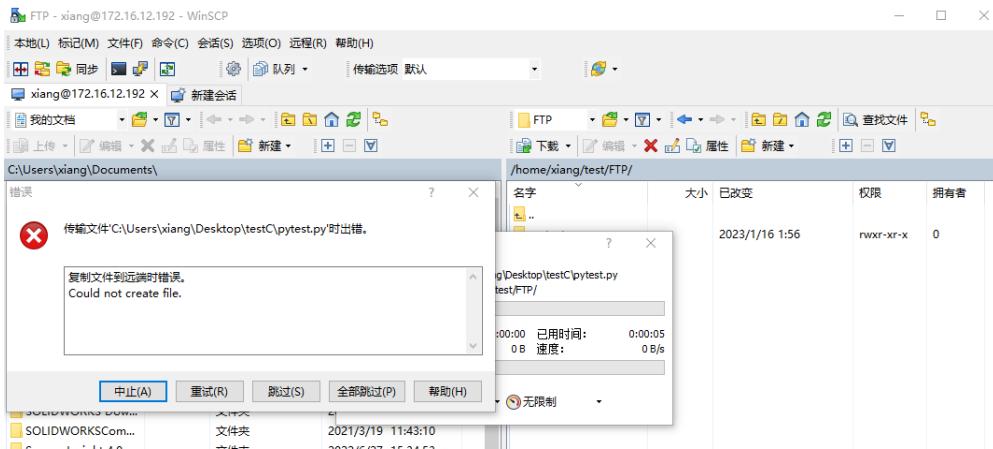
我的系统用户名是 xzz，密码是……。就是你每次登陆 ubuntu 的用户名和密码。我这里没有做新增系统用户，所以我就用自己的 ubuntu 用户。

```
xzz@ubuntu:~$ ifconfig  
ens33      Link encap:Ethernet  HWaddr 00:0c:29:7e:81:a7  
          inet  addr:192.168.0.112  Bcast:192.168.0.255  服务器 IP 地址
```



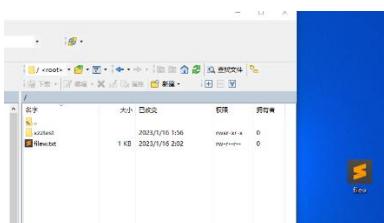
下面我们用自己写的 FTP 客户端试试。

还有一种情况，成功登录 FTP 服务器，但是无法传输文件



第1步，先看看文件是否能下载

```
root@ubuntu:/home/xiang/test/FTP# ls
filew.txt  xzztest
```



实际测试，文件是可以下载的。但是为什么不能上

传呢？

```
root@ubuntu:/home
filew.txt  FTP
root@ubuntu:/home
```

这是权限问题造成的，将 FTP 根目录设置成 chmod 777 FTP，就可以了

## Wireshark 抓包 VMware 虚拟机与本地 PC 主机的通信数据

VMware 虚拟机安装完成之后，会出现三张网卡，分别是 VMnet0, VMnet1, VMnet8,

名称	类型	外部连接	主机连接	DHCP	子网地址
VMnet0	桥接模式	2x2 11b/g/n Wireless LAN M...	-	-	-
VMnet1	仅主机...	-	已连接	已启用	192.168.150.0
VMnet8	NAT 模式	NAT 模式	已连接	已启用	192.168.62.0

其中 VMnet1, VMnet8，对应本地 PC 主机上的 VMware Network Adapter VMnet1 和 VMware Network Adapter VMnet8 两块虚拟网卡。



**Bridged (桥接模式)**： VMnet0 就是桥接模式的网卡，虚拟机的 IP 地址与真实 PC 主机 IP 地址在同一个网段，比如真实主机 ip 是 192.168.0.10，那么虚拟机可以是 192.168.0.112

在桥接模式下，windows 主机与虚拟机可以设置共享目录。所以桥接模式下，虚拟机会真实占用路由器的 ip，所以路由器 ip 不是很多的情况下，少用桥接模式。

在 Wireshark 或者真实 PC 主机中，网络适配器是没有 VMware Network Adapter VMnet0 的，网卡不可见，这相当于虚拟机是一台真实的物理 PC 主机。

**Host-Only**（仅主机模式）：VMnet1 就是仅主机模式，虚拟机和真实 PC 主机会在同一个私网中，但是不在同一个网段中。真实 PC 主机不会使用自己的网卡，而会使用 VMnet1 这个虚拟网卡，而非真实的物理网卡。这时候会有一个虚拟交换机连接 VMware 和真实的主机。

VMware 启动之后会从 192.168.x.x 中随机分配一个网段给真实主机，其它虚拟机也通过 DHC 获取 192.168.x.x 分配的网段。这相当于虚拟机在仅主机模式下变成了局域网。这种情况下

**VMware Network Adapter VMnet1** 是可见的，但是虚拟机不能与虚拟机组的局域网以外的公网进行连接。

**NAT**（地址转换模式）：如果虚拟机想用单独自己的网段连接外网，就可以使用这种方式。

如果你的网络 ip 资源紧缺，但是你又希望你的虚拟机能够联网，这时候 NAT 模式是最好的选择。

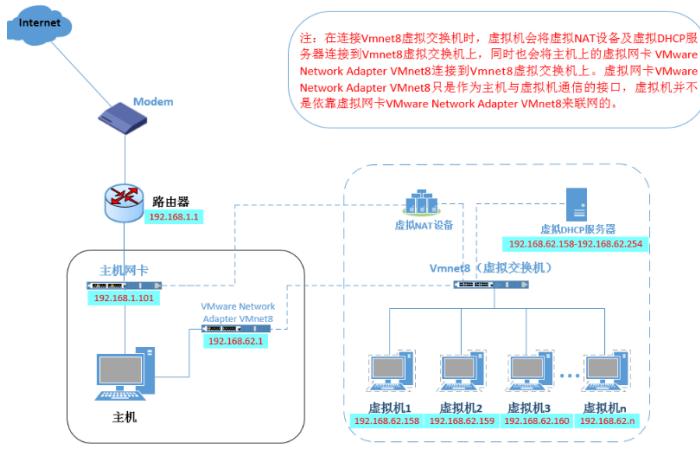


图2 NAT模式

在 NAT 模式中，主机网卡直接与虚拟 NAT 设备相连，然后虚拟 NAT 设备与虚拟 DHCP 服务器一起连接在虚拟交换机 VMnet8 上，这样就实现了虚拟机联网。那么我们会觉得很奇怪，为什么需要虚拟网卡 VMware Network Adapter VMnet8 呢？原来我们的 VMware Network Adapter VMnet8 虚拟网卡主要是为了实现主机与虚拟机之间的通信。

