第二十章套接服务

> 本文是Python通用编程系列教程，已全部更新完成，实现的目标是从零基础开始到精通Python编程语言。本教程不是对Python的内容进行泛泛而谈，而是精细化，深入化的讲解，共5个阶段，25章内容。所以，需要有耐心的学习，才能真正有所收获。虽不涉及任何框架的使用，但是会对操作系统和网络通信进行全局的讲解，甚至会对一些开源模块和服务器进行重写。学完之后，你所收获的不仅仅是精通一门Python编程语言，而且具备快速学习其他编程语言的能力，无障碍阅读所有Python源码的能力和对计算机与网络的全面认识。对于零基础的小白来说，是入门计算机领域并精通一门编程语言的绝佳教材。对于有一定Python基础的童鞋，相信这套教程会让你的Python水平更上一层楼。

一 远程控制

1. subprocess模块应用

我们写好的软件服务端都是部署在服务器上的，而服务器一般都是放在机房，由于成本问题，机房一般都是建在郊区甚至山区，不可能建在闹市区。那么我们需要查看服务器运行状态就需要远程登陆服务器，如果你非的要跑到山区去接显示器和键盘也没人拦着你。

服务端代码

```
from socket import *
import subprocess
server=socket(AF_INET,SOCK_STREAM)
server.bind(('127.0.0.1',8080))  # 127.0.0.1是本地回环地址，一般用来做测试
server.listen(5)
while True:
    conn,client_address=server.accept() #(连接对象，客户端的ip和端口)
    print(client_address)
    while True:
        try:
            cmd=conn.recv(1024)
            obj=subprocess.Popen(cmd.decode('utf-8'),
                                 shell=True,
                                 stdout=subprocess.PIPE,
                                 stderr=subprocess.PIPE
                                 )
            stdout=obj.stdout.read()
            stderr=obj.stderr.read()
            # 发送真实的数据
            conn.send(stdout)
            conn.send(stderr)
        except ConnectionResetError:
            break
    conn.close()
server.close()
```

客户端代码

```
from socket import *
client = socket(AF_INET, SOCK_STREAM)
client.connect(('127.0.0.1', 8080))
while True:
    cmd = input('>>>: ').strip()
    if not cmd: continue
    client.send(cmd.encode('utf-8'))
    res = client.recv(1024)
    print(res.decode('utf-8'))  # Windows系统为gbk
client.close()
```

这个代码就是我们上一章项目的作业，这就是一个远程控制的程序，是给运维人员所使用的，我们现在就是开发这样一个程序，当然他所能够执行的命令是非常有限的。

2. 粘包现象

上面程序写完了，除了一些命令不能实现，其实还是存在一些明显的问题的。Windows系统的同学可以分别执行 dir 命令和 ipconfig /all命令，查看结果，然后再执行dir命令，看一下和你第一次执行dir命令看到的结果是否有什么不同，MacOS系统的同学可以执行ls 命令和ifconfig，然后再执行ls命令，接下里的演示以MacOS系统为例。

分别执行ls命令和ifconfig命令后的结果

```
>>>: ls
01.py
02.py
03.py
04.py
05.py
06.py
07.py
08.py
09.py
10.py
11.py
12.py

>>>: ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 16384
    options=1203<RXCSUM,TXCSUM,TXSTATUS,SW_TIMESTAMP>
    inet 127.0.0.1 netmask 0xff000000
    inet6 ::1 prefixlen 128
    inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
    nd6 options=201<PERFORMNUD,DAD>
gif0: flags=8010<POINTOPOINT,MULTICAST> mtu 1280
stf0: flags=0<> mtu 1280
XHC20: flags=0<> mtu 0
en1: flags=8963<UP,BROADCAST,SMART,RUNNING,PROMISC,SIMPLEX,MULTICAST> mtu 1500
    options=60<TSO4,TSO6>
    ether 32:00:15:ad:20:00
```

再次执行ls命令的结果

```
    ether 32:00:15:ad:20:00
    media: autoselect <full-duplex>
    status: inactive
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
    ether 5c:f9:38:aa:e6:ce
    inet6 fe80::883:6c3d:da54:6529%en0 prefixlen 64 secured scopeid 0x6
    inet 192.168.43.138 netmask 0xffffff00 broadcast 192.168.43.255
    nd6 options=201<PERFORMNUD,DAD>
    media: autoselect
    status: active
bridge0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
    options=63<RXCSUM,TXCSUM,TSO4,TSO6>
    ether 32:00:15:ad:20:00
    Configuration:
        id 0:0:0:0:0:0 priority 0 hellotime 0 fwddel
>>>: ls
ay 0
        maxage 0 holdcnt 0 proto stp maxaddr 100 timeout 1200
        root id 0:0:0:0:0:0 priority 0 ifcost 0 port 0
        ipfilter disabled flags 0x2
    member: en1 flags=3<LEARNING,DISCOVER>
            ifmaxaddr 0 port 5 priority 0 path cost 0
    nd6 options=201<PERFORMNUD,DAD>
    media: <unknown type>
    status: inactive
p2p0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 2304
    ether 0e:f9:38:aa:e6:ce
```

