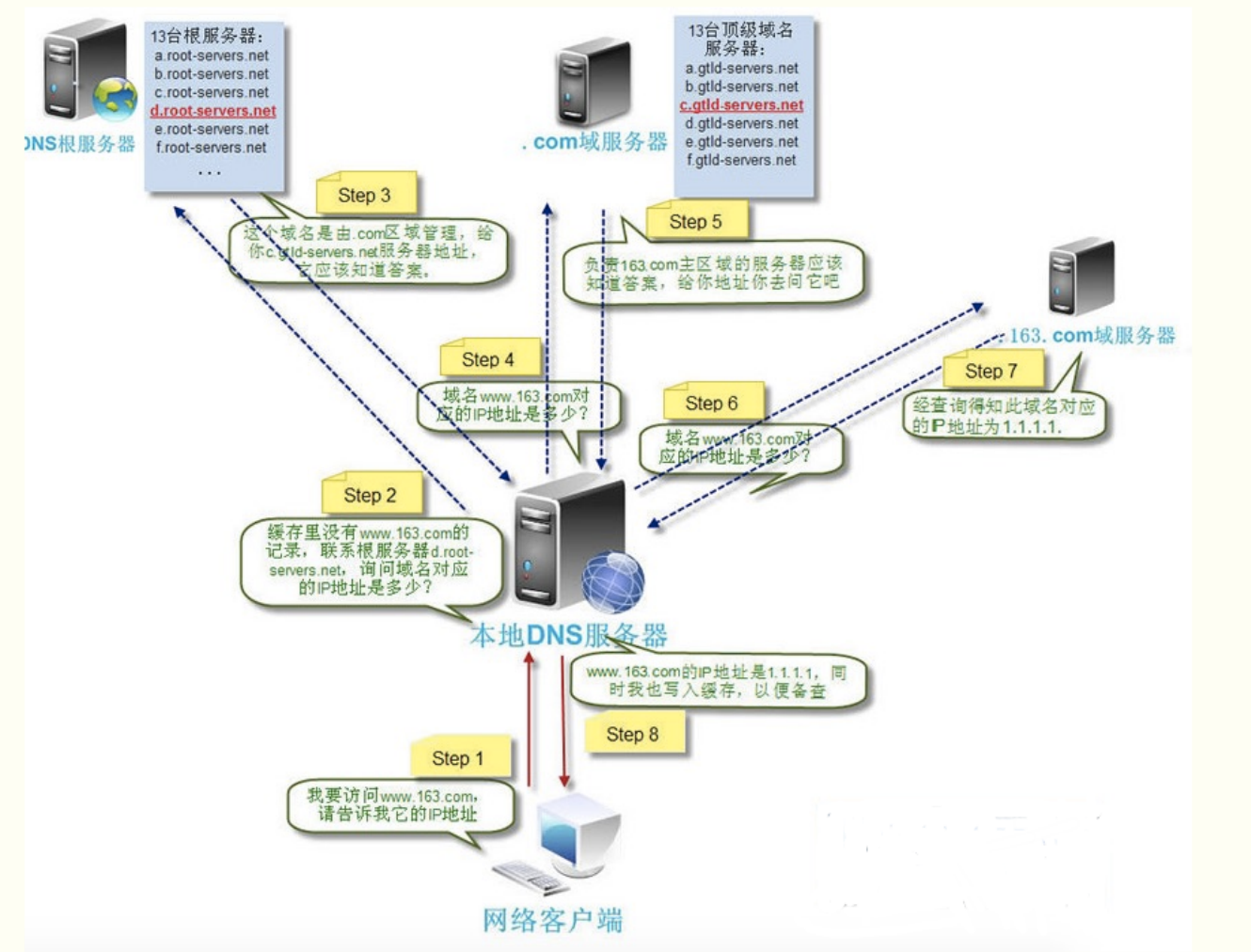
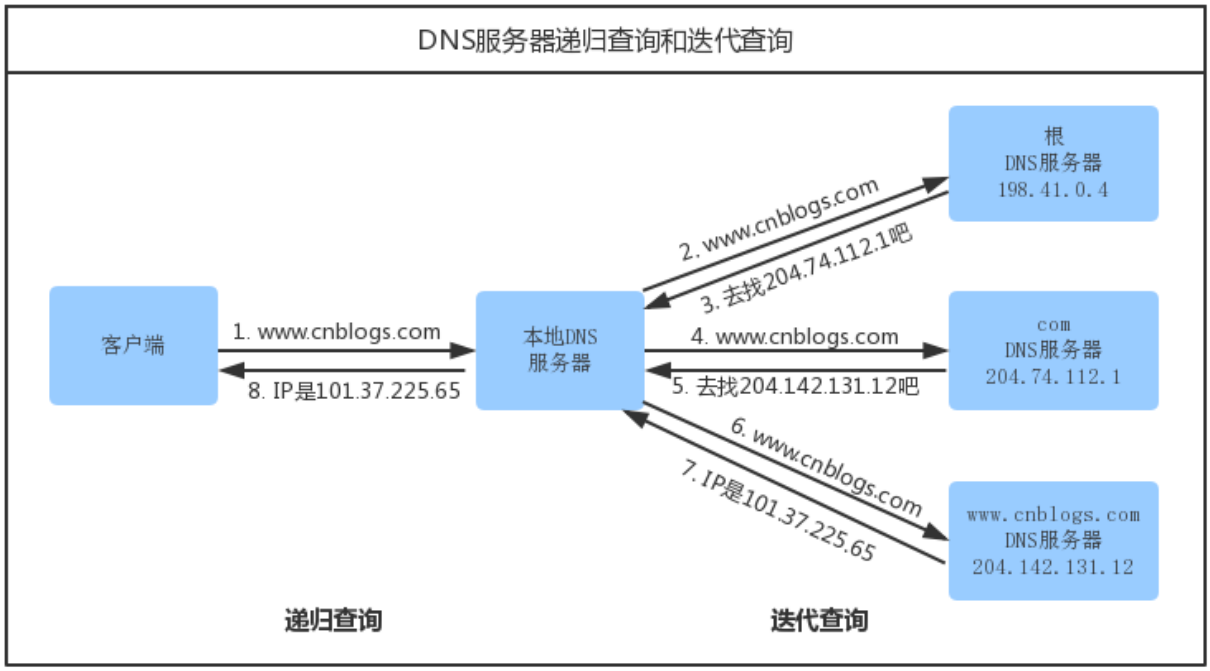
网络编程

