python-网络

讲师：尚玉杰

目录

网络基础

- 如何在网络中唯一标识一台计算机？ ip地址
- 同一台计算机上的多个程序如何共用网络而不冲突？ 网络端口
- 不同的计算机通信怎么才能互相理解？ 使用相同的协议

网络基础-IP地址

- IP地址： 用来在网络中标记一台电脑的一串数字， 比如192.168.1.1（c类）； 在同一网络上是惟一的（用来标记唯一的一台电脑）
- 每一个IP地址包括两部分： 网络地址和主机地址
- 主机号0,255两个数不能使用（网络号、广播地址）
-

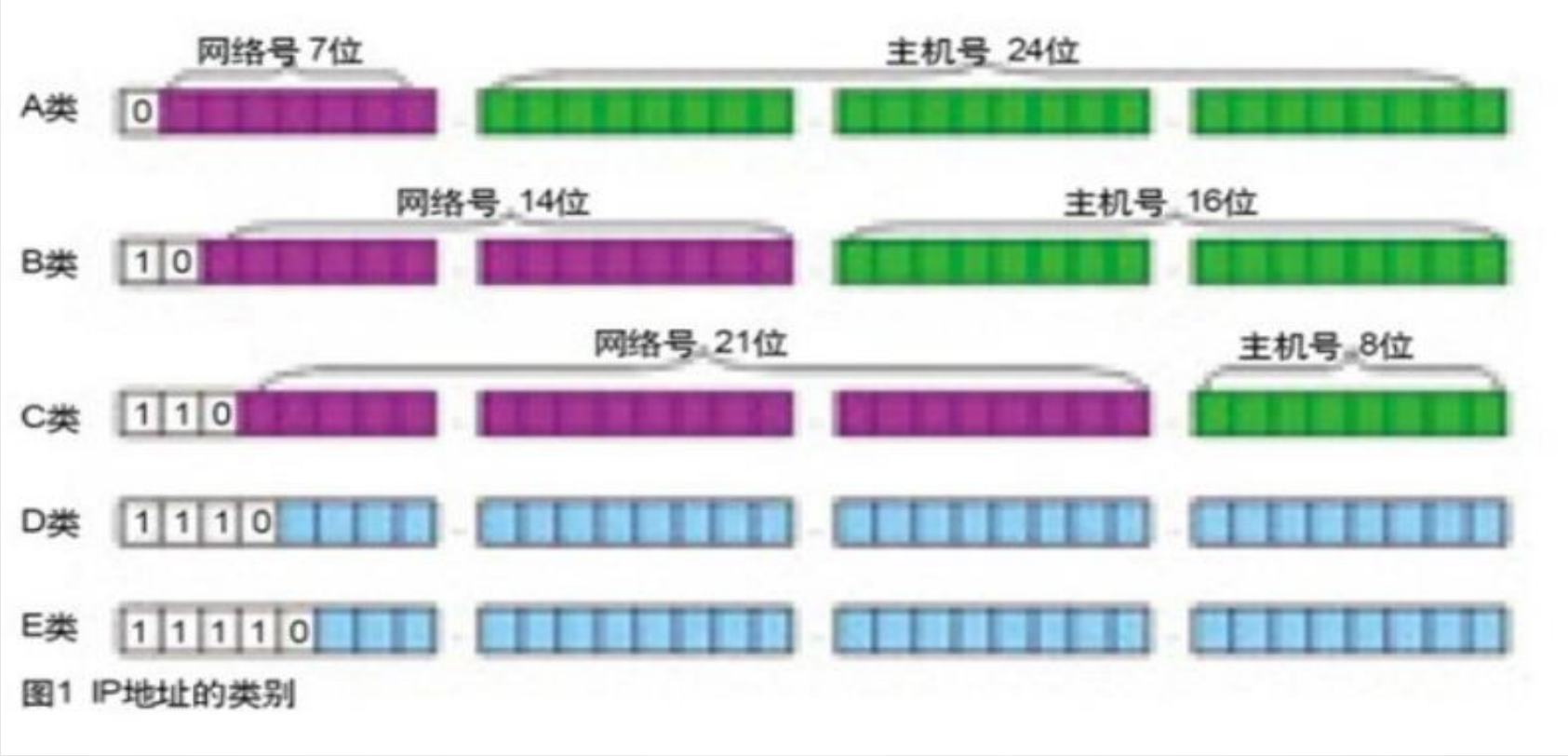


图1 IP地址的类别

网络基础-IP地址

- A类IP地址由1字节的网络地址和3字节主机地址组成， 网络地址的最高位必须是“0”，地址范围1.0.0.1-126.255.255.254可用的A类网络有126个， 每个网络能容纳1677214个主机
- B类IP地址由2个字节的网络地址和2个字节的主机地址组成， 网络地址的最高位必须是“10”，地址范围128.1.0.1-191.255.255.254 可用的B类网络有16384个，每个网络能容纳65534主机
- C类IP地址由3字节的网络地址和1字节的主机地址组成， 网络地址的最高位必须是“110”范围192.0.1.1-223.255.255.254 C类网络可达2097152个， 每个网络能容纳254个主机
- D类IP地址第一个字节以“1110”开始， 它是一个专门保留的地址。它并不指向特定的网络， 目前这一类地址被用在多点广播（一对多） 中多点广播地址用来一次寻址一组计算机 地址范围224.0.0.1-239.255.255.254
- E类IP地址以“1111”开始， 为将来使用保留 E类地址保留， 仅作实验和开发用

网络基础-IP地址

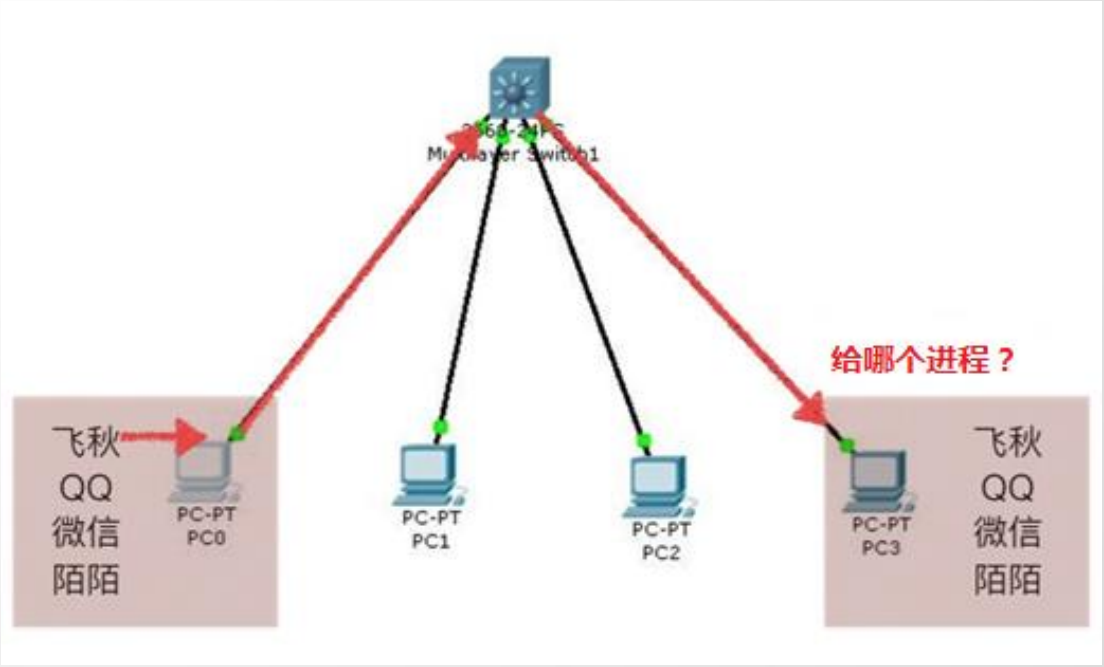
- 私有IP：**本地局域网上的IP**，专门为组织机构内部使用
- 在这么多网络IP中， 国际规定有一部分IP地址是用于我们的局域网使用， 属于私网IP， 不在公网中使用的， 它们的范围是：
  - 10.0.0.0～10.255.255.255
  - 172.16.0.0～172.31.255.255
  - 192.168.0.0～192.168.255.255
  - 私有IP：局域网通信，内部访问，不能在外网公用。私有IP禁止出现在Internet中，来自于私有IP的流量全部都会阻止并丢掉
  - 公有IP：全球访问
- IP地址127． 0． 0． 1用于回路测试
  - 测试当前计算机的网络通信协议
  - 如： 127.0.0.1可以代表本机IP地址， 用 http://127.0.0.1 就可以测试本机中配置的Web服务器
  - 常用来ping 127.0.0.1来看本地ip/tcp正不正常，如能ping通即可正常使用