我现在虚拟机使用的是桥接模式

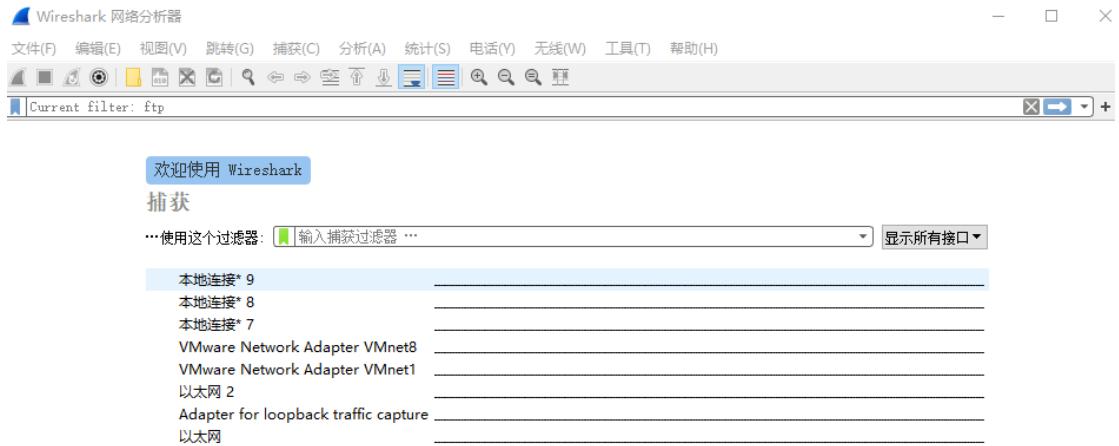
Wireshark 使用方法

因为我现在是本地 PC 主机与虚拟机进行 ftp 协议文件传输。所以先确认本地主机 IP

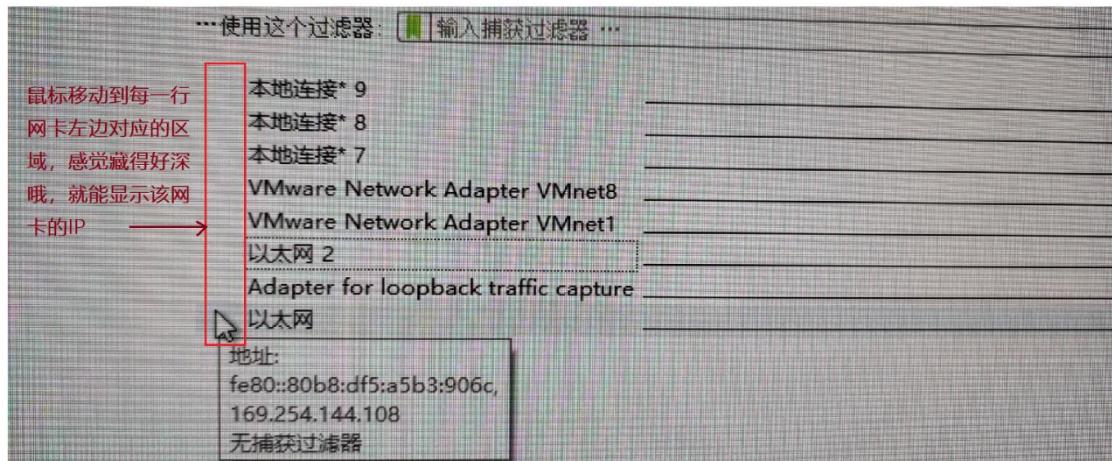
## 属性

本地链接 IPv6 地址: fe80::b90d:7cd2:4084:1e12%14  
IPv4 地址: 192.168.0.105  
IPv4 DNS 服务器: 192.168.0.1  
制造商: Realtek  
描述: Realtek PCIe GbE Family Controller  
驱动程序版本: 10.45.928.2020  
物理地址(MAC): 2C-F0-5D-0E-62-A0

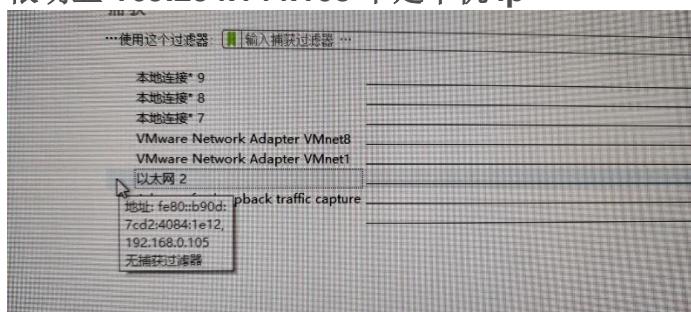
在本地主机适配器中找到本机 IP 为  
**192.168.0.105**



打开软件之后你不知道哪一张网卡名是你本地主机的，那么将鼠标移动到左边



很明显 169.254.144.108 不是本机 ip



以太网 2 才是本机 IP，双击进入

正在捕获 以太网 2

文件(F) 编辑(E) 视图(V) 跳转(G) 捕获(C) 分析(A) 统计(S) 电话(Y) 无线(W) 工具(T) 帮助(H)

Current filter: ftp

No.	Time	Source	Destination	Protocol	Length	Info
17763	8.236192	192.168.0.105	192.168.0.112	FTP	62	Request: T
17765	8.236492	192.168.0.112	192.168.0.105	FTP	85	Response: T
18169	8.502807	192.168.0.105	192.168.0.112	FTP	62	Request: T
18171	8.503017	192.168.0.112	192.168.0.105	FTP	84	Response: T
18173	8.503761	192.168.0.105	192.168.0.112	FTP	60	Request: P
18175	8.503887	192.168.0.112	192.168.0.105	FTP	106	Response: T
18177	8.503982	192.168.0.105	192.168.0.112	FTP	63	Request: L
18178	8.504244	192.168.0.112	192.168.0.105	FTP	93	Response: T
18197	8.504453	192.168.0.112	192.168.0.105	FTP	78	Response: T

> Frame 17763: 62 bytes on wire (496 bits), 62 bytes captured (496 bits) on interface '\Device\NPF\_{A856BF88-E74D-40A2-87F6-E300013F0B60}'  
> Ethernet II, Src: Micro-St\_0e:62:a0 (2c:0:5d:0e:62:a0), Dst: VMware\_7e:81:a7 (00:0c:29:7e:81:a7)  
> Internet Protocol Version 4, Src: 192.168.0.105, Dst: 192.168.0.112  
> Transmission Control Protocol, Src Port: 21, Seq: 1, Ack: 1, Len: 8  
> File Transfer Protocol (FTP)  
[Current working directory: ]

我打开 winscp, FTP 软件, wireshark 就收到了 FTP 数据包

主机与虚拟机通信的数据包  
所以只需要知道一个机器的网卡就可以做抓包  
工作了, 不需要再去知道虚拟机网卡

正在捕获 以太网 2

文件(F) 编辑(E) 视图(V) 跳转(G) 捕获(C) 分析(A) 统计(S) 电话(Y) 无线(W) 工具(T) 帮助(H)