你会发现再次执行ls命令是，打印出来的结果不是正常的结果，已经乱了。那么产生这个问题的原因是什么呢？要弄明白问题所在，我们可以试试在自己电脑上面输入以上命令会出现什么？

```
pogon!~ alberts ifconfig
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 16384
        options=1203<RXCSUM,TXCSUM,TXSTATUS,SW_TIMESTAMP>
        inet 127.0.0.1 netmask 0xff000000
        inet6 ::1 prefixlen 128
        inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
        nd6 options=201<PERFORMNUD,DAD>
gif0: flags=8010<POINTOPOINT,MULTICAST> mtu 1280
stf0: flags=0<> mtu 1280
XHC20: flags=0<> mtu 0
en1: flags=8963<UP,BROADCAST,SMART,RUNNING,PROMISC,SIMPLEX,MULTICAST> mtu 1500
        options=60<TSO4,TSO6>
        ether 32:00:15:ad:20:00
        media: autoselect <full-duplex>
        status: inactive
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        ether 5c:f9:38:aa:e6:ce
        inet6 fe80::883:6c3d:da54:6529%en0 prefixlen 64 secured scopeid 0x6
        inet 192.168.43.138 netmask 0xffffff00 broadcast 192.168.43.255
        nd6 options=201<PERFORMNUD,DAD>
        media: autoselect
        status: active
bridge0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        options=63<RXCSUM,TXCSUM,TSO4,TSO6>
        ether 32:00:15:ad:20:00
        Configuration:
                id 0:0:0:0:0:0 priority 0 hellotime 0 fwddelay 0
                maxage 0 holdcnt 0 proto stp maxaddr 100 timeout 1200
                root id 0:0:0:0:0:0 priority 0 ifcost 0 port 0
                ipfilter disabled flags 0x2
        member: en1 flags=3<LEARNING,DISCOVER>
                ifmaxaddr 0 port 5 priority 0 path cost 0
        nd6 options=201<PERFORMNUD,DAD>
        media: <unknown type>
        status: inactive
p2p0: flags=8843<UP,BROADCAST,RUNNING,SIMPLEX,MULTICAST> mtu 2304
        ether 0e:f9:38:aa:e6:ce
        media: autoselect
        status: inactive
awdl0: flags=8943<UP,BROADCAST,RUNNING,PROMISC,SIMPLEX,MULTICAST> mtu 1484
        ether 76:43:ba:aa:a2:a0
        inet6 fe80::7443:baff:feaa:a2a0%awdl0 prefixlen 64 scopeid 0x9
        nd6 options=201<PERFORMNUD,DAD>
        media: autoselect
```

通过在自己电脑终端上的测试，你可以发现，我们自己写的程序，显示出来的内容并没有错。他只是显示出来少了一部分，而当我们再次输入ls命令的时候，打印出来的内容会继续打印ifconfig命令未显示完整的内容。

这让你想到了什么，绞肉机，我自己是没有用过，但是我听到说相声的说过，绞肉机里面有两斤羊肉，你怎么让这两斤羊肉出来，那就是再放两斤羊肉把它顶出来，但是如果你放两斤牛肉进去，那么他出来一定是羊肉。

同理，如你所见，ls命令出来的结果其实是ifconfig残留的结果。至此，你会发现出现问题的原因其实就是客户端接收命令结果的时候超过1024个字节，我们输入的命令不可能超过1024个字节，所以我们只要把客

户端的recv修改的大一些就可以了，具体多大就够了，我们不清楚，但是只要足够大，他就不可能超过这个数字，我们改成102400，你会发现没有问题了，但是这样是不是就真的解决问题了呢？（你还是先改回1024吧，TCP协议没那么简单）