一、网络通信的流程

1.DNS域名解析 - > 获取ip地址

- 打开浏览器，想要请求访问京东，在地址栏输入了网址
- 先将请求信息发给了交换机，然后交给了路由器，路由发给DNS服务器，通过DNS协议去找我们要访问的京东的IP地址：

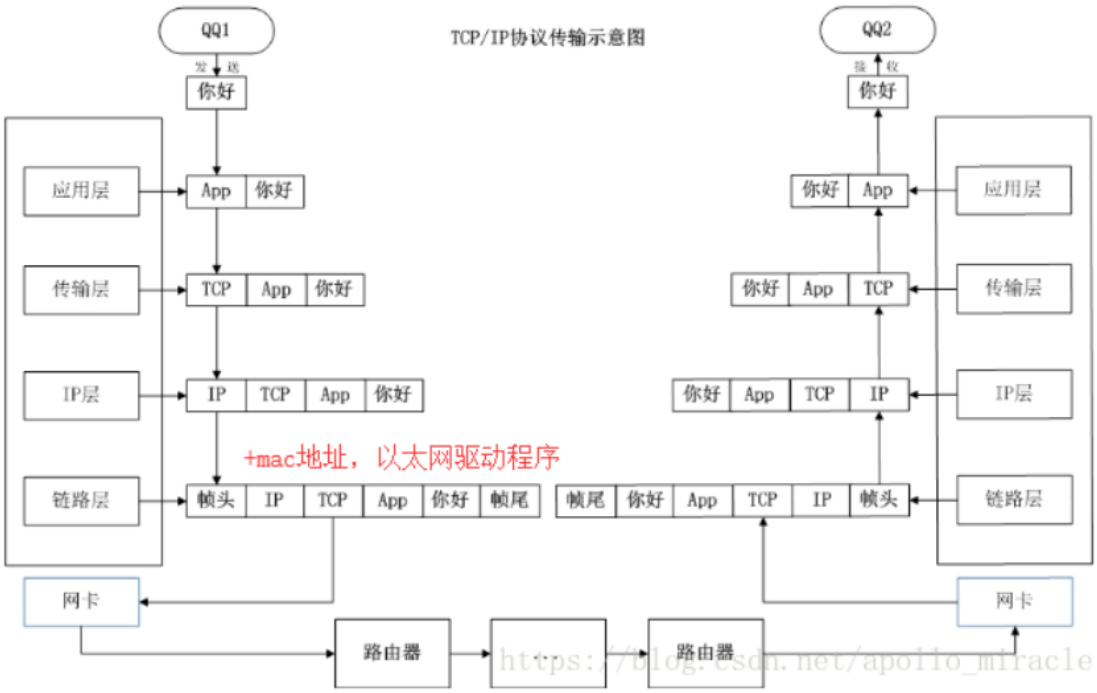




1. 浏览器客户端向本地DNS服务器发送一个含有域名www.cnblogs.com的DNS查询报文。
2. 本地DNS服务器把查询报文转发到根DNS服务器，根DNS服务器注意到其com后缀，于是向本地DNS服务器返回comDNS服务器的IP地址。
3. 本地DNS服务器再次向comDNS服务器发送查询请求，comDNS服务器注意到其www.cnblogs.com后缀并用负责该域名的权威DNS服务器的IP地址作为回应。
4. 最后，本地DNS服务器将含有www.cnblogs.com的IP地址的响应报文发送给客户端

2.建立TCP连接

- 拿到域名对应的IP地址之后，**User-Agent**（一般是指浏览器）会以一个随机端口（1024 < 端口 < 65535）向服务器的WEB程序（常用的有httpd,nginx等）80端口发起TCP的**连接请求**。
- 这个连接请求（原始的http请求经过TCP/IP4层模型的层层封包）到达服务器端后（这中间通过各种路由设备，局域网内除外），进入到网卡，然后是进入到内核的TCP/IP协议栈（用于识别该连接请求，解封包，一层一层的剥开），还有可能要经过Netfilter防火墙（属于内核的模块）的过滤，最终到达WEB程序，最终建立了TCP/IP的连接。