Current filter: 手机发往虚拟机数据包, 根据IP来判断的数据协议

No.	Title	Source	Destination	Protocol	Length	Info
18169	8.502807	192.168.0.105	192.168.0.112	FTP	62	Request: TYPE A
18171	8.503017	192.168.0.112	192.168.0.105	FTP	84	Response: 200 Switching to ASCII mode.
18173	8.503761	192.168.0.105	192.168.0.112	FTP	60	Request: PASV
18175	8.503887	192.168.0.112	192.168.0.105	FTP	106	Response: 227 Entering Passive Mode (192.168.0.105:21)
18177	8.503982	192.168.0.105	192.168.0.112	FTP	63	Request: LIST -a
18178	8.504244	192.168.0.112	192.168.0.105	FTP	93	Response: 150 Here comes the directory

虚拟机发往主机数据包  
每一行数据就是一个包

正在捕获 以太网 2

文件(F) 编辑(E) 视图(V) 跳转(G) 捕获(C) 分析(A) 统计(S) 电话(Y) 无线(W) 工具(T) 帮助(H)

如果我想看其行数据包真实数据, 双击就是

No.	Title	Source	Destination	Protocol	Length	Info
18175	8.503887	192.168.0.112	192.168.0.105	FTP	106	Request: 227 Entering Passive Mode
-	18178	8.504244	192.168.0.112	192.168.0.105	FTP	63 Request: LIST -a
-	18187	8.504453	192.168.0.112	192.168.0.105	FTP	93 Response: 150 Here comes the directo
-	18197	8.504453	192.168.0.112	192.168.0.105	FTP	93 Response: 226 Directory send OK.
-	71395	38.236447	192.168.0.105	192.168.0.112	FTP	62 Request: REST 0
-	71397	38.236447	192.168.0.105	192.168.0.112	FTP	98 Response: 350 Restart position accept
-	72371	68.236887	192.168.0.105	192.168.0.112	FTP	62 Request: REST 1
-	72372	68.236888	192.168.0.105	192.168.0.112	FTP	98 Response: 200 Switching to Binary mo
-	72373	68.236889	192.168.0.105	192.168.0.112	FTP	62 Request: REST 1
-	74545	98.234142	192.168.0.112	192.168.0.105	FTP	98 Response: 350 Restart position accept
-	75780	128.239272	192.168.0.105	192.168.0.112	FTP	62 Request: TYPE A

Header checksum: 0x0000 [validation disabled]  
[Header checksum status: Unverified]  
Source Address: 192.168.0.105

得到数据包对应的16进制

正在捕获 以太网 2 我在看视频, 我在这里输入http就能得到视频传输的http报文, 从而把ftp过滤掉

正在捕获 以太网 2 我输入ftp, 就过滤掉其它报文, 只看ftp

正在捕获 以太网 2

文件(F) 编辑(E) 视图(V) 跳转(G) 捕获(C) 分析(A) 统计(S) 电话(Y) 无线(W) 工具(T) 帮助(H)

No.	Title	Source	Destination	Protocol	Length	Info
3938	8.708795	192.168.0.105	205.119.109.16	HTTP	608	POST / HTTP/1.1
3945	1.812009	203.219.169.16	192.168.0.105	HTTP/1.1	107	HTTP/1.1 200 OK , JavaScript Object No
36691	18.115888	192.168.0.105	119.8.83.21	HTTP	55	Continuation
46929	22.856815	192.168.0.105	203.119.109.16	HTTP	540	POST /a HTTP/1.1
46941	22.900210	203.119.109.16	192.168.0.105	HTTP/1.1	107	HTTP/1.1 200 OK , JavaScript Object No
67145	33.059161	192.168.0.105	119.8.83.21	HTTP	566	GET /hotwords.json HTTP/1.1
67148	33.117573	119.8.83.21	192.168.0.105	HTTP	391	HTTP/1.1 304 Not Modified
69029	34.916535	192.168.0.105	192.168.0.112	HTTP	604	PUT /log HTTP/1.1
69226	34.916535	115.236.118.34	192.168.0.105	HTTP/1.1	76	HTTP/1.1 200 , JavaScript Object No
71237	34.936712	192.168.0.105	183.47.118.249	HTTP	827	POST /mtls/0000417d HTTP/1.1
71247	34.965569	192.168.0.105	192.168.0.1	HTTP/X	667	POST /ct/IPCConn HTTP/1.1

Frame 3945: 107 bytes on wire (856 bits), 107 bytes captured (856 bits) on interface '\Device\NPF\_{A856BF88-E74D-40A2-87F6-E300013F0B60}'  
Interface Id: 0 (\Device\NPF\_{A856BF88-E74D-40A2-87F6-E300013F0B60})  
Encapsulation type: Ethernet (1)

Header checksum: 0x0000 [validation disabled]  
[Header checksum status: Unverified]  
Source Address: 192.168.0.105  
Destination Address: 192.168.0.112  
Transmission Control Protocol, Src Port: 21, Dst Port: 21, Seq: 1, Ack: 1, Len: 8

如果感觉窗口的数据包太多了, 想清除窗口历史内容, 就只有点击重新捕获

正在捕获 以太网 2

文件(F) 编辑(E) 视图(V) 跳转(G) 捕获(C) 分析(A) 统计(S) 电话(Y) 无线(W) 工具(T) 帮助(H)

Current filter: ftp

No.	Time	Source	Destination	Protocol	Length	Info
21	1.360650	192.168.0.105	192.168.0.112	FTP	62	Request: TYPE A
23	1.361141	192.168.0.112	192.168.0.105	FTP	84	Response: 200 Switching to ASCII mode.

## 标准 FTP 实现

```
root@ubuntu:/home/xzz/FTP# ls  
test.txt  
root@ubuntu:/home/xzz/FTP# service vsftpd restart  
服务器启动 FTP 服务端，标准的 vsftpd 服务
```

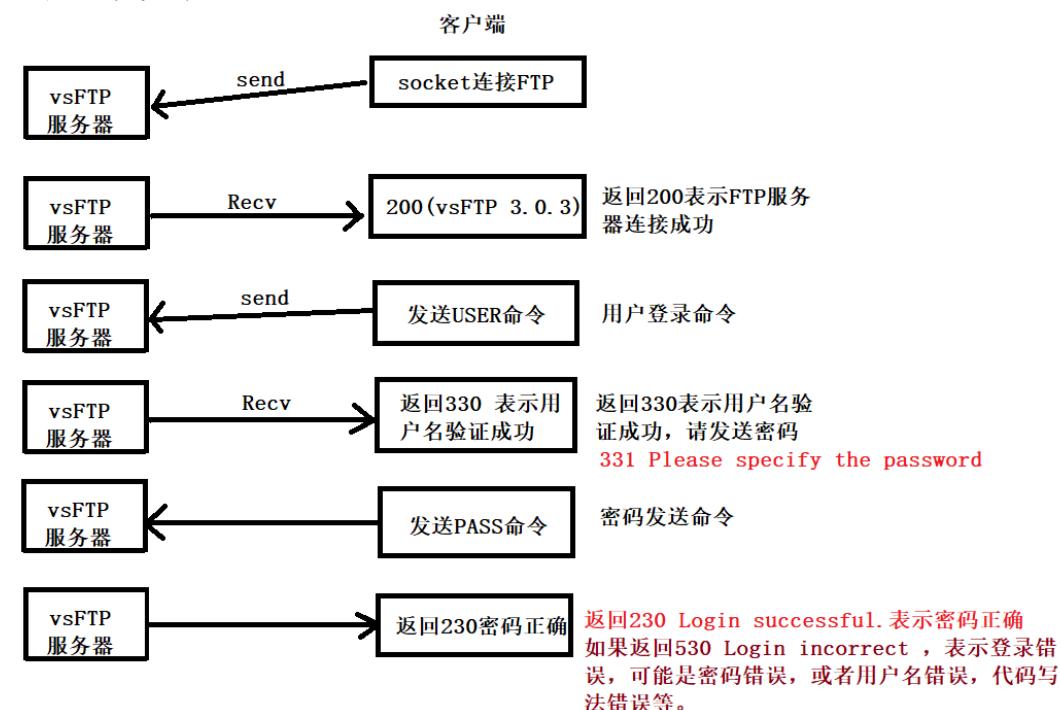
```
import socket  
import struct  
import json  
  
client = socket.socket(socket.AF_INET,socket.SOCK_STREAM)  
client.connect(('192.168.0.112',21))  
  
data = client.recv(1024) #接收FTP服务器返回的数据  
print(data)  
client.close()
```