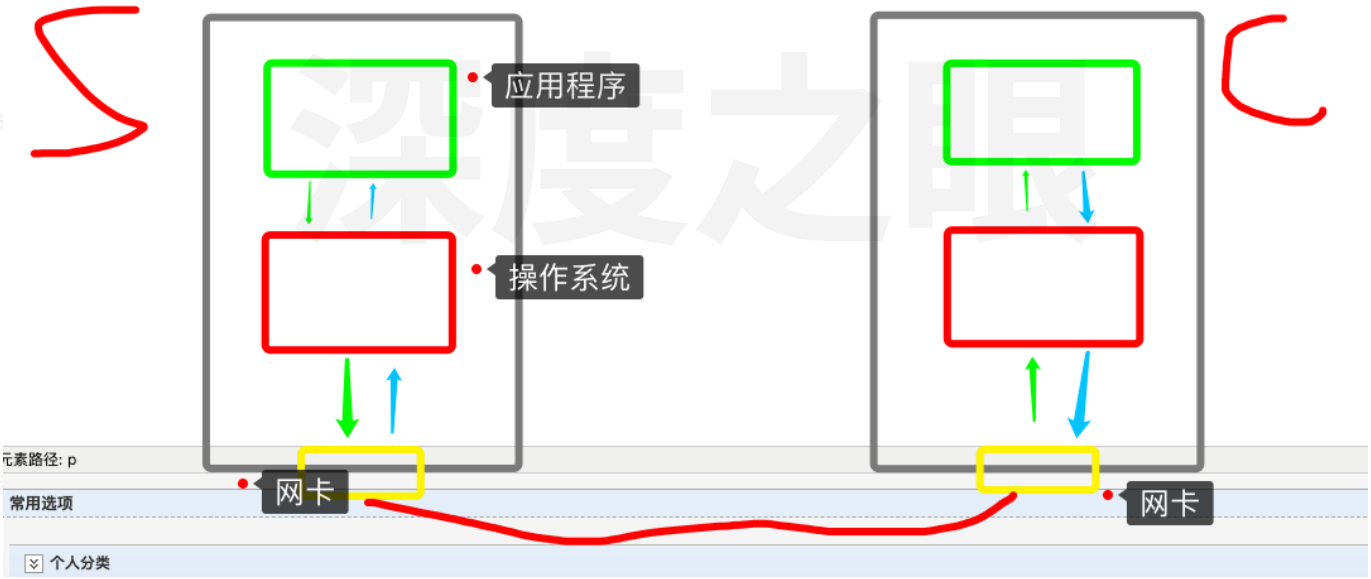
为了能让大家清楚知道命令的结果，我们可以控制服务端打印命令结果的字节长度，服务端代码修改如下

```
15                                  shell=True,
16                                  stdout=subprocess.PIPE,
17                                  stderr=subprocess.PIPE
18                                  )
19                  stdout = obj.stdout.read()
20                  stderr = obj.stderr.read()
21
22                  print(len(stdout + stderr))
23                  # 发送真实的数据
24                  conn.send(stdout)
25                  conn.send(stderr)
26              except ConnectionResetError:
27                  break
28
29      conn.close()
```

只需要添加第22行代码就可以了，执行你会发现，ifconfig命令的结果长度是1849，我们最大接收的长度是1024，所以还剩余825个字节留在管道里面，为了说是管道呢，因为TCP协议他是一个流式协议（像水流一样），接下来我们会先从逻辑意义上来解析。

如图所示，管道中总长度1024个字节，命令1的结果有825个字节，我们可用通过打印命令2结果的长度得知它的长度是72个字节，那么72+825<1024,也就是说我们按照顺序执行命令ls-ifconfig-ls后，第二次执

行ls命令的结果应该是有两部分，可你看一下结果并不是这样。这涉及到TCP协议的Nagle算法，接下来我们接下来我们先从工作流程图上来解释TCP协议，再慢慢揭开Nagle算法的面纱。



前面的图是逻辑意义图，而这个图是工作流程图，我们以为的TCP协议是直接发送给对方其实不然，上图绿色和红色分别表示应用程序和操作系统的内存，对于应用程序来说只需要完成send那一行代码之后，就是发送出去了，而其实只是把需要发送的数据复制给了操作系统内存一份，对于操作系统而言，这只是工作的开始，操作系统开始控制传输层和传输层以下下进行发送。对于接收的一方来说，先到达的也是它的操作系统，如果接收方没有recv或者recv没有接收完全，那么数据就会滞留在接收方的操作系统内，直到下一次发送方发送数据，接收方接收数据就会把残留的结果顶出来，这就是流式协议，也是产生粘包现象的原因。由于操作系统缓存的存在，才会产生上文中的第二次执行ls命令结果不是两部分的现象。

3. Nagle算法

流式协议有一种优化机制， 就是TCP协议的Nagle算法，是用来优化数据传输的，特点是：会将数据量比较小并且时间间隔比较短的数据合成一个包来发送，这样做的目的是为了尽可能减少网络IO（可以理解为网络延迟）操作同时兼并程序执行效率。

> 这就像是卡车运送苹果，我先给你一个苹果，紧接着再给你10箱苹果，你最好是两次一起运送这个苹果；但是如果我先给你一个苹果，过了一年，再给你一个苹果，你肯定要先运送一次之前的一个苹果，在运送一次后面的苹果，因为时间间隔已经远远超过了运送一次的时间；再如果我给了200吨苹果，紧接着又给了你500吨苹果，你这个小卡车一次顶多也就能运送100吨，第一次给的苹果你都一次运不完，就没必要等着两次一起来运送了。