网络基础-IP地址

- 子网掩码：是我们测量两个IP是否属于同一个网段的工具
  - 子网掩码不能单独存在， 它必须结合IP地址一起使用
  - 子网掩码只有一个作用， 就是将某个IP地址划分成网络地址和主机地址两部分
  - 子网掩码的设定必须遵循一定的规则：
    与IP地址相同， 子网掩码的长度也是32位，
    左边是网络位， 用二进制数字“1”表示；
    右边是主机位， 用二进制数字“0”表示
  - 假设IP地址为“192.168.1.1”子网掩码为“255.255.255.0”。
    其中， “1”有24个， 代表与此相对应的IP地址左边24位是网络号；
    “0”有8个， 代表与此相对应的IP地址右边8位是主机号

网络基础-端口号

- 端口号： 用来标记区分进程
- 一台拥有IP地址的主机可以提供许多服务， 比如HTTP（万维网服务） 、 FTP（文件传输） 、 SMTP（电子邮件） 等， 这些服务完全可以通过1个IP地址来实现。那么， 主机是怎样区分不同的网络服务呢？
- 显然不能只靠IP地址， 因为IP地址与网络服务的关系是一对多的关系。实际上是通过“IP地址+端口号”来区分不同的服务的。



网络基础-端口号

- 端口号是一个数字，只有整数， 范围是从0到65535 （分为知名和动态两种）
- 知名端口是众所周知的端口号（用来做固定事情）， 范围从0到1023
  - 80端口分配给HTTP服务（网站）
  - 21端口分配给FTP服务（文件下载）
  - 可以理解为， 一些常用的功能使用的号码是固定的
- 动态端口的范围是从1024到65535
  之所以称为动态端口， 是因为它一般不固定分配某种服务， 而是动态分配。
  动态分配是指当一个系统进程或应用程序进程需要网络通信时， 它向主机申请一个端口， 主机从可用的端口号中分配一个供它使用

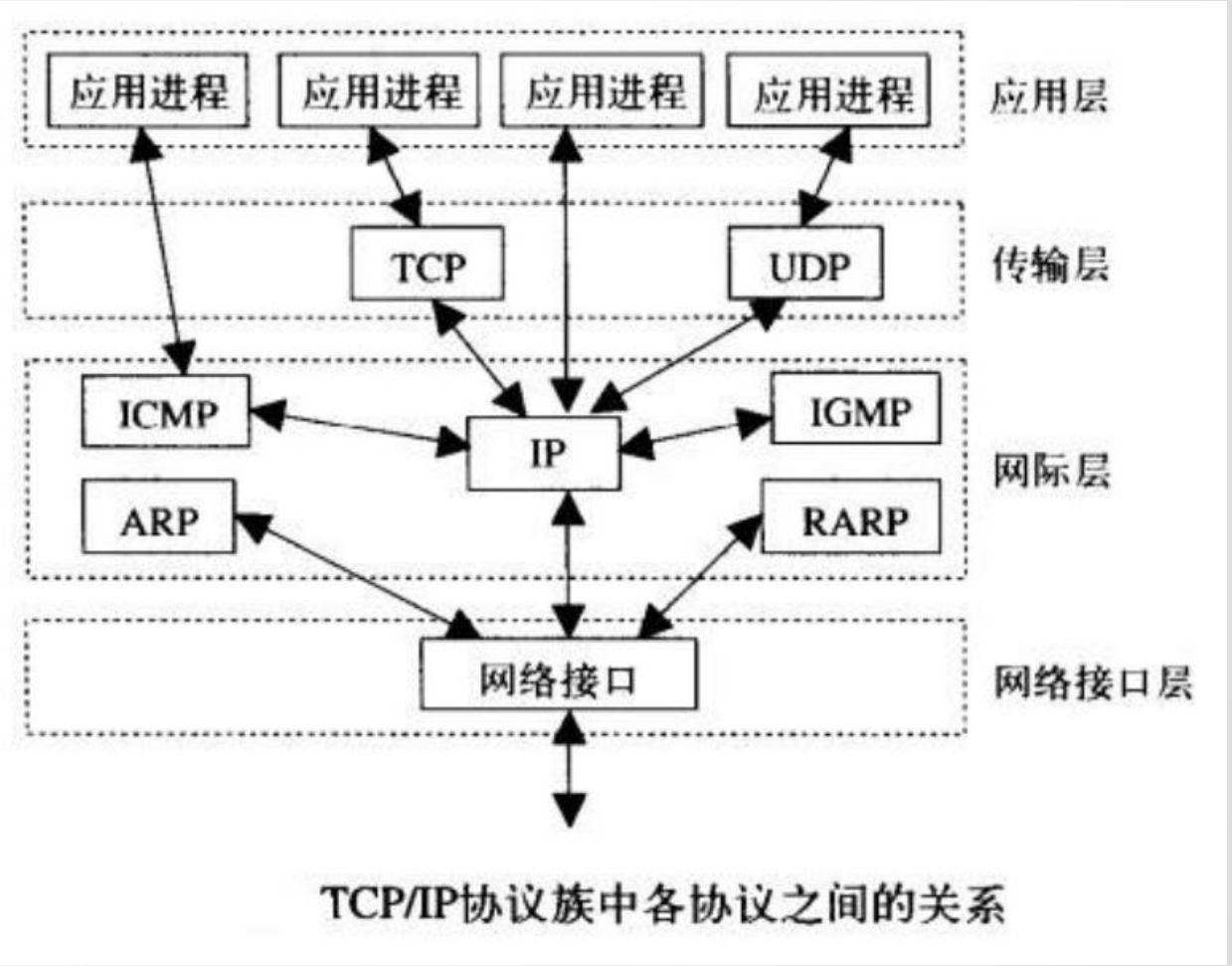
网络基础-协议

- 协议：约定好的规范
- 早期的计算机网络， 都是由各厂商自己规定一套协议， IBM、 Apple和Microsoft都有各自的网络协议， 互不兼容（语言、方言、阿帕网）
  为了把全世界的所有不同类型的计算机都连接起来， 就必须规定一套全球通用的协议， 为了实现互联网这个目标， 互联网协议簇（Internet Protocol Suite） 就是通用协议标准。
  因为互联网协议包含了上百种协议标准， 但是最重要的两个协议是TCP和IP协议， 所以， 大家把互联网的协议总称TCP/IP协议
  （大家都遵循的最基本网络通信协议）
- 是完成进程之间通信的规范

网络基础-协议

- 根据TCP/IP协议簇功能的不同，将它分为了几种层次（ TCP/IP协议簇层次划分）（重点记住）
  - 网络接口层（链路层）
  - 网络层
  - 传输层
  - 应用层
  - （写代码按四层划分）