客户端发送连接到 FTP 服务器

220 (vsFTPD 3.0.3) 服务器返回当前 FTP 服务器版本。

这里 220 代表 Service ready for new user，就是服务端已经准备好对一个新用户开放了。

登陆流程如下：



```

import socket
import struct
import json

client = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
client.connect(('192.168.0.112',21))

data = client.recv(1024)
print(data)
client.sendall('USER xzz' + '\r\n') #USER name login
data = client.recv(1024)
print(data)
client.sendall('PASS 6751147' + '\r\n') #password login
data = client.recv(1024)
print(data)
client.close()

```

客户端发送用户名必须是 **USER** + 你自己服务器的用户名，最后必须是'\r\n'结尾。这是 **FTP** 命令规则。

客户端发送密码必须是 **PASS** + 你自己服务器的密码，最后必须是'\r\n'结尾。这是 **FTP** 命令规则。

```

220 (vsFTPD 3.0.3)

331 Please specify the password.

230 Login successful.

```

登陆成功

```

client.sendall('PASS 67511476' + '\r\n') #password login
data = client.recv(1024)
print(data)

client.sendall('TYPE A' + '\r\n') #ASCII transfer
data = client.recv(1024)
print(data)

client.sendall('PASV' + '\r\n') #Passive Mode
data = client.recv(1024)
print(data)

client.close()

```

```

230 Login successful.

200 Switching to ASCII mode.

227 Entering Passive Mode (192,168,0,112,167,162).

```

登陆成功之后，  
发送 **TYPE A** 表示传输方式采用 **ASCII** 方式

发送 **TYPE B** 表示传输方式采用二进制方式

发送 **PASV** 表示客户端进入被动模式

被动模式返回的 227 Entering Passive Mode (192,168,0,112,167,162).

(192,168,0,112,167,162)

前面4个表示vsFTP服  
务器的IP

后面两个表示vsFTP服  
务器的端口号，要进行  
运算得到一个端口号

公式为  $p1 * 256 + p2$

意思就是  $167 * 256 + 162 = 42914$

这之后我们新开一个 `socket` 对象来连接这个 (192.168.0.112) IP 的  
42914 端口，用于数据流。而原来 21 端口创建的 `socket` 保留用来做控制流

## FTP PWD 命令

**PWD** 显示当前登录后所处于的目录路径

```
root@ubuntu:/home/xiang/test/FTP# ls  
pytest.py  xzztest
```

现在 FTP

是根目录，应该显示 “/”

```
root@ubuntu:/home/xiang/test/FTP/xzztest# ls  
test1.txt  test2.txt  test3.txt
```

现在要实现进入 **xzztest** 目录，显示三个文本文件

## FTP CWD 命令

**CWD** 目的是切换目录，进入自己指定的目录

```
import socket  
  
client = socket.socket(socket.AF_INET,socket.SOCK_STREAM) #连接服务器  
client.connect(('172.18.251.153',21))  
data = client.recv(1024)  
print(data)  
  
client.sendall(('USER xiang' + '\r\n').encode('utf-8')) #有些编辑器编码问题，所以封装  
成 encode utf8  
data = client.recv(1024)  
print(data)  
  
client.sendall(('PASS 67511476' + '\r\n').encode('utf-8'))  
data = client.recv(1024)  
print(data)  
  
client.sendall(('TYPE A' + '\r\n').encode('utf-8'))  
data = client.recv(1024)  
print(data)  
  
client.sendall(('PASV' + '\r\n').encode('utf-8'))  
data = client.recv(1024)  
print(data)
```

```

client.sendall((('PWD' + '\r\n').encode('utf-8')) #当前目录
data = client.recv(1024)
print(data)

client.sendall((('CWD xzztest' + '\r\n').encode('utf-8')) #进入指定目录 xzztest
data = client.recv(1024)
print(data)

client.sendall((('PWD' + '\r\n').encode('utf-8')) #查看当前的目录
data = client.recv(1024)
print(data)

client.close()

```

```

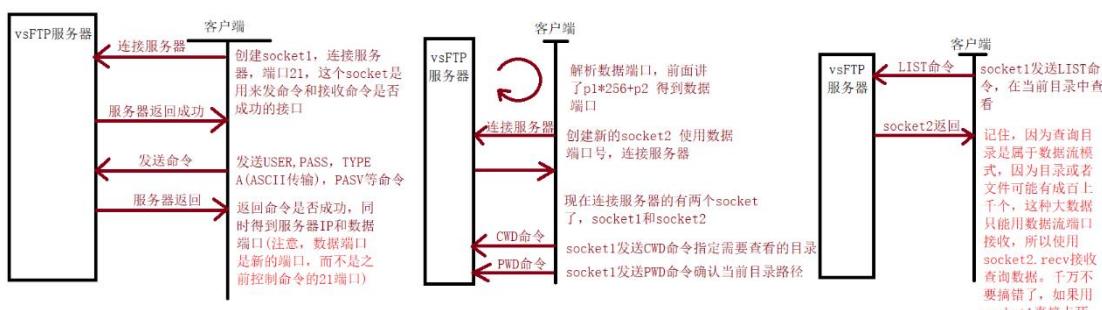
b'220 (vsFTPd 3.0.2)\r\n' 连接服务器
b'331 Please specify the password.\r\n'
b'230 Login successful.\r\n' 登录成功
b'200 Switching to ASCII mode.\r\n' 切换成ASCII字符串传输
b'227 Entering Passive Mode (172,18,251,153,173,71).\r\n' 得到数据流端口
b'257 "/"\r\n' 当前目录为FTP根目录
b'250 Directory successfully changed.\r\n' 使用CWD命令切换到新目录
b'257 "/xzztest"\r\n' 当前新目录为xzztest目录