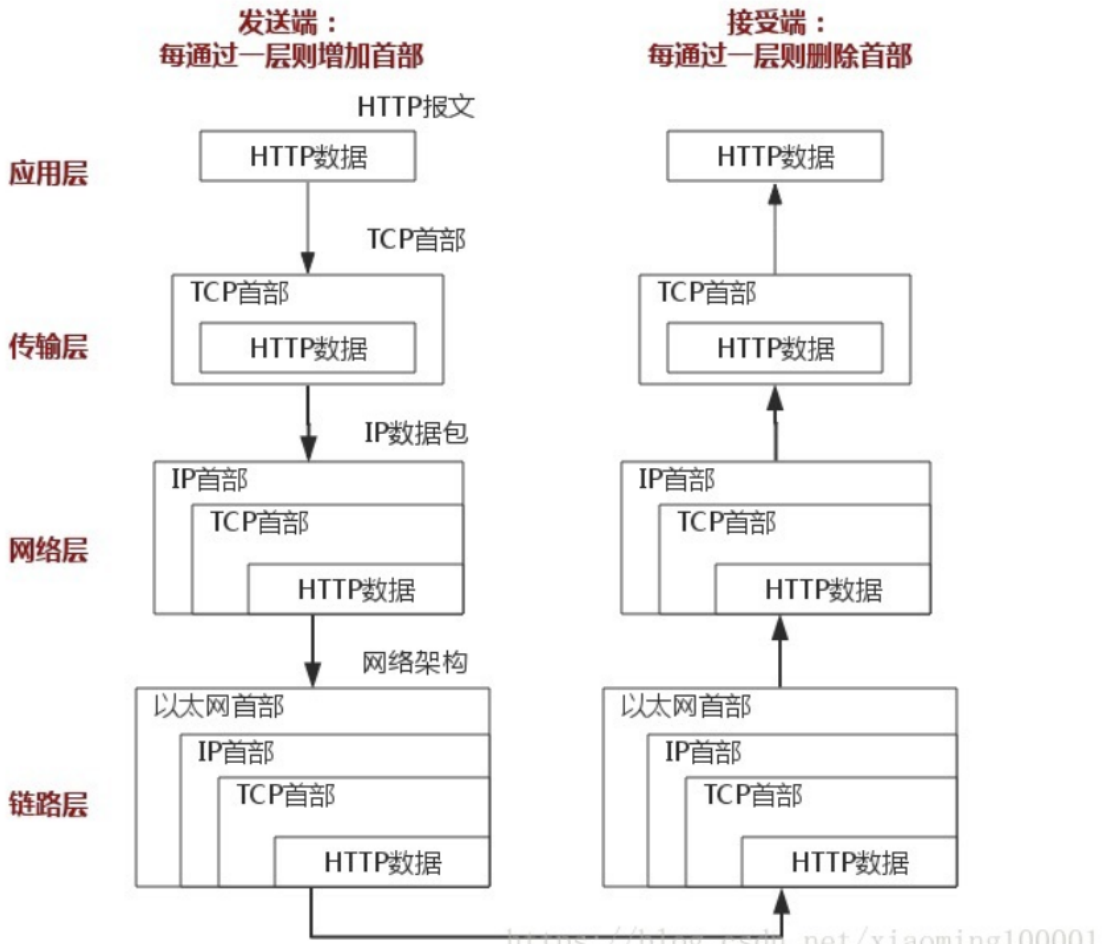
- TCP三次握手

  ```
  Client首先发送一个连接试探
  Server监听到连接请求报文后，如同意建立连接，则向Client发送确认
  Client收到确认后还需再次发送确认，同时携带要发送给Server的数据
  ```

- 目前在Internet中所有的传输都是通过TCP/IP进行的，**HTTP协议作为TCP/IP模型中应用层的协议也不例外，TCP是一个端到端的可靠的面向连接的协议**，所以HTTP基于传输层TCP协议不用担心数据的传输的各种问题
- **网络层IP协议查询MAC地址**

  IP协议的作用是把TCP分割好的各种数据包传送给接收方。而要保证确实能传到接收方还需要接收方的MAC地址，也就是物理地址。IP地址和MAC地址是一一对应的关系，一个网络设备的IP地址可以更换，但是MAC地址一般是固定不变的。ARP协议可以将IP地址解析成对应的MAC地址。当通信的双方不在同一个局域网时，需要多次中转才能到达最终的目标，在中转的过程中需要通过下一个中转站的MAC地址来搜索下一个中转目标。
- 在找到对方的MAC地址后，就将数据发送到数据链路层传输。这时，客户端发送请求的阶段**结束**