  - 物理层
  - 数据链路层
  - 网络层
  - 传输层
  - 会话层
  - 表示层
  - 应用层
  - （理论上由七层组成）

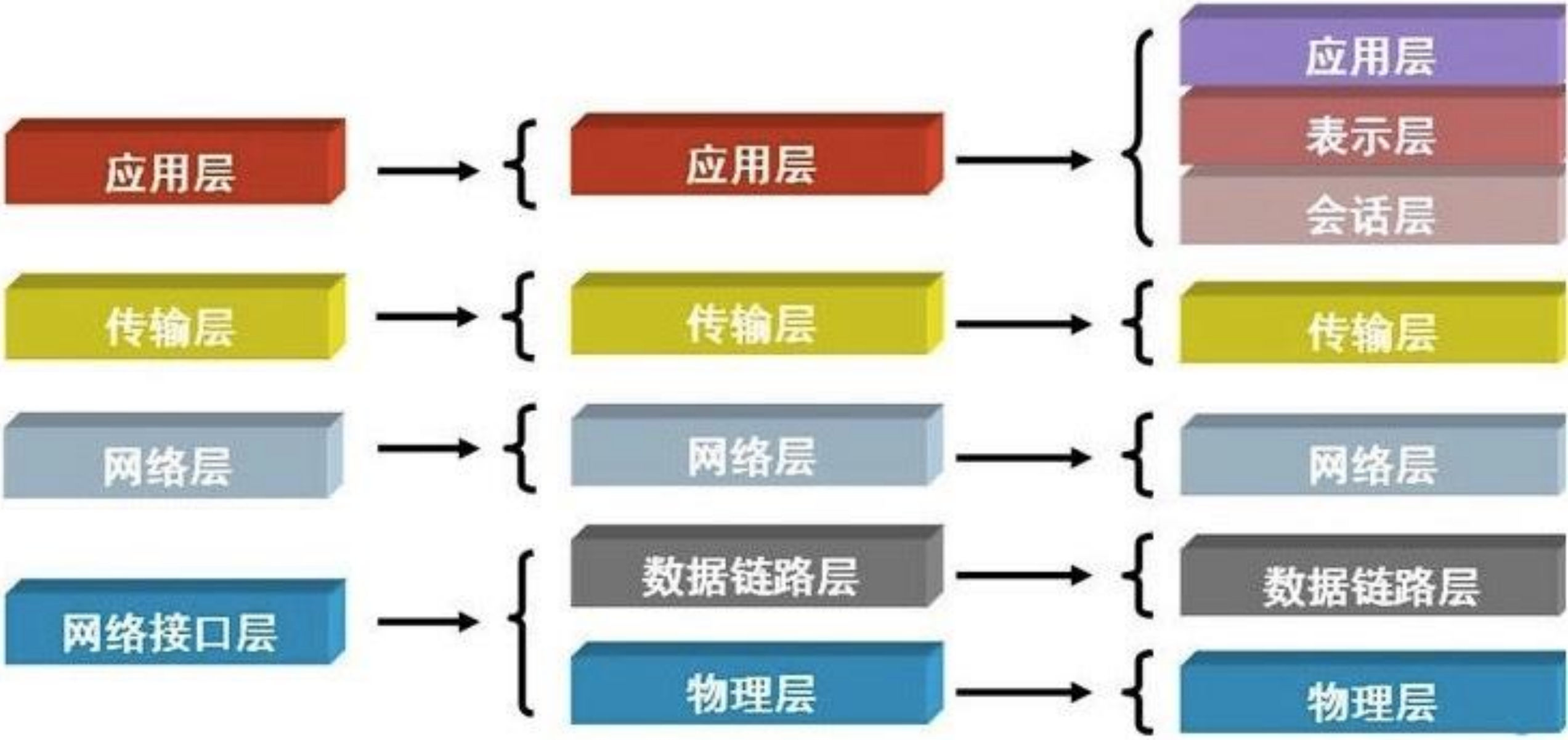


TCP/IP协议族中各协议之间的关系

网络基础-协议

- 在早期，不同的公司都推出了属于自己的私有网络协议，相互之间不能兼容
- 于是，ISO（国际标准化组织）站出来：干脆这样，我给大家制定一个通用的网络通信协议，该协议是国际标准
- 于是ISO博览众家之长，制订了“一堆”详细的，复杂的，繁琐的，精确的网络通信协议
- 不过这堆协议太复杂了，为了理清思路，便于学习，将他们分了7类（也就是分了7层），不同层代表不同的功能，并把这些协议归到相应的层里面去
- 国际标准出来了，接下来就要软件/硬件厂商去实现了。但实际上各厂商并没有完整实现7层协议，因为7层协议栈追求全能、完善，导致它太过复杂，实现起来太难了

- 于是，实际使用时，按4层划分（5层划分非官方）


应用层
应用层
应用层
表示层
会话层
传输层
传输层
传输层
网络层
网络层
网络层
网络接口层
数据链路层
数据链路层
物理层
物理层

网络基础-协议

- OSI七层协议，是英文Open System Interconnect的缩写，中文翻译开放系统互联
- TCP/IP定义了电子设备如何连入因特网，以及数据如何在它们之间传输的标准
- 4层的层级结构中，每一层都呼叫它的下一层所提供的网络来完成自己的需求
- 其中的应用层关注的是应用程序的细节，而不是数据在网络中的传输活动
  其他三层主要处理所有的通信细节，对应用程序一无所知；
  - 应用层：应用程序间沟通的层，不同的文件系统有不同的文件命名原则和不同的文本行表示方法等，不同的系统之间传输文件还有各种不兼容问题，这些都将由应用层来处理
  - 传输层：在此层中，它提供了节点间的数据传送服务，如传输控制协议（TCP）、用户数据报协议（UDP）等，这一层负责传送数据，并且确定数据已被送达并接收
  - 网络层：负责提供基本的数据包传送功能，让每一块数据包都能够到达目的主机。网络层接收由更低层发来的数据包，并把该数据包发送到更高层，相反，IP层也把从TCP或UDP层接收来的数据包传送到更低层
  - 网络接口层：对实际的网络媒体的管理，定义如何使用实际网络来传送数据（处理机械的、电气的和过程的接口）