现在我们来用代码验证一下Nagle的存在

验证一：数据量比较小时间间隔比较短的数据合成一个包来发送

服务端代码

```
from socket import *

server = socket(AF_INET, SOCK_STREAM)
server.bind(('127.0.0.1', 8080))
server.listen(5)

conn, client_address = server.accept()

res1 = conn.recv(1024)
print('第一次：', res1)
res2 = conn.recv(1)
print('第二次: ', res2)

conn.close()
server.close()
```

客户端代码

```
from socket import *

client = socket(AF_INET, SOCK_STREAM)
client.connect(('127.0.0.1', 8080))

client.send(b'hello')
client.send(b'world')

client.close()
```

小菜鸟级解决方案

客户端修改如下

```
from socket import *
import time

client = socket(AF_INET, SOCK_STREAM)
client.connect(('127.0.0.1', 8080))

client.send(b'hello')
time.sleep(1)  # 只要睡一秒就好
```

```
client.send(b'world')

client.close()
```

这样low的行为虽然解决了客户端的粘包问题，但是带来的隐患是服务端就没有可能产生粘包吗？如果我们把服务端按照如下修改，你再看一下打印结果

```
from socket import *

server = socket(AF_INET, SOCK_STREAM)
server.bind(('127.0.0.1', 8080))
server.listen(5)

conn, client_address = server.accept()

res1 = conn.recv(1)
print('第一次：', res1)
res2 = conn.recv(1024)
print('第二次: ', res2)

conn.close()
server.close()
```

大菜鸟级解决方案

服务端修改如下（客户端不做修改）

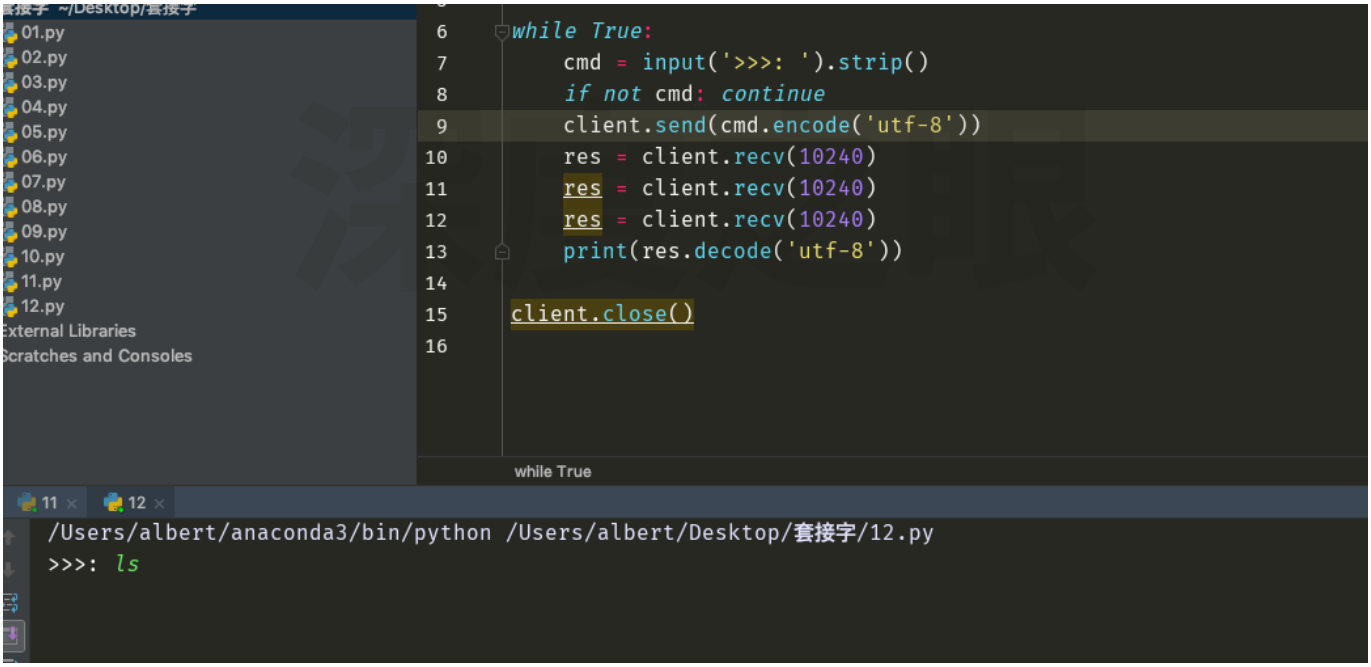
```
from socket import *

server = socket(AF_INET, SOCK_STREAM)
server.bind(('127.0.0.1', 8080))
server.listen(5)

conn, client_address = server.accept()

res1 = conn.recv(5)  # 发了五个就收五个
print('第一次：', res1)
res2 = conn.recv(5)
print('第二次: ', res2)
```

```
conn.close()
server.close()
```