```

查看目录正常返回码为 257, CWD 目录切换成功返回码为 250

## FTP 目录列表/文件列表查询

FTP目录列表/文件列表查询分两步走



这就是两步走的逻辑，socket1和socket2

```

import socket

client = socket.socket(socket.AF_INET,socket.SOCK_STREAM) #连接服务器
client.connect(('172.18.251.153',21))
data = client.recv(1024)
print(data)
client.sendall((('USER xiang' + '\r\n').encode('utf-8')) #有些编辑器编码问题，所以封装成 encode utf8
data = client.recv(1024)
print(data)
client.sendall((('PASS 67511476' + '\r\n').encode('utf-8'))

```

```

data = client.recv(1024)
print(data)
client.sendall((('TYPE A' + '\r\n').encode('utf-8')))
data = client.recv(1024)
print(data)
client.sendall((('PASV' + '\r\n').encode('utf-8')))
data = client.recv(1024)
print(data)

#将得到的 IP 地址保持不变，数据流端口号变成元组形式提取
StramData = data.decode('utf-8')
strpro = StramData[26:] #获取中括号里面 IP 和端口
xstr = strpro.rstrip('.\r\n') #取消掉结尾的换行符和‘.’
xtuple = eval(xstr) #将 IP 和端口转换成元组形式
print(xtuple)

StreamPort = xtuple[4]*256+xtuple[5] #得到数据流新端口

StreamClient = socket.socket(socket.AF_INET,socket.SOCK_STREAM) #连接服务器
StreamClient.connect(('172.18.251.153',StreamPort))

client.sendall((('PWD'+ '\r\n').encode('utf-8')) #当前目录
data = client.recv(1024)

client.sendall((('CWD xzztest'+ '\r\n').encode('utf-8'))) #进入指定目录 xzztest
#data = client.recv(1024)

client.sendall((('PWD'+ '\r\n').encode('utf-8'))) #查看当前的目录
data = client.recv(1024)
print(data)

client.sendall((('LIST'+ '\r\n').encode('utf-8'))) #查看当前目录内容
sdata = StreamClient.recv(1024) #使用 socket2 来得到查询目录的数据

print(sdata.decode('utf-8')) #获取的数据一定要解码，不然数据看起来不对齐
client.close()
StreamClient.close()

```

```

b'220 (vsFTPD 3.0.2)\r\n'
b'331 Please specify the password.\r\n'
b'230 Login successful.\r\n'
b'200 Switching to ASCII mode.\r\n'
b'227 Entering Passive Mode (172,18,251,153,170,187).\r\n'
(172, 18, 251, 153, 170, 187)
b'250 Directory successfully changed.\r\n'
b'257 "/xzztest"\r\n'
150 Here comes the directory listing.\r\n
-rw-r--r--    1 0        0           26 Jan 15 18:43 test1.txt

-rw-r--r--    1 0        0           24 Jan 15 18:43 test2.txt

-rw-r--r--    1 0        0           33 Jan 15 18:43 test3.txt

```

得到服务端 FTP 指定目录下，查询的文件

## FTP 客户端下载文件

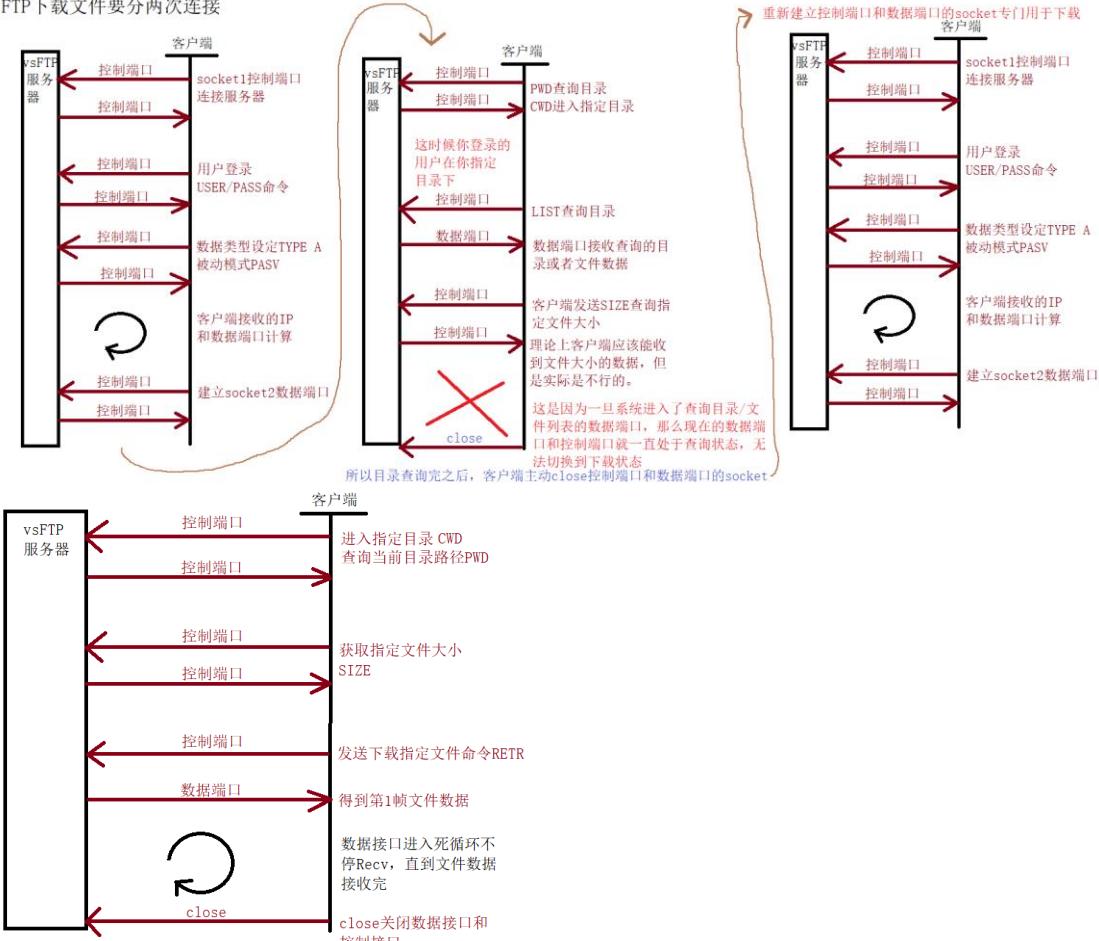
### FTP SIZE 命令

查询指定文件的大小，下载文件时要先查询文件大小

### FTP RETR 命令

控制类 socket 端口发出下载命令，然后用数据端口的 socket 接收

FTP 下载文件要分两次连接



```
root@ubuntu:/home/xiang/test/FTP/xzztest# ls  
test1.txt test2.txt test3.txt
```