3. 建立TCP连接后发起HTTP请求



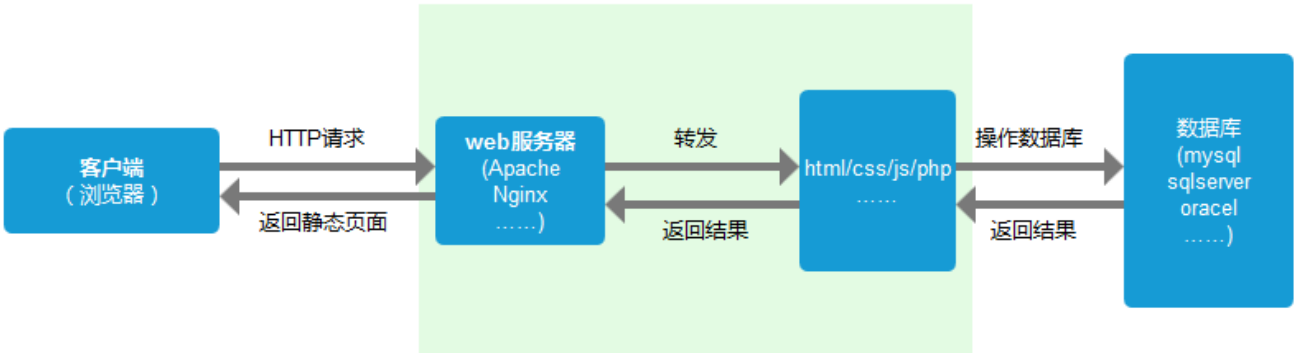
4.服务端处理请求

1.数据交换主要通过 http 协议

2.服务器 server 模型

3.server 处理流程

web服务器解析用户请求，知道了需要调度哪些资源文件，再通过相应的这些资源文件处理用户请求和参数，并调用数据库信息，最后将结果通过web服务器返回给浏览器客户端。



- server 这边 Nginx 拿到请求，进行一些验证，比如黑名单拦截之类的，然后 Nginx 直接处理静态资源请求，其他请求 Nginx 转发给后端服务器
- 这里我用 uWSGI, 他们之间通过 uwsgi 协议通讯，uWSGI 拿到请求，可以进行一些逻辑， 验证黑名单、判断爬虫等，根据 wsgi 标准，把拿到的 environs 参数扔给 Django

- Django 根据 wsgi 标准接收请求和 env， 然后开始 start_response
- Django 拿到请求执行 request middleware 内的相关逻辑，然后路由到相应 view 执行逻辑，出错执行 exception middleware 相关逻辑，接着 response 前执行 response middleware 逻辑，最后通过 wsgi 标准构造 response
- 拿到需要返回的东西，设置一些 headers，或者 cookies 之类的，最后 finish_response 返回，再通过 uWSGI 给 Nginx ， Nginx 返回给浏览器。

5. 返回响应结果

6.关闭TCP连接的请求 - 四次握手

```
四次挥手:
    1.客户端向服务端发出要断开连接的请求
    2.服务端收到请求后,向服务端发出等待的消息,清理通信的残余信息
    3.服务端向处于等待状态的客户端发送可以断开连接的信息
    4.客户端收到服务端信息后,向服务端发送断开连接确认
```

7.浏览器解析HTML，渲染页面

二、TCP、IP协议

1.python 底层的网络交互模块有哪些?

2. 简述 OSI 七层协议

```
应用层: 为用户的应用程序提供网络服务,规定应用程序的数据格式
表示层: 对来自应用层的命令和数据进行解析，按照一定格式传给会话层。如编码、数据格式转换、加密解密、 压缩解压
会话层: 建立客户端与服务端连接,主要在你的系统之间发起会话或者接受会话请求（设备之间需要互相认识 可以是IP也可以是MAC或者是主机名）
传输层: 定义了一些传输数据的协议和端口号（www端口80等),主要是将从下层接收的数据进行分段和传输， 到达目的地址后再进行重组。建立端口到端口的通信(端对端通信） TCP|UDP
网络层: 引入一套新的地址用来区分不同的广播域/子网，这套地址即网络地址
数据链路层: 定义了电信号的分组方式
物理层: 主要是基于电器特性发送高低电压（电信号），高电压对应数字1，低电压对应数字0
```

3.什么是C/S和B/S架构?

```
c/s架构，就是client（客户端）与server（服务端）即：客户端与服务端的架构。
b/s架构，就是brosver（浏览器端）与sever（服务端）即：浏览器端与服务端架构
优点：统一了所有应用程序的入口、方便、轻量级
```

4..简述三次握手,四次挥手流程

三次握手：

1.客户端向服务端发起建立连接的请求
2.服务端收到请求后，如果同意则返回确认信息
3.客户端收到服务端可以连接的确认后，向服务端发送确认连接

四次挥手：

1.客户端向服务端发出要断开连接的请求
2.服务端收到请求后，向服务端发出等待的消息，清理通信的残余信息
3.服务端向处于等待状态的客户端发送可以断开连接的信息
4.客户端收到服务端信息后，向服务端发送断开连接确认