现在你清楚了不管是接收和发送并不是直接和对方对接，而是通过自己的操作系统缓存，最开始我们设定的最大接收的值是1024个字节，如果要发送的数据是一个T，你把这个值也无限调大到一个T，那么，请你先给我造一台内存超过一个T的电脑出来。

> 涉及到操作系统缓存，其实就像是从大海里面捞鱼，鱼有很多，你不可能造一个无限大的网，但是你可以用一个小网重复捞，循环捞。

我们可以多次recv接收，那么怎么样才能算是接收干净了呢？第一个思路肯定是接收空了就是接收干净了呗，想法很美好，现实很残酷，实现不了，因为当客户端的接收为空了，也就是他自己的操作系统缓存为空，那么就会一直在原地等着服务端发送，而服务端其实已经发送出去了，那么客户端就会停在那里，也就是卡住了，你一定想写一个判断条件，我试过了，真的写不出来，能写出来的也会卡住，当然你也可以试试。



刚才的思路是一种直接的思路，接收空了就是接收干净了，除此之外我们还有另外一种思路，发送方发送了一万个字节，接收方接收了一万个字节也是接收干净了，就像刚才大菜鸟的解决方案一样。

> 需要注意的是：接收和发送并不是一一对应的关系，因为他们不是直接对接的，而是通过自己的操作系统来完成这个过程的

4. 解决粘包问题

要解决这个粘包问题的根本就是接收方必须要清楚的知道发送方发了多少数据，进而再进行循环接收，那么这样不可避免的涉及到了自定义报头，这这里补充一个小的知识点struct这个模块的用法。

```
import struct
res1 = struct.pack('i', 1231)  # 把数字转化成bytes，并且固定长度
res2 = struct.pack('i', 1)
res3 = struct.pack('i', 1473985)
print(res1,len(res1))
print(res2,len(res2))
print(res3,len(res3))
res4 = struct.unpack('i', res3)
print(res4)
```

解决TCP协议粘包问题

二 UDP协议通信

1. UDP与TCP比较说明

UDP通信不需要建立双向连接，也不需要等ack=1的确认信息，所以它的传输会比TCP快，但是通过UDP通信由发送方传给接收方的数据，一旦在传输过程中丢失，发送方也会把自己的缓存清理掉，而TCP协议则是必须要等到接收方回传ack=1才会清理自己的缓存，如是，UDP快但不安全，TCP慢但是安全，UDP快的主要原因是少了一个确认的回传信息，也就是少了一个网络延迟，而不是他不需要连接（当然这也要花时间），那么同理，TCP比UDP可靠也是因为它的确认回传信息，而不是它的双向连接。

2. UDP套接字

UDP通信不需要建立连接，但是服务端依然是需要绑定ip和端口的，程序启动的时候也可以先启动客户端，客户端不管服务端是否接收了，发了就完事。
服务端代码：

```
import socket
server=socket.socket(socket.AF_INET,socket.SOCK_DGRAM) # 数据报协议
server.bind(('127.0.0.1',8080))
while True:
    client_data,client_addr=server.recvfrom(1024)
```

```
    msg=input('回复%s:%s>>>:hello' %(client_addr[0],client_addr[1]))
    server.sendto(msg.encode('utf-8'),client_addr)
```

客户端代码：

```
import socket
client=socket.socket(socket.AF_INET,socket.SOCK_DGRAM) # 数据报协议
while True:
    msg=input('>>>: ').strip()
    client.sendto(msg.encode('utf-8'),('127.0.0.1',8080))
    res,server_addr=client.recvfrom(1024)
    print(res.decode('utf-8'))
```

你可以把以上客户端代码复制多份，同时向服务端发送消息，你可以看到UDP通信看似可以同时服务于多个客户端，那只是因为他不需要连接服务的速度很快，快到你感知不出来，像这种同时服务于多个客户端快到我们感知不出来的情况，这就叫做并发。如果你把客户端代码复制100万份，使用100万台电脑和100万个人同时向服务端发送消息，那么不同客户端收到消息是一定会有先后顺序的，因为说到底，服务端终究只是一个人在服务，就像我们的TCP通信的服务端一样，客户端少了你自然感知不出来这个时间差，而当客户端多了之后，服务端的服务是一定会有一个先后顺序的，如何解决这个问题呢？就说在服务端造出来多个人来提供服务。假如服务端有100万个人提供服务，同时向服务端发送消息的客户端也要有100万个，那么一个萝卜一个坑，这个就是真正的服务端同时服务于多个客户端了，这就叫做并行。但实际上，假如一开一个饭店，每天同时来的客人有100位，你会找了100为服务员？除非你人傻钱多，否则绝对不会的，我们可以让你个服务员轮流服务客人，只要能够运转开就可以了。同理，服务员就是你的机器的硬件。