还是下载服务器上 FTP 目录下 test1.txt 文件

代码例程

```
import socket  
  
client = socket.socket(socket.AF_INET,socket.SOCK_STREAM) #第 1 次 socket1 控制端口连接服务器  
client.connect(('172.18.251.153',21))  
data = client.recv(1024)  
print(data)  
  
client.sendall((b'USER xiang\r\n').encode('utf-8')) #有些编辑器编码问题，所以封装成 encode utf8  
data = client.recv(1024)  
print(data)  
  
client.sendall((b'PASS 67511476\r\n').encode('utf-8')) #控制端口用户登录  
data = client.recv(1024)  
print(data)  
  
client.sendall((b'TYPE A\r\n').encode('utf-8')) #控制端口数据类型设定，ASCII 传输  
data = client.recv(1024)  
print(data)  
  
client.sendall((b'PASV\r\n').encode('utf-8')) #控制端口要求进入被动模式
```

```

data = client.recv(1024)
print(data)

#将得到的 IP 地址保持不变，数据流端口号变成元组形式提取(客户端接收的 IP 和数据端口计算)
StramData = data.decode('utf-8')
strpro = StramData[26:] #获取中括号里面 IP 和端口
xstr = strpro.rstrip('.\r\n') #取消掉结尾的换行符和‘.’
xtuple = eval(xstr) #将 IP 和端口转换成元组形式
print(xtuple)

StreamPort = xtuple[4]*256+xtuple[5] #得到数据端口

StreamClient = socket.socket(socket.AF_INET,socket.SOCK_STREAM) #建立 socket2 数据端口连接服务器
StreamClient.connect(('172.18.251.153',StreamPort))

client.sendall((('PWD'+ '\r\n').encode('utf-8')) #查询当前目录路径
data = client.recv(1024)
print(data)

client.sendall((('CWD xzztest'+ '\r\n').encode('utf-8')) #CWD 进入指定目录 xzztest
data = client.recv(1024)
print(data)

client.sendall((('PWD'+ '\r\n').encode('utf-8')) #查看当前进入的指定目录路径
data = client.recv(1024)
print(data)

client.sendall((('LIST'+ '\r\n').encode('utf-8')) #查看当前目录内容
sdata = StreamClient.recv(1024) #使用 socket2 来得到查询目录的数据
print(sdata.decode('utf-8')) #获取的数据一定要解码，不然数据看起不对齐

#这时候如果我在查询目录使用的数据端口之后，再写下载命令的程序就会死机

client.close() #只有关闭第 1 次控制端口 socket1
StreamClient.close() #只有关闭第 1 次数据端口 socket1

#####
#下载， 下载必须第 2 次连接控制端口和数据端口 socket2#####
client = socket.socket(socket.AF_INET,socket.SOCK_STREAM) #第 2 次连接服务器 socket1 控制端口
client.connect(('172.18.251.153',21))
data = client.recv(1024)
print(data)

client.sendall((('USER xiang' + '\r\n').encode('utf-8')) #有些编辑器编码问题，所以封装成 encode utf8
data = client.recv(1024)
print(data)

client.sendall((('PASS 67511476' + '\r\n').encode('utf-8')))
data = client.recv(1024)
print(data)

client.sendall((('TYPE A' + '\r\n').encode('utf-8')))
data = client.recv(1024)
print(data)

client.sendall((('PASV' + '\r\n').encode('utf-8')))
data = client.recv(1024)
print(data)

#将得到的 IP 地址保持不变，数据流端口号变成元组形式提取
StramData = data.decode('utf-8')
strpro = StramData[26:] #获取中括号里面 IP 和端口
xstr = strpro.rstrip('.\r\n') #取消掉结尾的换行符和‘.’
xtuple = eval(xstr) #将 IP 和端口转换成元组形式
print(xtuple)

StreamPort = xtuple[4]*256+xtuple[5] #得到数据流新端口

StreamClient = socket.socket(socket.AF_INET,socket.SOCK_STREAM) #第 2 次数据端口 socket2
StreamClient.connect(('172.18.251.153',StreamPort))

client.sendall((('CWD xzztest'+ '\r\n').encode('utf-8')) #进入指定目录 xzztest
data = client.recv(1024)
print(data)

client.sendall((('PWD'+ '\r\n').encode('utf-8')) #查看当前的目录
data = client.recv(1024)

```

```

print(data)

#直接放弃查询当前目录的内容，避免使用数据端口，这样可以将数据端口留给下载的时候用

client.sendall((SIZE test1.txt' + '\r\n').encode('utf-8')) #查看指定文件大小
data = client.recv(1024)
print(data)

client.sendall(('RETR test1.txt' + '\r\n').encode('utf-8')) #下载
data = StreamClient.recv(1024) #使用数据端口接收文件数据
print(data)

client.close()
StreamClient.close()

```

```

b'220 (vsFTPd 3.0.2)\r\n'
b'331 Please specify the password.\r\n'
b'230 Login successful.\r\n' 第1次socket1连接的内容
b'200 Switching to ASCII mode.\r\n'
b'227 Entering Passive Mode (172,18,251,153,169,57).\r\n'
(172, 18, 251, 153, 169, 57)
b'257 "/"\r\n'
b'250 Directory successfully changed.\r\n'
b'257 "/xzztest"\r\n' 查询目录内容消耗了数据接口
-rw-r--r--    1 0        0            26 Jan 15 18:43 test1.txt

-rw-r--r--    1 0        0            24 Jan 15 18:43 test2.txt

-rw-r--r--    1 0        0            33 Jan 15 18:43 test3.txt

b'220 (vsFTPd 3.0.2)\r\n'
b'331 Please specify the password.\r\n'
b'230 Login successful.\r\n' 第2次socket1/2连接的
b'200 Switching to ASCII mode.\r\n'
b'227 Entering Passive Mode (172,18,251,153,158,132).\r\n'
(172, 18, 251, 153, 158, 132)
b'250 Directory successfully changed.\r\n'
b'257 "/xzztest"\r\n' 得到文件大小26字节，本来文件是25字节，不
b'213 26\r\n'           知道是空格还是换行占用了1字节
b'11111111111111111111111111111111\r\n' 第2次数据接口获得文件数据

```

## FTP 文件上传

### FTP STOR 命令

STOR 是文件上传命令

```

root@ubuntu:/home/xiang/test/FTP/xzztest# ls
test1.txt  test2.txt  test3.txt

```

将设备端文件上传到服务

器 FTP/xzztest 目录下。

```

#####
#上传文件#####
client = socket.socket(socket.AF_INET,socket.SOCK_STREAM) #socket1 控制端口建立
client.connect(('172.18.251.153',21))
data = client.recv(1024)
print(data)

client.sendall(('USER xiang' + '\r\n').encode('utf-8')) #有些编辑器编码问题，所以封装
成 encode utf8
data = client.recv(1024)
print(data)

client.sendall(('PASS 67511476' + '\r\n').encode('utf-8'))
data = client.recv(1024)
print(data)