5.TCP 的三次握手与四次挥手过程，各个状态名称与含义， TIMEWAIT 的作用。 TCP 的三次握手过程？为什么会采用三次握手，若采用二次握手可以吗？

三次握手过程

- 假设A是客户端，B是服务端。A首先向B发出连接请求报文段，这个时候首部中的同步位SYN=1，同时选择一个初始的序号x。此时报文段不能携带数据。此时A进入到SYN_SENT(同步已发送)状态。
- B受到连接请求报文，同意建立连接，向A发出确认。确认报文中，SYN和ACK都置1，确认号是x+1,与此同时，自己选择一个初始序号y,这个报文也不能携带数据。此时B进入SYN_RCVD（同步收到）状态。
- A收到B的确认后，还要给B确认。这时可以携带数据，A进入到ESTABLISHED状态。这就是三次握手的过程。

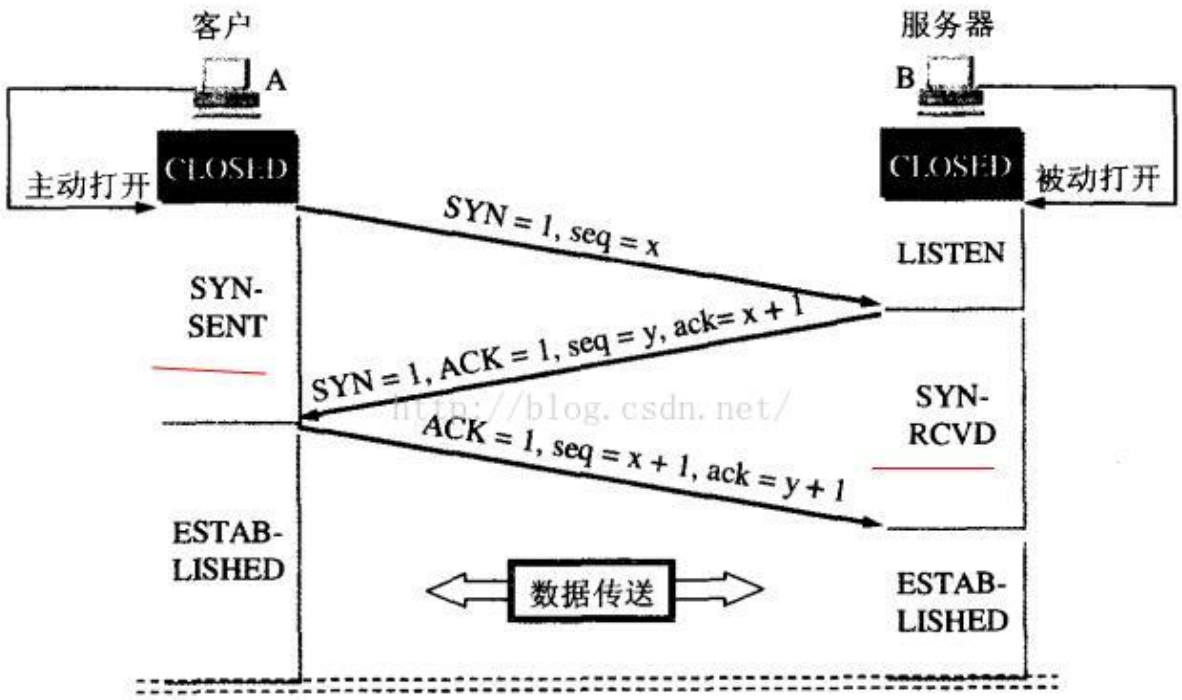


图 5-31　用三次握手建立 TCP 连接

A为什么还要发送一次确认

- 为了防止已经失效的连接请求报文又突然传送到了B,而产生错误。
- 假设一种异常，A发出的请求由于网络阻塞没有及时到达B，后又重传请求，之后B响应了，且建立了连接，之后连接又释放了。
- 此时假设A发出的第一个请求到达B,B误以为是A再次请求连接，B建立连接，如果采用两次握手，此时连接建立，而A又不发送数据，浪费了B的资源。

四次挥手

- 数据传输结束后，通信双方都可以释放连接。
- 现在A和B都处于ESTABLISHED状态，A的应用进程向其TCP发出连接释放报文段，主动关闭TCP连接。**A进入FIN_WAIT1（终止等待1）状态。**
- 然后B确认，**B进入CLOSE_WAIT(关闭等待)状态。**此时TCP处于半关闭状态，A已经没有数据要发送了，如果B仍要发送数据，B仍然接收。
- **A收到B的确认后，就进入FIN_WAIT2（终止等待2）状态**，等待B发出连接释放报文。
- 如果B已经没有向A发送的数据，则B发送请求释放报文，**B进入LAST_ACK（最后确认）阶段**，等待A的确认。
- **A在收到B的请求后，要发出确认**，然后进入TIME_WAIT（时间等待）状态。此时，连接还未释放，必须等待时间等待计时器设定的时间的2MSL后，A才进入CLOSED状态。