Socket编程-简介

- socket:通过网络完成进程间通信的方式（区别于一台计算机之间进程通信）
- Socket的英文原义是“插孔”。通常也称作"套接字“

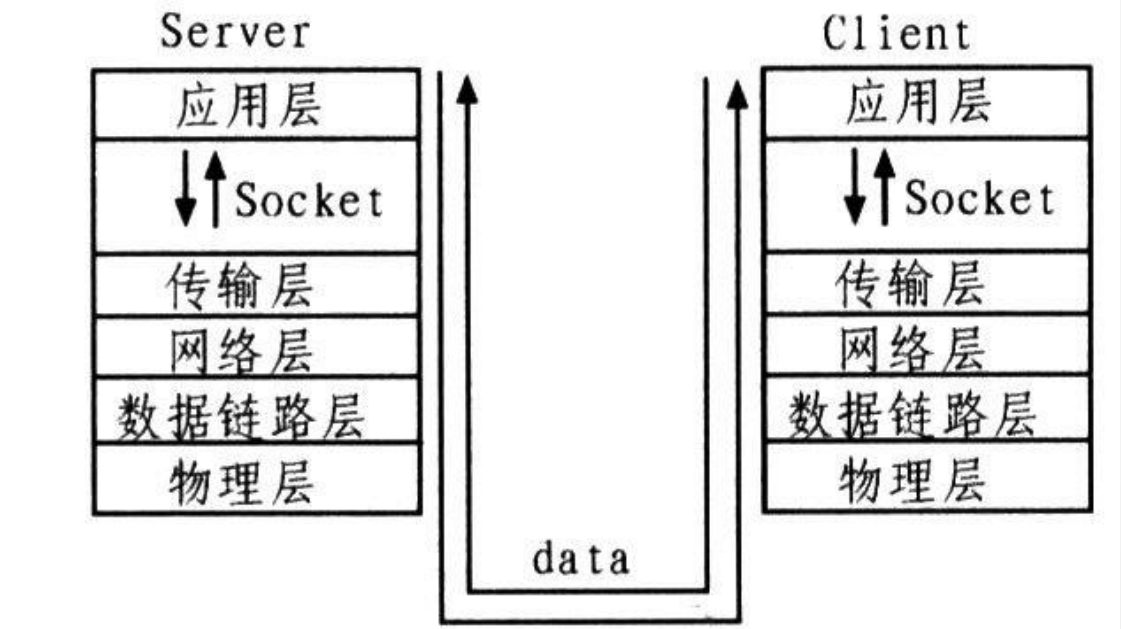


图 2 Socket 是应用层与传输层之间的桥梁

Socket编程-简介

- Socket本质是编程接口(API)： Socket 是对 TCP/IP 协议的封装，Socket 只是个编程接口不是协议，通过 Socket 我们才能使用 TCP/IP 协议簇（程序员层面）

- TCP/IP也要提供可供程序员做网络开发所用的接口，这就是Socket编程接口；HTTP是轿车，提供了封装或者显示数据的具体形式；Socket是发动机，提供了网络通信的能力
- 最重要的是，Socket是面向客户/服务器模型而设计的，针对客户和服务器程序提供不同的Socket系统调用
- 套接字之间的连接过程可以分为三个步骤：服务器监听，客户端请求，连接确认

Socket编程-创建Socket

- 创建Socket：

```
import socket
#导入套接字模块
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
#s此时是一个socket对象，拥有发送和接收网络数据的功能
```

  - 该函数带有两个参数（参数必须写）
    - AF_INET（ipv4协议用于 Internet 进程间通信）
    - 套接字类型， 可以是 SOCK_STREAM（流式套接字， 用于 TCP 协议） 或者 SOCK_DGRAM（数据报套接字， 用于 UDP 协议）
      - TCP慢但是稳定不会丢数据
      - UDP快但是可能会丢数据（黑客攻击）
  - 确定了IP地址端口号（ipv4协议），TCP或UDP协议之后，计算机之间可以进行通信

Socket编程-udp和tcp

- **UDP --- User Data Protocol，用户数据报协议，** 是一个无连接的简单的面向数据报的传输层协议。 UDP不提供可靠性， 它只是把应用程序传给IP层的数据报发送出去， 但是并不能保证它们能到达目的地。 由于UDP在传输数据报前不用在客户和服务器之间建立一个连接， 且没有超时重发等机制， 故而传输速度很快

- UDP一般用于多点通信和实时的数据业务， 比如：
  - 语音广播
    视频
    QQ
    TFTP(简单文件传送)

  - 可以理解为写信

Socket编程-udp和tcp

- TCP（Transmission Control Protocol，**传输控制协议**）是面向连接的协议，也就是说，在收发数据前，必须和对方建立可靠的连接

- 一个TCP连接必须要经过三次“对话”才能建立起来，其中的过程非常复杂，只简单的描述下这三次对话的简单过程：
  - 主机A向主机B发出连接请求数据包：“我想给你发数据，可以吗？”，这是第一次对话
  - 主机B向主机A发送同意连接和要求同步（同步就是两台主机一个在发送，一个在接收，协调工作）的数据包：“可以，你什么时候发？”，这是第二次对话
  - 主机A再发出一个数据包确认主机B的要求同步：“我现在就发，你接着吧！”，这是第三次对话
  - 三次“对话”的目的是使数据包的发送和接收同步，经过三次“对话”之后，主机A才向主机B正式发送数据

  - 可以理解为打电话，先建立通道

Socket编程-udp和tcp

- **TCP与UDP的区别：**
  1. 基于连接与无连接
  2. 对系统资源的要求（TCP较多，UDP少）
  3. UDP程序结构较简单
  4. 流模式与数据报模式
  5. TCP保证数据正确性，UDP可能丢包，TCP保证数据顺序，UDP不保证

Socket编程-udp编程

- 发送数据：为看到效果先安装“网络调试助手”

```
from socket import *
s = socket(AF_INET, SOCK_DGRAM) #创建套接字
addr = ('192.168.1.17', 8080) #准备接收方地址
data = input("请输入：")
s.sendto(data.encode(),addr)
#发送数据时，python3需要将字符串转成byte
#encode(‘utf-8’)# 用utf-8对数据进行编码，获得bytes类型对象
#decode（）反过来
s.close()
```

Socket编程-udp编程

- 发送数据给飞秋

  飞秋使用：2425端口

  发送普通数据，飞秋不会响应，必须发送特殊格式的内容

  1:123123:吴彦祖:吴彦祖-pc:32:haha

  飞秋有自己的应用层协议

- 1，表示版本
- 后面的数字发送的时间，随便写
- 32代表发送消息
- 飞秋炸弹：循环不延时发消息（可能会造成卡死）

注意：IP和端口在网络通信中缺一不可，用到的协议也要匹配，例如飞秋用的是udp协议，使用TCP协议发数据是无效的

udp理解为写信（只有收件人地址），TCP理解为打电话（先拨号建立通路，需要通路稳定）

Socket编程-udp编程

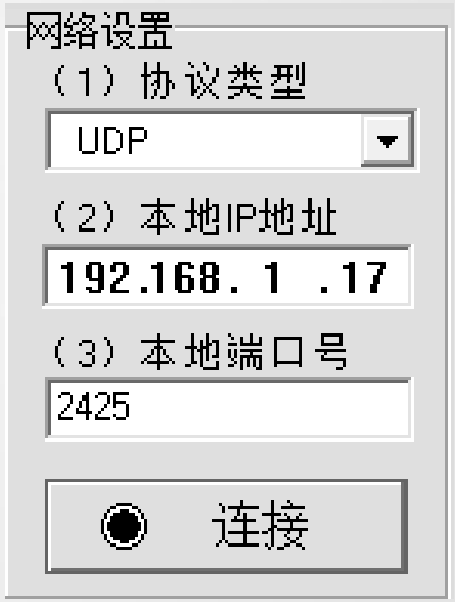
- 接收数据

```
from socket import *
s = socket(AF_INET, SOCK_DGRAM) #创建套接字
addr = ('192.168.1.17', 8080) #准备接收方地址
data = input("请输入：")
s.sendto(data.encode(),addr)
#等待接收数据
redata = s.recvfrom(1024)
#1024表示本次接收的最大字节数
print(redata)
s.close()
```

Socket编程-udp编程

- 绑定信息：
  如果信息没有绑定，每发送一次信息，系统会随机分配一个端口

```
【Receive from 192.168.1.17 : 49884 】: hehe
【Receive from 192.168.1.17 : 65344 】: hehe
【Receive from 192.168.1.17 : 65345 】: hehe
【Receive from 192.168.1.17 : 65346 】: hehe
【Receive from 192.168.1.17 : 65347 】: hehe
```



- 还要避免同一台计算机上的不同进程端口号相同的问题

Socket编程-udp编程

- 绑定信息：让一个进程可以使用固定的端口
- 一般情况下，发送方不绑定端口，接收方会绑定

```
from socket import *
s = socket(AF_INET, SOCK_DGRAM) #创建套接字
s.bind(('', 8788))    #绑定一个端口，ip地址和端口号，ip一般不用写
addr = ('192.168.1.17', 8080)    #准备接收方地址
data = input("请输入：")
s.sendto(data.encode(),addr)
redata = s.recvfrom(1024) #1024表示本次接收的最大字节数
print(redata)
s.close()
```