```

```

client.sendall((('TYPE A' + '\r\n').encode('utf-8'))
data = client.recv(1024)
print(data)

client.sendall((('PASV' + '\r\n').encode('utf-8'))
data = client.recv(1024)
print(data)

#将得到的 IP 地址保持不变，数据流端口号变成元组形式提取
StramData = data.decode('utf-8')
strpro = StramData[26:] #获取中括号里面 IP 和端口
xstr = strpro.rstrip('.\r\n') #取消掉结尾的换行符和‘.’
xtuple = eval(xstr) #将 IP 和端口转换成元组形式
print(xtuple)

StreamPort = xtuple[4]*256+xtuple[5] #得到数据流新端口

StreamClient = socket.socket(socket.AF_INET,socket.SOCK_STREAM) #数据端口 socket2
StreamClient.connect(('172.18.251.153',StreamPort))

client.sendall((('CWD xzztest' + '\r\n').encode('utf-8')) #进入指定目录 xzztest
data = client.recv(1024)
print(data)
client.sendall((('PWD'+ '\r\n').encode('utf-8')) #查看当前的目录
data = client.recv(1024)
print(data)

#下面是上传文件

client.sendall((('STOR upload.txt' + '\r\n').encode('utf-8')) #上传文件，指定文件名，服务器会自动创建文件
data = client.recv(1024)
print(data)

StreamClient.sendall((('12345678910' + '\r\n').encode('utf-8')) #向上传的文件中写数据，用数据端口

data = client.recv(1024) #使用控制端口接收返回值，不知道对不对
print(data)
client.close()
StreamClient.close()

b'220 (vsFTPd 3.0.2)\r\n'
b'331 Please specify the password.\r\n'
b'230 Login successful.\r\n'
b'200 Switching to ASCII mode.\r\n'
b'227 Entering Passive Mode (172,18,251,153,167,157).\r\n'
(b'172, 18, 251, 153, 167, 157)
b'250 Directory successfully changed.\r\n'
b'257 "/xzztest"\r\n'
b'553 Could not create file.\r\n'

```

文件上传过程中发现，无法在 FTP 服务器建立文件。这是因为你的 **FTP/xzztest** 目录的问题。也就是 **FTP** 目录是权限放开的，**chmod 777 FTP**，但是 **FTP** 下面的子目录 **xzztest** 权限是没放开的。所以你可以上传文件到 **FTP** 目录，但是上传不到 **FTP/xzztest** 目录。

```
root@ubuntu:/home/xiang/test/FTP# ls  
pytest.py  xzztest  
root@ubuntu:/home/xiang/test/FTP# chmod 777 xzztest  
root@ubuntu:/home/xiang/test/FTP# ls  
pytest.py  xzztest
```

放开 xzztest 目录的权限

```
b'220 (vsFTPd 3.0.2)\r\n'  
b'331 Please specify the password.\r\n'  
b'230 Login successful.\r\n'  
b'200 Switching to ASCII mode.\r\n'  
b'227 Entering Passive Mode (172,18,251,153,173,78).\r\n'(172, 18, 251, 153, 173, 78)  
b'250 Directory successfully changed.\r\n'  
b'257 "/xzztest"\r\n'  
b'150 Ok to send data.\r\n'
```

文件上传成功

```
root@ubuntu:/home/xiang/test/FTP/xzztest# ls  
test1.txt  test2.txt  test3.txt  upload.txt
```

文件成功上传

```
1 12345678910  
~
```

文件内容正确

## FTP 实现大文件下载

需要前面的 TCP 黏包问题解决的知识。

### FTP QUIT 命令

退出 FTP 登陆，用在关闭 FTP 客户端软件的时候使用。

```
#coding=UTF-8 #解决 Vscode 中文编码无法运行问题  
import socket  
  
client = socket.socket(socket.AF_INET,socket.SOCK_STREAM) #控制端口 21, 连接  
client.connect(('192.168.0.112',21))  
data = client.recv(1024)  
print(data)  
  
client.sendall(('USER xzz' + '\r\n').encode('utf-8')) #数据编码问题，所以将命令封装成 utf-8  
data = client.recv(1024)  
print(data)  
  
client.sendall(('PASS 67511476' + '\r\n').encode('utf-8')) #控制端密码登陆  
data = client.recv(1024)  
print(data)  
  
client.sendall(('TYPE A' + '\r\n').encode('utf-8')) #ASCII 码传输
```

```

data = client.recv(1024)
print(data)

client.sendall((('PASV' + '\r\n').encode('utf-8'))) #被动模式
data = client.recv(1024)
print(data)

StramData = data.decode('utf-8')
strpro = StramData[26:] #获取中括号里面 IP 和端口
xstr = strpro.rstrip('.\r\n') #取消掉结尾的换行符和‘.’
xtuple = eval(xstr) #将 IP 和端口转换成元组形式
print(xtuple)

StreamPort = xtuple[4]*256+xtuple[5] #得到数据端口

StreamClient = socket.socket(socket.AF_INET,socket.SOCK_STREAM) #建立 socket2 数据端口连接服务器
StreamClient.connect(('192.168.0.112',StreamPort))

client.sendall((('PWD'+ '\r\n').encode('utf-8'))) #当前目录
data = client.recv(1024)
print(data)

client.sendall((('CWD xzztest'+ '\r\n').encode('utf-8'))) #进入指定目录 xzztest
data = client.recv(1024)
print(data)

client.sendall((('PWD'+ '\r\n').encode('utf-8'))) #当前目录
data = client.recv(1024)
print(data)

client.sendall((('LIST'+ '\r\n').encode('utf-8'))) #查看当前目录内容
sdata = StreamClient.recv(1024) #使用 socket2 来得到查询目录的数据
print(sdata.decode('utf-8')) #获取的数据一定要解码, 不然数据看起来不对齐

client.close()
StreamClient.close()

#####
#####大文件下载#####
client = socket.socket(socket.AF_INET,socket.SOCK_STREAM) #控制端口 21, 连接
client.connect(('192.168.0.112',21))
data = client.recv(1024)
print(data)

client.sendall((('USER xzz' + '\r\n').encode('utf-8'))) #数据编码问题, 所以将命令封装成 utf-8
data = client.recv(1024)
print(data)

client.sendall((('PASS 67511476' + '\r\n').encode('utf-8'))) #控制端密码登陆
data = client.recv(1024)
print(data)

client.sendall((('TYPE A' + '\r\n').encode('utf-8'))) #ASCII 码传输
data = client.recv(1024)
print(data)

client.sendall((('PASV' + '\r\n').encode('utf-8'))) #被动模式
data = client.recv(1024)
print(data)

StramData = data.decode('utf-8')
strpro = StramData[26:] #获取中括号里面 IP 和端口
xstr = strpro.rstrip('.\r\n') #取消掉结尾的换行符和‘.’
xtuple = eval(xstr) #将 IP 和端口转换成元组形式
print(xtuple)

StreamPort = xtuple[4]*256+xtuple[5] #得到数据端口

StreamClient = socket.socket(socket.AF_INET,socket.SOCK_STREAM) #建立 socket2 数据端口连接服务器
StreamClient.connect(('192.168.0.112',StreamPort))

client.sendall((('CWD xzztest'+ '\r\n').encode('utf-8'))) #进入指定目录 xzztest
data = client.recv(1024)
print(data)