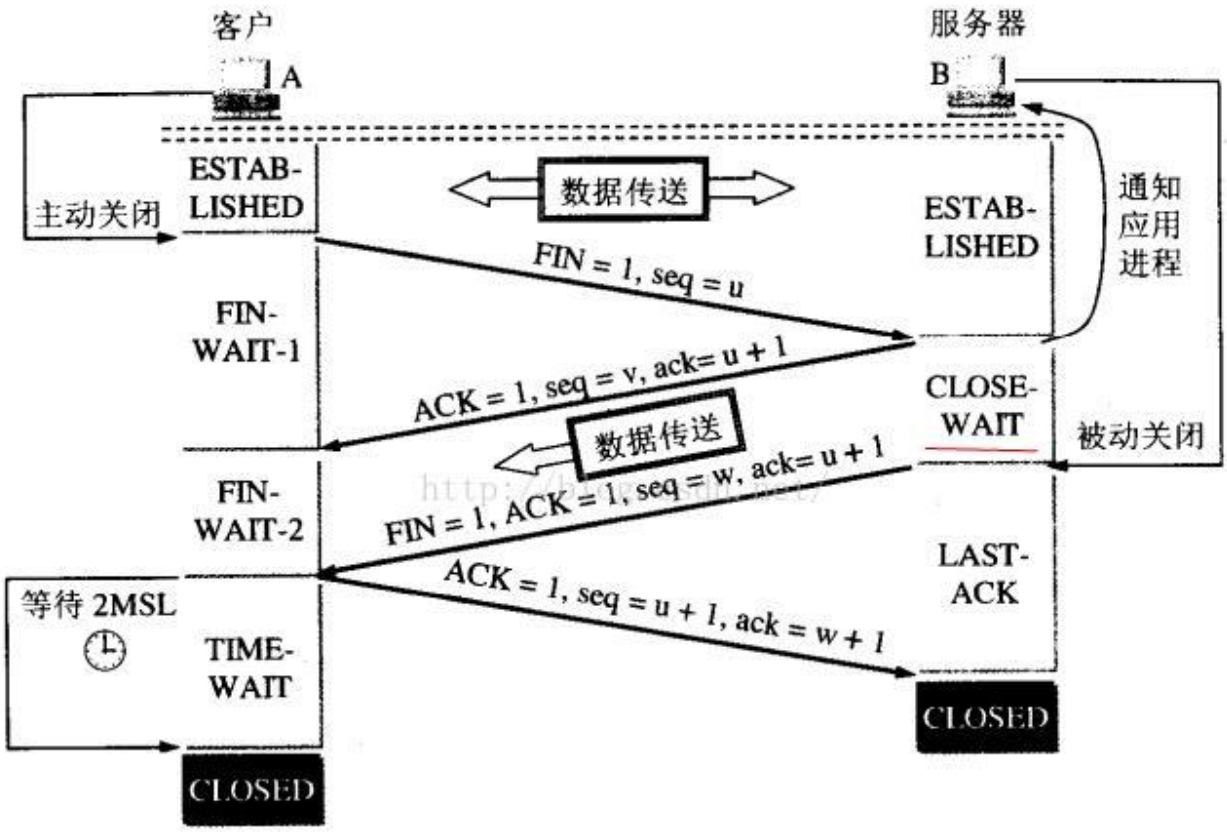
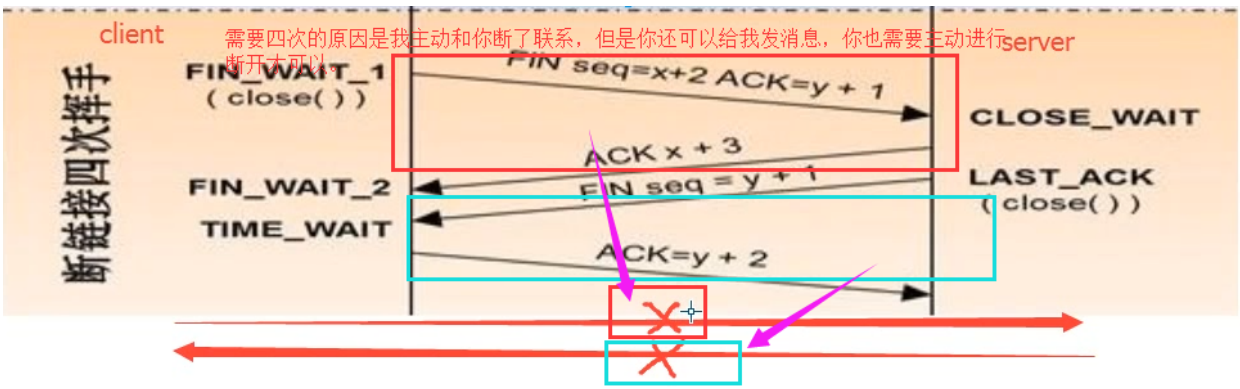


图 5-32　TCP 连接释放的过程

为什么最后要等一个TIME_WAIT时间呢?