3. UDP协议套接字的特点

UDP协议通信一个发送对应一个接收，不会产生粘包问题，因为操作系统监测到你是用数据报协议发送数据会自动定义报头，以此来保证一个发送对应一个接收。
验证UDP协议客户端不粘包
服务端代码

```
import socket
server=socket.socket(socket.AF_INET,socket.SOCK_DGRAM) # 数据报协议
server.bind(('127.0.0.1',8080))
res1,client_addr=server.recvfrom(512)
print(res1)
res2,client_addr=server.recvfrom(512)
```

```
print(res2)
res3,client_addr=server.recvfrom(512)
print(res3)
```

客户端代码

```
import socket
client=socket.socket(socket.AF_INET,socket.SOCK_DGRAM) # 数据报协议
client.sendto(b'hello',('127.0.0.1',8080))
client.sendto(b'world',('127.0.0.1',8080))
client.sendto(b'albert',('127.0.0.1',8080))
```

验证UDP协议服务端不粘包
服务端代码修改如下（客户端代码不变）

```
import socket
server=socket.socket(socket.AF_INET,socket.SOCK_DGRAM) # 数据报协议
server.bind(('127.0.0.1',8080))
res1,client_addr=server.recvfrom(1)  # b'h' 剩下的就不要了，不会粘到下一次接收
print(res1)
res2,client_addr=server.recvfrom(2)  # b'wo'
print(res2)
res3,client_addr=server.recvfrom(3)  # b'alb'
print(res3)
```

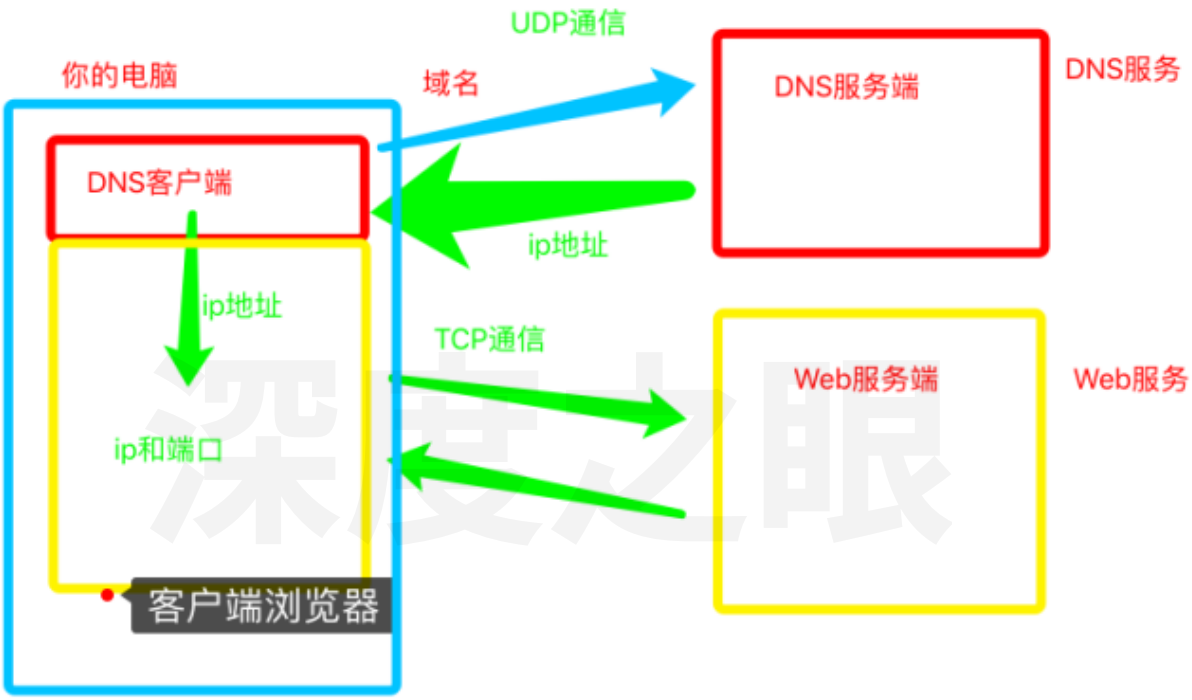
MacOS系统或者Linux系统会看到你想要的结果，Windows系统会报错，其实是干了一件多余的事，就是告诉你接收的参数设置的太小了。UDP协议最大有效传输的是512字节（网上有很多人说是548个字节，我不喜欢这个数字，也不想冒着丢包的风去险挑战UDP的底线），数据再多了就有可能产生丢包的现象。