```

```

client.sendall((('PWD' + '\r\n').encode('utf-8')) #查看当前的目录
data = client.recv(1024)
print(data)

#直接放弃查询当前目录的内容，避免使用数据端口，这样可以将数据端口留给下载的时候用

client.sendall((('SIZE test1.txt' + '\r\n').encode('utf-8')) #查看指定文件大小
data = client.recv(1024)
print(data)

DecodeData = data.decode('utf-8') #解码文件大小
RecvFileSize = int(DecodeData[4:]) #截取 213 功能码后面的文件大小值
print(RecvFileSize) #要下载的文件大小，字节为单位

client.sendall((('RETR test1.txt' + '\r\n').encode('utf-8')) #下载

rSIZE = 0 #接收了多大的文件
currentSIZE = 0 #当前 socket 接收文件数据的长度

wf = open('F:\\data\\test\\test1.txt', 'w') #因为无法加载中文路径，所以选择了英文路径盘符

while rSIZE < RecvFileSize: #文件数据没接收完，死循环下载
    data = StreamClient.recv(1024) #使用数据端口接收文件数据
    currentSIZE = len(data)
    rSIZE = rSIZE + currentSIZE #本次接收数据大小 + 上次接收数据大小
    wf.write(data)

wf.flush() #文件保存到磁盘
wf.close() #关闭文件

StreamClient.close() #一定是先关闭数据端口
client.sendall((('QUIT' + '\r\n').encode('utf-8')) #控制端口发送 QUIT 退出 FTP 登录
data = client.recv(1024)
print(data)

client.close() #关闭控制端口

print("下载结束")

```

1 111111111111111111111111111111111111  
2 111111111111111111111111111111111111  
3 111111111111111111111111111111111111  
4 111111111111111111111111111111111111  
5 111111111111111111111111111111111111  
6 111111111111111111111111111111111111  
7 111111111111111111111111111111111111  
8 111111111111111111111111111111111111  
9 111111111111111111111111111111111111  
10 111111111111111111111111111111111111  
11 111111111111111111111111111111111111  
12 111111111111111111111111111111111111  
13 111111111111111111111111111111111111  
14 111111111111111111111111111111111111  
15 111111111111111111111111111111111111  
16 111111111111111111111111111111111111  
17 111111111111111111111111111111111111  
18 111111111111111111111111111111111111  
19 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31  
20 32,33,34,35,36,37,38,39,40

这是服务端 test1.txt 文件，数字部分是超出了 1024

字节后写的。

```

220 (vsFTPd 3.0.3)          下载部分的代码执行回显

331 Please specify the password.

230 Login successful.        登陆成功

200 Switching to ASCII mode.

227 Entering Passive Mode (192,168,0,112,160,161). 被动模式

(192, 168, 0, 112, 160, 161)
250 Directory successfully changed.

257 "/xzztest" is the current directory 进入服务器FTP/xzztest目录

213 1065 下载test1.txt文本文件，得到文件大小，213是SIZE命令执行成功，1065字节
          是文件大小
1065 将1065字符串转换成整形
150 Opening BINARY mode data connection for test1.txt (1065 bytes).
226 Transfer complete.发送QUIT命令后显示的，传输完成

```

下载结束

这是客户端程序执行过程

客户端接收到的数据正确无

误，和服务端 test1.txt 内容一致

下面传输一部 mv 文件试试

```
99 client.sendall(('SIZE cylj.rmvb' + '\r\n').encode('utf-8')) #查看指定文件大小
100 data = client.recv(1024)          修改文件名为cylj, 获取服务器MV视频文件的大小
101 print(data)
102
103 DecodeData = data.decode('utf-8') #解码文件大小
104 RecvFileSize = int(DecodeData[4:]) #截取213功能码后面的文件大小值
105 print(RecvFileSize) #要下载的文件大小,字节为单位
106
107 client.sendall(('RETR cylj.rmvb' + '\r\n').encode('utf-8')) #下载一部MV
108
109 rSIZE = 0           #接收了多少大的文件
110 currentSIZE = 0 #当前socket接收文件数据的长度
111
112 wf = open('F:\\data\\test\\cylj.rmvb', 'w') #因为无法加载中文路径, 所以选择了英文路径盘符
113                                         创建mv文件
114 while rSIZE < RecvFileSize: #文件数据没接收完, 循环下载
115     data = StreamClient.recv(4096) #使用数据端口接收文件数据
116     currentSIZE = len(data)
117     rSIZE = rSIZE + currentSIZE #本次接收数据大小 + 上次接收数据大小
118     wf.write(data)

```

root@ubuntu:/home/xzz/FTP/xzztest# ls  
cylj.rmvb [ test1.txt test2.txt test3.txt  
这是要下载的文件

257 "/xzztest" is the current directory  
213 268453476 运行过程中得到mv文  
件是268M字节  
268453476  
150 Opening BINARY mode data connection for cylj.rmvb (268453476 bytes)  
下载结束  
但是这里没有显示传输完成

HDD (F): > datedest  
cylj.rmvb ← 最后播放下载的视频,  
居然是花的

我怀疑这不是TCP黏包, 而是TCP每一次  
send, 第1次send的不一定先到客户端, 有  
可能第2次send是先到客户端的。TCP数据时  
间不一致问题

其实不是 TCP 先后顺序的问题，而是文件打开方式的问题

```
wf = open('F:\\data\\test\\cylj.rmvb', 'wb') #因为无法加载中文路径，所以选择了英文路径盘符。(注意视频文件要二进制写入)
                                                将'w'改成'wb'二进制写入就对了
while rSIZE < RecvFileSize: #文件数据没接收完，死循环下载
    data = StreamClient.recv(4096) #使用数据端口接收文件数据
    currentSIZE = len(data)
    rSIZE = rSIZE + currentSIZE #本次接收数据大小 + 上次接收数据大小
    wf.write(data)
```

这下，下载的视频文件播放流畅，无花屏。

## FTP 大文件下载，断点续传问题

## FTP REST 命令

这个 REST 就是断点续传命令

C 语言版本

```
1. ....
2. /* 命令 "REST offset\r\n" */
3. sprintf(send_buf,"REST %ld\r\n", offset); //文件断续下载,主要是REST命令,传入偏移字符
4. /* 客户端发送命令指定下载文件的偏移量 */
5. write(control_sock, send_buf, strlen(send_buf));
6. /* 客户端接收服务器的响应码和信息,
7. * 正常为 "350 Restarting at <position>. Send STORE or RETRIEVE to initiate transfer." */
8. read(control_sock, read_buf, read_len);
9. ....
10.
11. /* 命令 "RETR filename\r\n" */
12. sprintf(send_buf,"RETR %s\r\n",filename);
13. /* 客户端发送命令从服务器端下载文件,并且跳过该文件的前offset字节*/
14. write(control_sock, send_buf, strlen(send_buf));
15. /* 客户端接收服务器的响应码和信息, *
16. * 正常为 "150 Connection accepted, restarting at offset <position>" */
17. read(control_sock, read_buf, read_len);
18. ....
19. file_handle = open(disk_name, CRFLAGS, RWXALL);
20. /* 指向文件写入的初始位置 */
21. lseek(file_handle, offset, SEEK_SET);
22. ....
```

这里就不测试了，有项目需要自己实现。