- 为了保证最后一个ACK能够到达B,防止丢失了，B重传，A不能回复确认。
- 为了防止之前提到的“已经失效的连接请求报文段“出现在连接中”。A发送完最后一个ACK，再经过时间2MSL，可以使本连接产生的所有请求报文从网络中消失
- 

6. 什么是arp协议

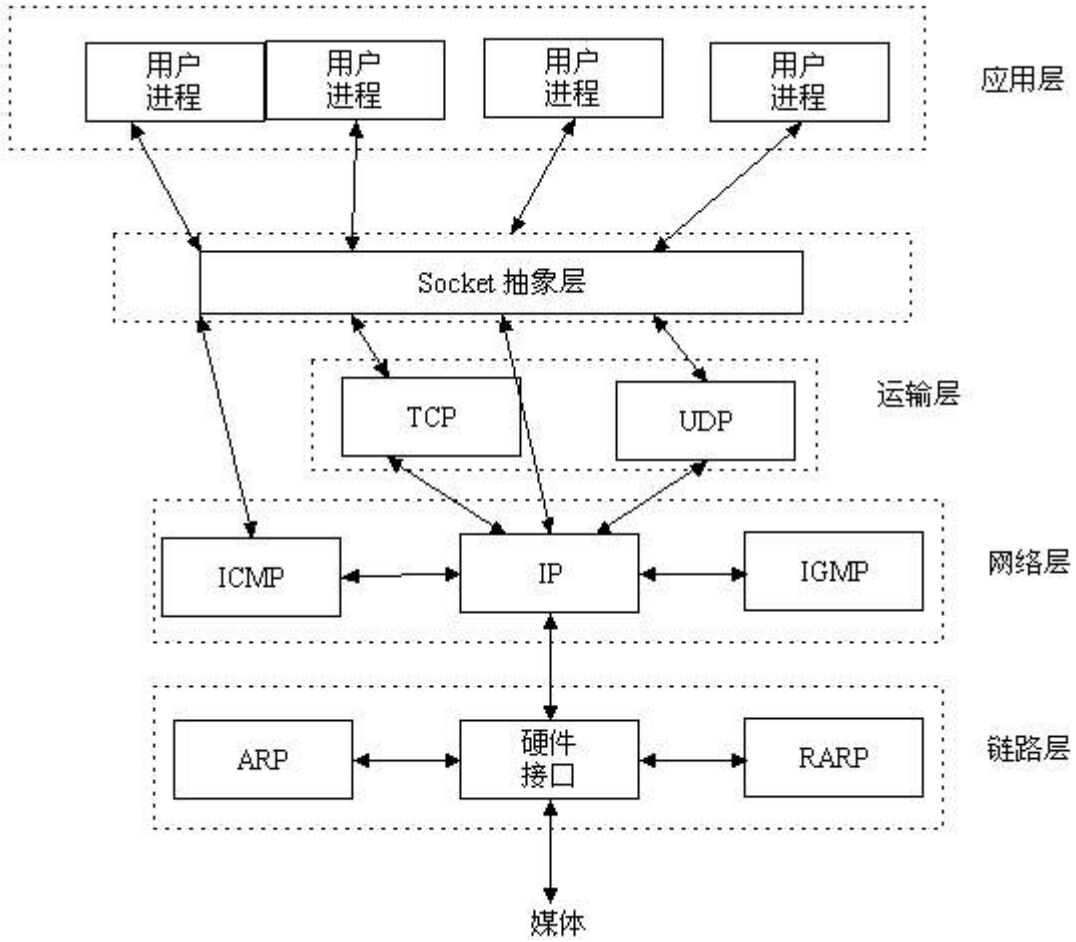
是根据IP地址获取物理地址的一个TCP/IP协议，主机发送信息时将包含目标IP地址的ARP请求广播到网络上的所有主机，并接收返回消息，以此确定目标的物理地址。

arp协议功能：广播的方式发送数据包，获取目标主机的mac地址

7.TCP和UDP的区别？为何基于tcp协议的通信比基于udp协议的通信更可靠?

TCP和UDP的区别：
    tcp协议：面向连接，消息可靠，相对udp来讲，传输速度慢，消息是面向流的，无消息保护边界
    udp协议：面向无连接，消息不可靠，传输速度快，消息是面向包的，有消息保护边
传输可靠性：
TCP：是面向连接、可靠的字节流服务，因为只要对方回了确认收到信息，才发下一个，如果没收到确认信息就重发
UDP：数据报协议，只是将消息发送出去，并不能保证它们能到达目的地，且没有超时重发等机制