Socket编程-udp编程

- echo服务器：Echo服务是一种非常有用的用于调试和检测的工具。这个协议的作用也十分简单，接收到什么原封发回

```
from socket import *
#1创建套接字
udpSocket = socket(AF_INET, SOCK_DGRAM)
#2绑定本地信息，不使用随机分配的端口
bindAddr = ("",7088)
udpSocket.bind(bindAddr)#绑定
num = 0
```

```
while True:
    #接收对方发送的数据
    recvData = udpSocket.recvfrom(1024)
    print(recvData)
    #将接收到的数据回发给对方
    udpSocket.sendto(recvData[0],recvData[1])
    num += 1
    print("已将接收到的第%d个数据返回给对方，"%num)
udpSocket.close()
```

Socket编程-udp编程

- 聊天室

```
from socket import *
import time
#1创建套接字
udpSocket = socket(AF_INET, SOCK_DGRAM)
bindAddr = ("",7088)
udpSocket.bind(bindAddr)#绑定
while True:
    #接收对方发送的数据
    recvData = udpSocket.recvfrom(1024)
    print('【%s】 %s.%s' %(time.ctime(),recvData[1],recvData[0].decode("GB2312")))
    a = input("请输入：")
    udpSocket.sendto(a.encode('GB2312'),('192.168.1.17',8080))
#5关闭套接字
udpSocket.close()
```

Socket编程-udp

udp网络通信过程：（类似于发快递）

1，应用层编写数据（你好），然后向下层传递

2，传输层在数据前面加上端口号（包括发送端口和目的端口）

3，网络层继续在前面加上IP地址（包括原IP和目的IP）

4，链路层再在前面加上mac地址（mac：硬件地址，用来定义网络设备的位置）

此时数据变成了：mac地址 IP地址 端口号 数据内容

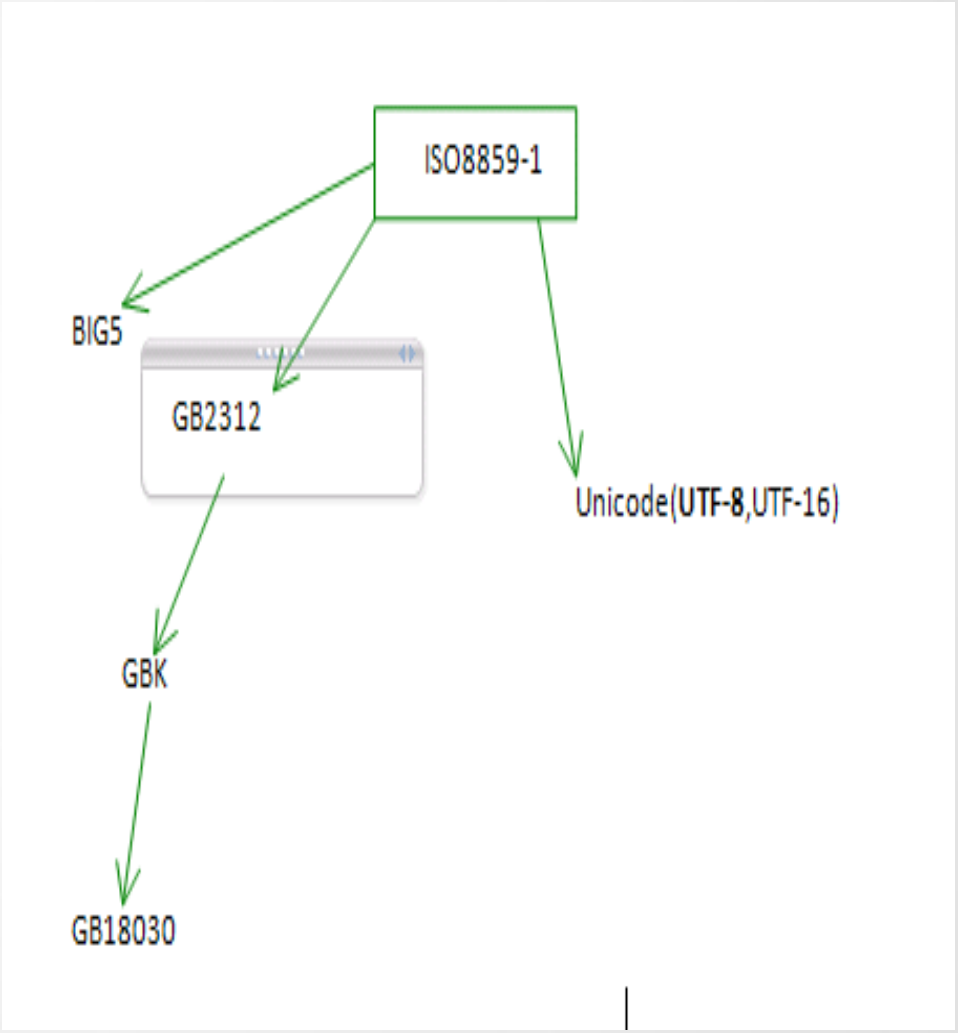
之后通过网络传输给另一台计算机的链路层开始逐步解析判断

Socket编程-udp编程

- 练习：
- 使用多线程完成一个全双工的聊天程序
  - 全双工（Full Duplex）是通讯传输的一个术语。通信允许数据在两个方向上同时传输（电话）
  - 单工是只允许甲方向乙方传送信息，而乙方不能向甲方传送（收音机）
  - 半双工：甲方发消息时乙方只能收不能发（对讲机）

字符集

- ASCII
  - 英文字符集 1个字节
- ISO8859-1
  - 西欧字符集　1个字节
- BIG5
  - 台湾的大五码，表示繁体汉字　2个字节
- GB2312
  - 大陆使用最早、最广的简体中文字符集 2个字节
- GBK
  - GB2312的扩展，可以表示繁体中文　2个字节
- GB18030
  - 最新GBK的扩展，可以表示汉字、维吾尔文、藏文等中华民族字符　2个字节
- Unicode
  - 国际通用字符集　2个字节