4. UDP协议套接字应用

我们知道浏览器是一个套接字客户端软件，我们如果需要上网的话，只需要输入被访问的网址链接就可以了。我们以“> https://zhuanlan.zhihu.com/p/54491788“ 这个链接为例，这是一个url地址，叫统一资源定位符，用来定位互联网上独一无二的一个资源，其实就是一份数据或者一个文件。 url地址分为三部分（1）https://（2）"> zhuanlan.zhihu.com"（3）/p/54491788第一部分是应用层的协议，第二部分是域名，第三部分是文件路径，我们浏览器的其中一个功能就是把这个文件下载到本地，给用户观看。浏览

器需要找到服务端主机在哪里，找了服务端主机之后，服务端主机一行一行的读取文件内容，在发送给客户端浏览器，这就是最基本的上过的过程。

那么问题是客户端软件怎么找到服务端软件，自然是通过ip和端口，但是我们现在看到的只有一个域名和服务端主机有关联，域名能帮我们定位到服务端在哪里吗？那么必然会发生的事情就是这个这个域名能够转化的ip和端口，其实这个域名后面省略了一个东西就是“:80”，这个端口指的是服务端的端口，所以我们访问 https://zhuanlan.zhihu.com:80/p/54491788也是一样的。既然已经有了默认的端口，那么必然有一个软件将这个域名解析成一个ip地址，这个软件的客户端是你机器上面自带的，叫做DNS。由于一个域名和一个ip地址这样的数据量远没有达到512个字节，为了是通信更迅速，我们使用UDP协议，这个DNS就是基于UDP协议通信的。



上网通信的原理就是你的客户端浏览器先通过以UDP通信（DNS服务端ip地址可以在你的电脑上查看，端口是默认的53）的套接字DNS服务端建立连接，拿到ip地址，这个DNS服务端叫做DNS服务，这样浏览器客户端有了ip和端口之后，再与Web服务端建立TCP通信，这个Web服务端叫做Web服务。

Web服务有很多种，都是别人写好的，当然你可以可以自己写一个，客户端不用你写，只要写服务端就可以了，常见的Web服务有Nginx，httpd，apache等等。

三 网络通信基本常识

1. 本机获取

想要上网，要做的第一件事就是开机，开机之后你的机器会自动从DHCP服务（也就是向DHCP服务端发送一个请求）中获取以下地址

- 本机的IP地址：192.168.1.100
- 子网掩码：255.255.255.0
- 网关的IP地址：192.168.1.1
- DNS的IP地址：8.8.8.8

2. 打开浏览器

想要访问Google，在地址栏输入了网址：http://www.google.com

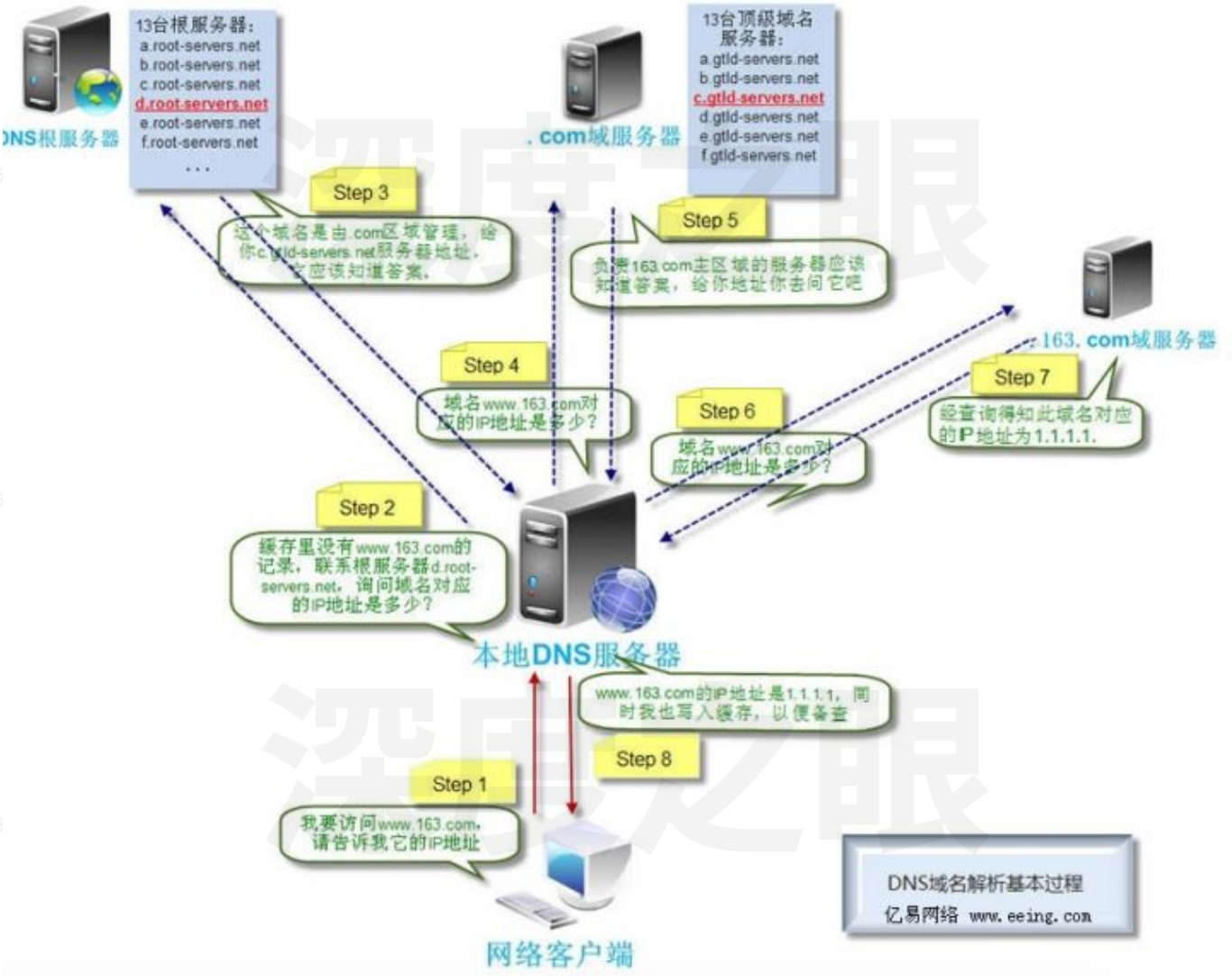
3. DNS协议(基于UDP协议)