8.什么是socket？简述基于TCP | UDP 协议的套接字通信流程



socket(套接字)：
    socket是在应用层和传输层之间的一个抽象层，它把TCP/IP层复杂的操作抽象为几个简单的接口供应用层调用来实现进程在网络中通信。

```
'''-----------TCP协议的套接字通信流程 ------------
1.服务器端先初始化Socket，然后与端口绑定(bind)，对端口进行监听(listen)，调用accept阻塞，等待客户端连接。
```

```
2.在这时如果有个客户端初始化一个Socket，然后连接服务器(connect)，如果连接成功，这时客户端与服务器端的连接就建立了。
3.客户端发送数据请求，服务器端接收请求并处理请求，然后把回应数据发送给客户端，客户端读取数据，最后关闭连接，一次交互结束
'''
#-------server ------
import socket
sk = socket.socket()
sk.bind(('127.0.0.1',8898))  #把地址绑定到套接字
sk.listen()           #监听链接
conn,addr = sk.accept() #接受客户端链接
ret = conn.recv(1024)  #接收客户端信息
print(ret)        #打印客户端信息
conn.send(b'hi')         #向客户端发送信息
conn.close()        #关闭客户端套接字
sk.close()         #关闭服务器套接字(可选)


#------client -------
import socket
sk = socket.socket()            # 创建客户套接字
sk.connect(('127.0.0.1',8898))     # 尝试连接服务器
sk.send(b'hello!')
ret = sk.recv(1024)          # 对话(发送/接收)
print(ret)
sk.close()             # 关闭客户套接字
```

```
# -----udp 协议的socket -----

# ---server -----
import socket
#创建一个udp协议下的socket,需要使用参数type
udp_server = socket.socket(type = socket.SOCK_DGRAM)      #DGRAM:数据报
#拿到一个地址,启动程序时候,高速电脑,给这个程序分配8888端口
ip_port = ("192.168.15.51",8888)
udp_server.bind(ip_port)

print("可以接受消息了.....")

#可以接受消息
from_udp_client_msg ,client_addr = udp_server.recvfrom(1024)
print(11111)
print(from_udp_client_msg.decode("utf-8"))
print(client_addr)

udp_server.sendto("hehe".encode("utf-8"),client_addr)
udp_server.close()

# -----client ------
import socket
udp_client = socket.socket(type = socket.SOCK_DGRAM)
udp_server_ip_port =("192.168.15.51",8888)
udp_client.connect(udp_server_ip_port)
```

```
#可以发消息了
udp_client.sendto("哈哈".encode("utf-8"),udp_server_ip_port)

from_udp_server_msg,server_addr = udp_client.recvfrom(1024)
print (from_udp_server_msg.decode("utf-8"))
print(server_addr)

udp_client.close
```

9.什么是粘包？ socket 中造成粘包的原因是什么？ 哪些情况会发生粘包现象？粘包怎么解决?

```
------什么是粘包-------
粘包现象分两种:
1.连续发送小的数据,间隔时间很短,有可能一次就接收到了这几个连续的拼接在一起的小数据.
2.当你一次接收的数据长度小于你一次发送的数据长度,那么一次接受完剩下的数据会在下一次接收数据的时候被一起接收

------造成粘包的原因----:
    所谓粘包问题主要还是因为接收方不知道消息之间的界限，不知道一次性提取多少字节的数据所造成的
    即:两端互相不知道对方发送数据的长度

---发生粘包的情况---
1、发送端需要等缓冲区满才发送出去，造成粘包（发送数据时间间隔很短，数据了很小，会合到一起，产生粘包）
2、接收方不及时接收缓冲区的包，造成多个包接收（客户端发送了一段数据，服务端只收了一小部分，
   服务端下次再收的时候还是从缓冲区拿上次遗留的数据，产生粘包
```

```
#------粘包的解决方案 ------------
解决方案一：将要发送的字节流总大小让接收端知晓，然后接收端发一个确认消息给发送端，然后发送端再发送过来后面
        的真实内容，接收端再来一个死循环接收完所有数据。
解决方案二:
        通过struck模块将需要发送的内容的长度进行打包，打包成一个4字节长度的数据发送到接收端,再发送真实内
容,
        接收端取出前4个字节，然后对这四个字节的数据进行解包，拿到发送的内容的长度
        然后通过这个长度来循环继续接收我们实际要发送的内容
```

```
# ------ server ----------
import socket
import subprocess
import struct
server = socket.socket()
ip_port = ("192.168.15.51",8002)
server.bind(ip_port)
server.listen(3)
while 1:
    conn,addr = server.accept()
    flag=0
    while not flag:
        #来自客户端的指令
        from_client_cmd = conn.recv(1024).decode("utf-8")
        print(from_client_cmd)
```

```
        sub_obj =
subprocess.Popen(from_client_cmd,shell=True,stderr=subprocess.PIPE,stdout=subprocess.PIPE)
        server_cmd_msg = sub_obj.stdout.read()
        # server_cmd_err = sub_obj.stderr.read()
        #首先计算出你将要发送的数据的长度
        cmd_msg_len = len(server_cmd_msg)
        #先对数据长度进行打包,打包成4个字节的数据,目的是为了和你将要发送的数据拼接在一起
        msg_len_stru = struct.pack("i",cmd_msg_len)
        conn.send(msg_len_stru)  #首先发送打包成功后的那4个字节的数据
        conn.sendall(server_cmd_msg)  #循环send数据,知道数据全部发送成功


    conn.close()



# ------ client ------
import socket
import struct
client = socket.socket()
server_ip_port = ip_port = ("192.168.15.51",8002)
client.connect(server_ip_port)

flag = 0
while not flag:
    cmd = input("请输入要执行的指令>>>")
    client.send(cmd.encode("utf-8"))
    #先接收服务端要发送给我的信息长度,前四个字节,固定的
    from_server_msglen = client.recv(4)
    unpack_len_msg = struct.unpack("i",from_server_msglen)[0]
    #接收数据长度统计,和服务端发给我的数据长度作比较,来确定跳出循环的条件
    recv_msg_len = 0
    #统计拼接接收到的数据,注意这个不是统计长度
    all_msg =b''
    while recv_msg_len < unpack_len_msg:
        every_recv_data = client.recv(1024)
        #将每次接收到的数据进行拼接和统计
        all_msg += every_recv_data
        #对每次接收到的数据的长度进行累加
        recv_msg_len +=len(every_recv_data)
    print(all_msg.decode("gbk"))

client.close()
```