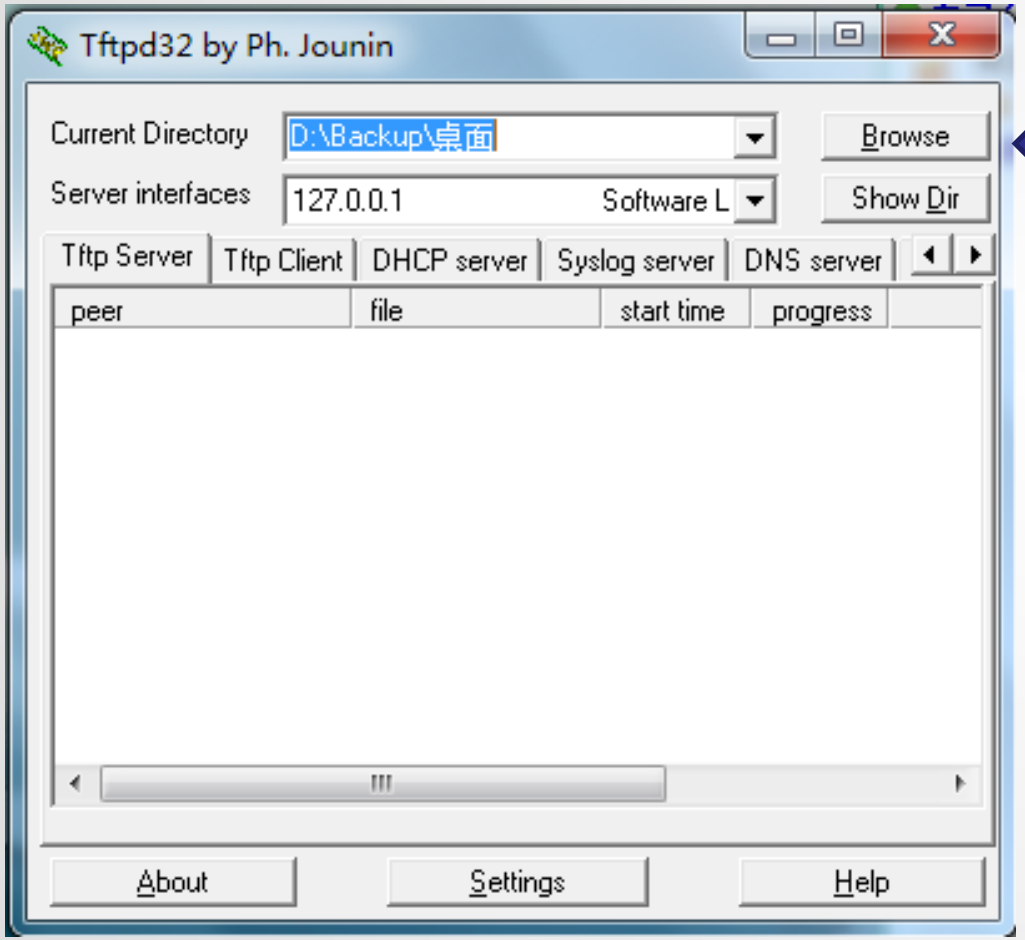
TFTP介绍

- TFTP（Trivial File Transfer Protocol,简单文件传输协议）是TCP/IP协议簇中的一个用来在客户端与服务器之间进行简单文件传输的协议

- 使用tftp这个协议，就可以实现简单文件的下载
- 特点：
  - 简单
    占用资源小
    适合传递小文件
    适合在局域网进行传递
    端口号为69
    基于UDP实现

TFTP介绍

- Tftpd32：共享服务器（可以从本机共享文件）



确定文件分享的路径

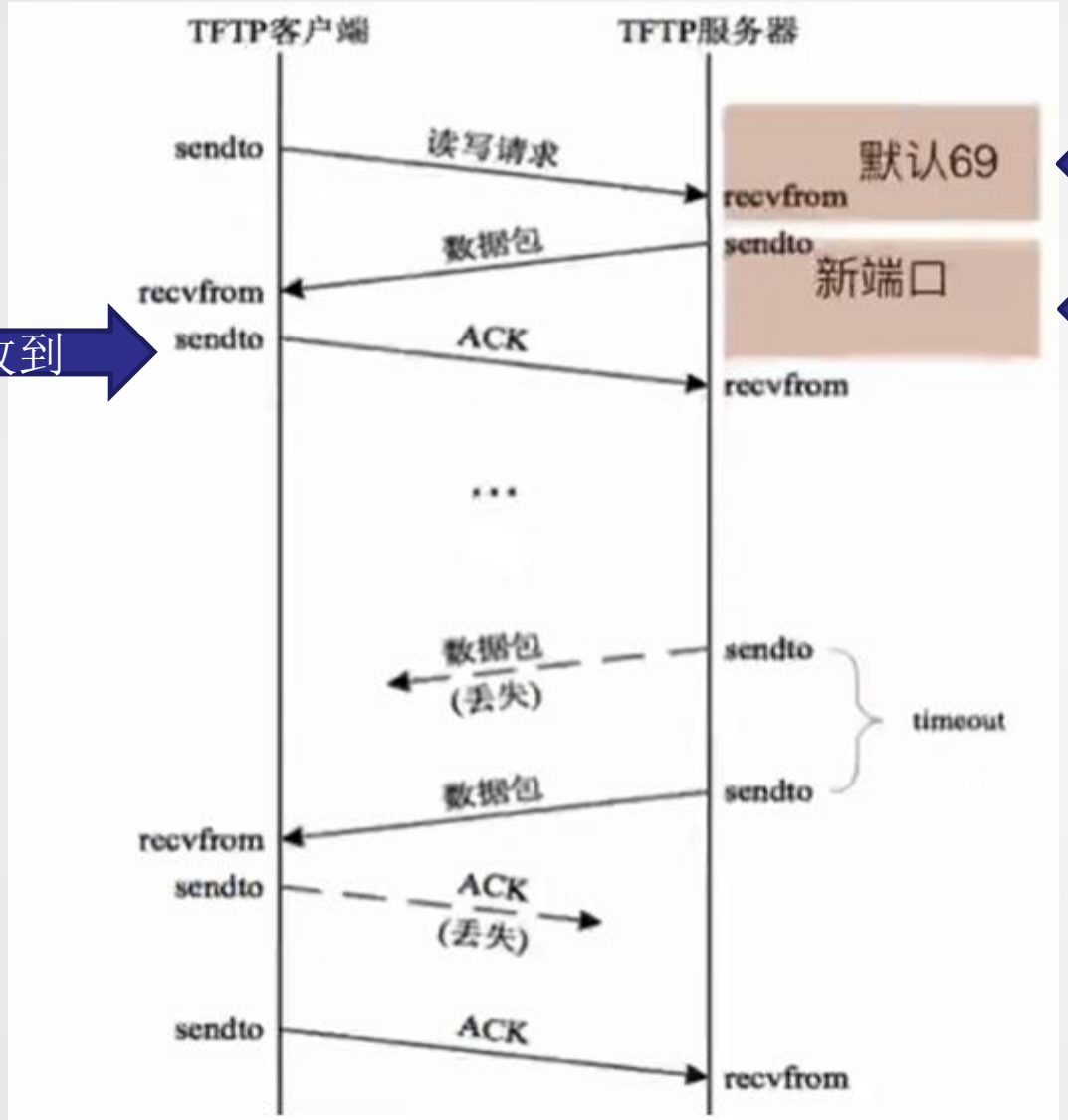
- browse：选择一个文件夹，确定给客户端文件时的搜索路径

- 客户端：数据接收方
- 服务器：数据发送方

TFTP介绍

- 有了服务器之后，还需要编写一个下载器（客户端）
- 实现TFTP下载器：
  - 下载：从服务器上将一个文件复制到本机上
  - 下载的过程：
    - 在本地创建一个空文件（与要下载的文件同名）
    - 向里面写数据（接收到一点就向空文件里写一点）
    - 关闭（接受完所有数据关闭文件）