DNS服务器要把域名转化成ip地址，忽略政府部门的因素，我们应该可以访问到全世界的网站，我们没有可能把全世界的网站都放到一台机器上，所以每台机器上都有一个本地DNS，查找的时候会先从自己本地DNS查找，如果没有就会去它的同级的DNS查找。我们写的百度的域名是http://www.baidu.com，其实还省略了很多，真正完整的域名是www.baidu.com.root，简写是www.baidu.com.（注意这后面还有一个点），跟域名“.root”对于所有的域名都是一样的，我们经常省略。根域名的下一级，叫做"顶级域名"（top-level domain，缩写为TLD），比如`.com`、`.net`；再下一级叫做"次级域名"（second-level domain，缩写为SLD），比如`www.baidu.com`里面的`.`baidu，这一级域名是用户可以注册的；再下一级是主机名（host），比如`www.baidu.com`里面的`www`，又称为"三级域名"，这是用户在自己的域里面为服务器分配的名称，是用户可以任意分配的。
总结一下，域名的层级结构如下：

```
主机名.次级域名.顶级域名.根域名
# 即
host.sld.tld.root
```

所以排除本地DNS的情况，如果我们想要找到一台机器应该是先问跟域，在按照跟域的指示依次往前询问，但如果真的是这样的话，全世界的机器每天都要都要去跟服务器走一圈，那么无论什么样的硬件配置也的给你干崩了， 所以最好的就是把它保存在本地DNS上，先从本地找，直接就找到了。
刚才说的这些就是本地DNS的情况，除此之外他还有一些同级的姐妹DNS，兄弟DNS等等，如果本地DNS找不到就会去他同级的DNS去寻找，如果找了本地一圈都找不到那就要去找所有域名他爹：跟域名服务器。全球有13台跟域名服务器，跟域名服务器只存储顶级域名的地址， 你想找的域名地址顶级域也不知道，他是他知道谁知道，接下来就是按照域名的顺序依次往左询问查找这台主机的ip地址。这个询问的过程不是由浏览器发起的，而是有DNS发起的，这个查询每一次的查询都是基于上一次的结果来的，是一个

迭代查询的过程。找了半天不容易找到了，他一定会把这个地址存在本地DNS上面，以后就不用这么麻烦的寻找了。



13台根DNS：

A.root-servers.net198.41.0.4美国

B.root-servers.net192.228.79.201美国（另支持IPv6）

C.root-servers.net192.33.4.12法国

D.root-servers.net128.8.10.90美国

E.root-servers.net192.203.230.10美国

F.root-servers.net192.5.5.241美国（另支持IPv6）

G.root-servers.net192.112.36.4美国

H.root-servers.net128.63.2.53美国（另支持IPv6）

I.root-servers.net192.36.148.17瑞典

J.root-servers.net192.58.128.30美国

K.root-servers.net193.0.14.129英国（另支持IPv6）

L.root-servers.net198.32.64.12美国

M.root-servers.net202.12.27.33日本（另支持IPv6）

> 也许你会有这样的想法：为啥中国不自己造一台DNS呀，难道是因为穷吗？这个问题以后千万别问别人，丢人。。。。。。

DNS是基于UDP协议工作的，UDP协议有效传输的数据量是512bytes，这512个字节既要包含你要查询的信息，又要包含这13台跟DNS的地址，13台是一个极限，再多一台UDP协议就不能保证稳定传输了。