TFTP介绍

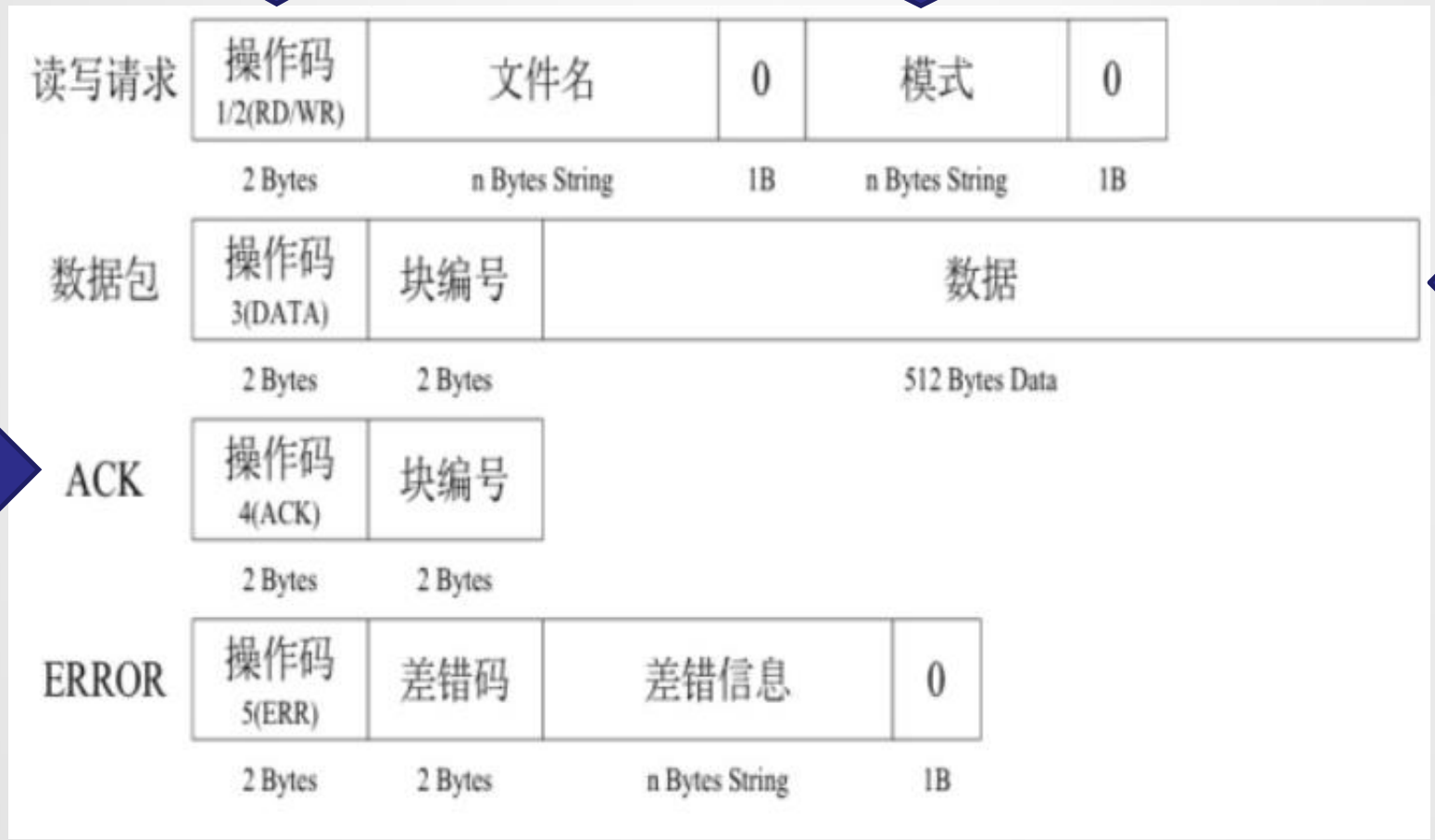


注意：服务器的69端口只用来接收读写请求，ack包不要发到69端口

TFTP格式要求

1下载
2上传

octet
传输格式固定



每次发送文件中的512个字节

确认包

TFTP介绍

- 当客户端接收到的数据小于516（2字节操作码+2个字节的序号+512字节数据） 时，就意味着服务器发送完毕了 （如果恰好最后一次数据长度为516，会再发一个长度为0的数据包）

- 构造下载请求数据：“1test.jpg0octet0”
  - import struct
  - cmb_buf = struct.pack(“!H8sb5sb”,1,b“test.jpg”,0,b“octet”,0)
    如何保证操作码（1/2/3/4/5）占两个字节？如何保证0占一个字节？

    #!H8sb5sb: ！ 表示按照网络传输数据要求的形式来组织数据（占位的格式）
    H 表示将后面的 1 替换成占两个字节
    8s 相当于8个s（ssssssss）占8个字节
    b 占一个字节

struct模块使用

- struct模块可以按照指定格式将Python数据转换为字符串,该字符串为字节流
- struct模块中最重要的三个函数是pack(), unpack(), calcsize()
  - # 按照给定的格式(fmt)，把数据封装成字符串(实际上是类似于c结构体的字节流)
    pack(fmt, v1, v2, ...)
  - # 按照给定的格式(fmt)解析字节流string，返回解析出来的元组
    unpack(fmt, string)
  - # 计算给定的格式(fmt)占用多少字节的内存
    calcsize(fmt)

- struct.pack(“!H8sb5sb”,1,“test.jpg”,0,“octet”,0)
- struct.pack("!HH",4,p_num)
- cmdTuple = struct.unpack("!HH", recvData[:4])

struct模块使用

FORMAT	C TYPE	PYTHON TYPE	STANDARD SIZE
x	pad byte	no value	
c	char	string of length 1	1
b	signed char	integer	1
B	unsigned char	integer	1
?	_Bool	bool	1
h	short	integer	2
H	unsigned short	integer	2
i	int	integer	4
I	unsigned int	integer	4
l	long	integer	4
L	unsigned long	integer	4
q	long long	integer	8
Q	unsigned long long	integer	8
f	float	float	4
d	double	float	8
s	char[]	string	
p	char[]	string	
P	void *	integer	

q和Q只适用于64位机器;
每个格式前可以有一个数字,表示这个类型的个数,如s格式表示一定长度的字符串,4s表示长度为4的字符串

struct模块使用

- 向服务器发送请求：

```
import struct
from socket import *
cmb_buf = struct.pack("!H9sb5sb",1,b"test1.jpg",0,b"octet",0)
udpSocket = socket(AF_INET, SOCK_DGRAM)
udpSocket.sendto(cmb_buf,("192.168.187.1", 69))
udpSocket.close()
```

向其他电脑发送这个请求，查看是否能抓到这个数据包（udp.port == 69，tftp）

TFTP客户端编程（下载）

```
import struct
from socket import *
filename = 'test1.jpg'
server_ip = '192.168.1.3'
send_data = struct.pack('!H%dsb5sb' % len(filename), 1, filename.encode(), 0, 'octet'.encode(), 0)
s = socket(AF_INET, SOCK_DGRAM)
s.sendto(send_data, (server_ip, 69))  # 第一次发送，连接服务器69端口
f = open(filename, 'ab')  #a:以追加模式打开（必要时可以创建）append;b:表示二进制
while True:
    recv_data = s.recvfrom(1024)  # 接收数据
    caozuoma, ack_num = struct.unpack('!HH', recv_data[0][:4])  # 获取数据块编号
    rand_port = recv_data[1][1]  # 获取服务器的随机端口
    if int(caozuoma) == 5:
        print('文件不存在...')
        break
    print("操作码：%d,ACK：%d,服务器随机端口：%d,数据长度：%d"%(caozuoma, ack_num, rand_port, len(recv_data[0])))
    f.write(recv_data[0][4:])#将数据写入
    if len(recv_data[0]) < 516:
        break
    ack_data = struct.pack("!HH", 4, ack_num)
    s.sendto(ack_data, (server_ip, rand_port))  # 回复ACK确认包
```

TFTP客户端编程（上传）

```
import struct
from socket import *

filename = "桌面.jpg"
data = struct.pack(f"!H{len(filename.encode('gb2312'))}sb5sb",2,filename.encode("gb2312"),0,b"octet",0)
s = socket(type=SOCK_DGRAM)
s.sendto(data,("127.0.0.1",69))
f = open(filename,"rb")
while True:
    ret = s.recvfrom(1024)
    duankouhao = ret[1]
    data1,data2 = struct.unpack("!HH",ret[0][:4])
    if data1 == 4:
        data = f.read(512)
        dabao = struct.pack(f"!HH{len(data)}s",3,data2+1,data)
        s.sendto(dabao,duankouhao)
        if len(data) < 512:
            break
```

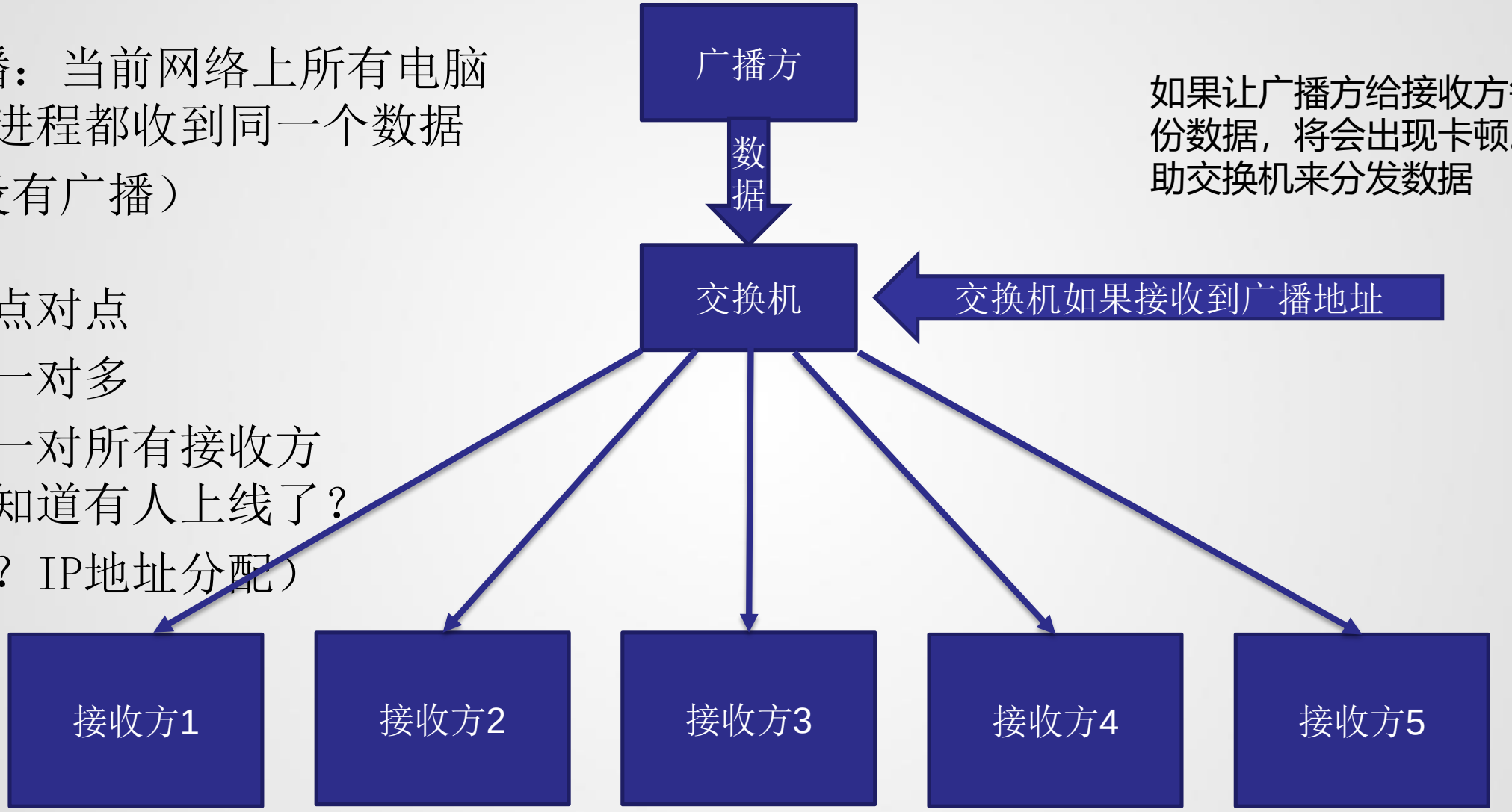
udp广 播

udp广播：当前网络上所有电脑的某个进程都收到同一个数据
（tcp没有广播）

单播：点对点

多播：一对多

广播：一对所有接收方
（飞秋知道有人上线了？
谁在线？IP地址分配）



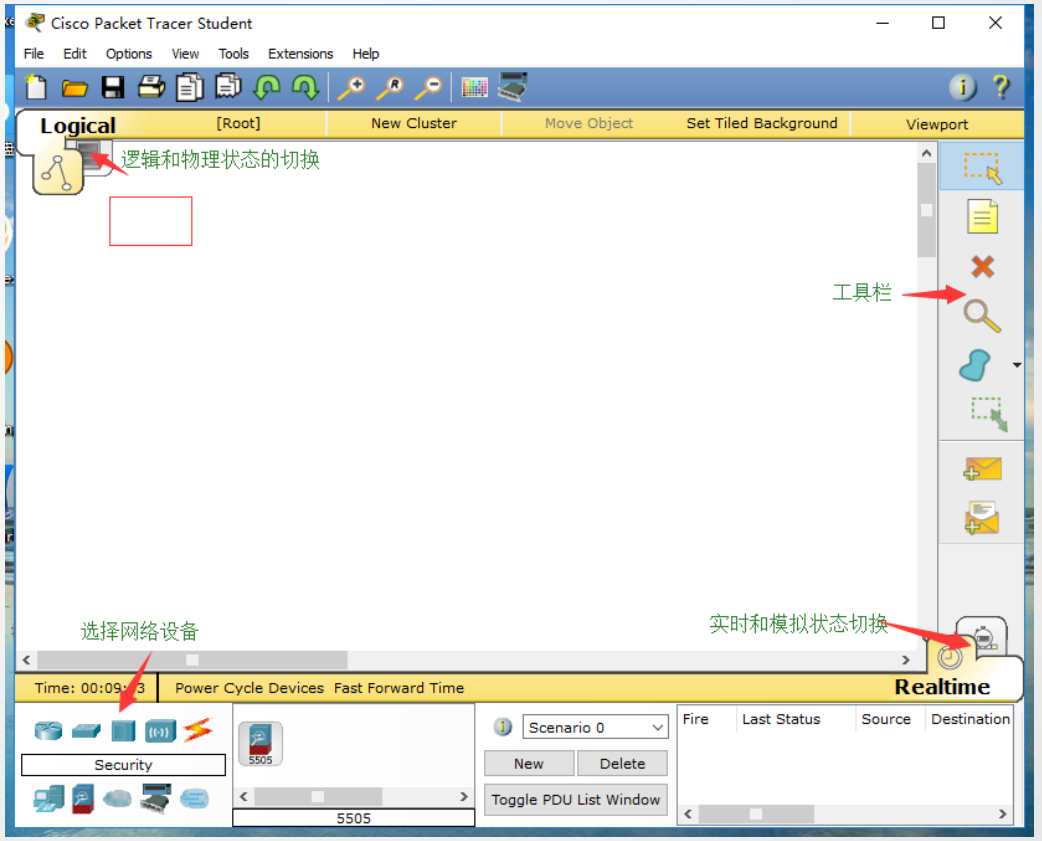
如果让广播方给接收方每人发一份数据，将会出现卡顿。所以借助交换机来分发数据

udp广播

```
import socket
dest = ( ‘<broadcast>’ ,7788）
#<broadcast>自动识别当前网络的广播地址
#创建udp套接字
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
#对这个需要发送广播数据的套接字进行修改设置，否则不能发送广播数据
s.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1）#允许s发送广播数据
#setsocketopt 设置套接字选项
#以广播形式发送数据到本网络的所有电脑中
s.sendto(b’Hi’,dest)
print("等待回复")
while True:
    (buf, address) = s.recvfrom(2048)
    print(address,buf.decode(“GB2312”))
```

Packet Tracer 介绍&安装

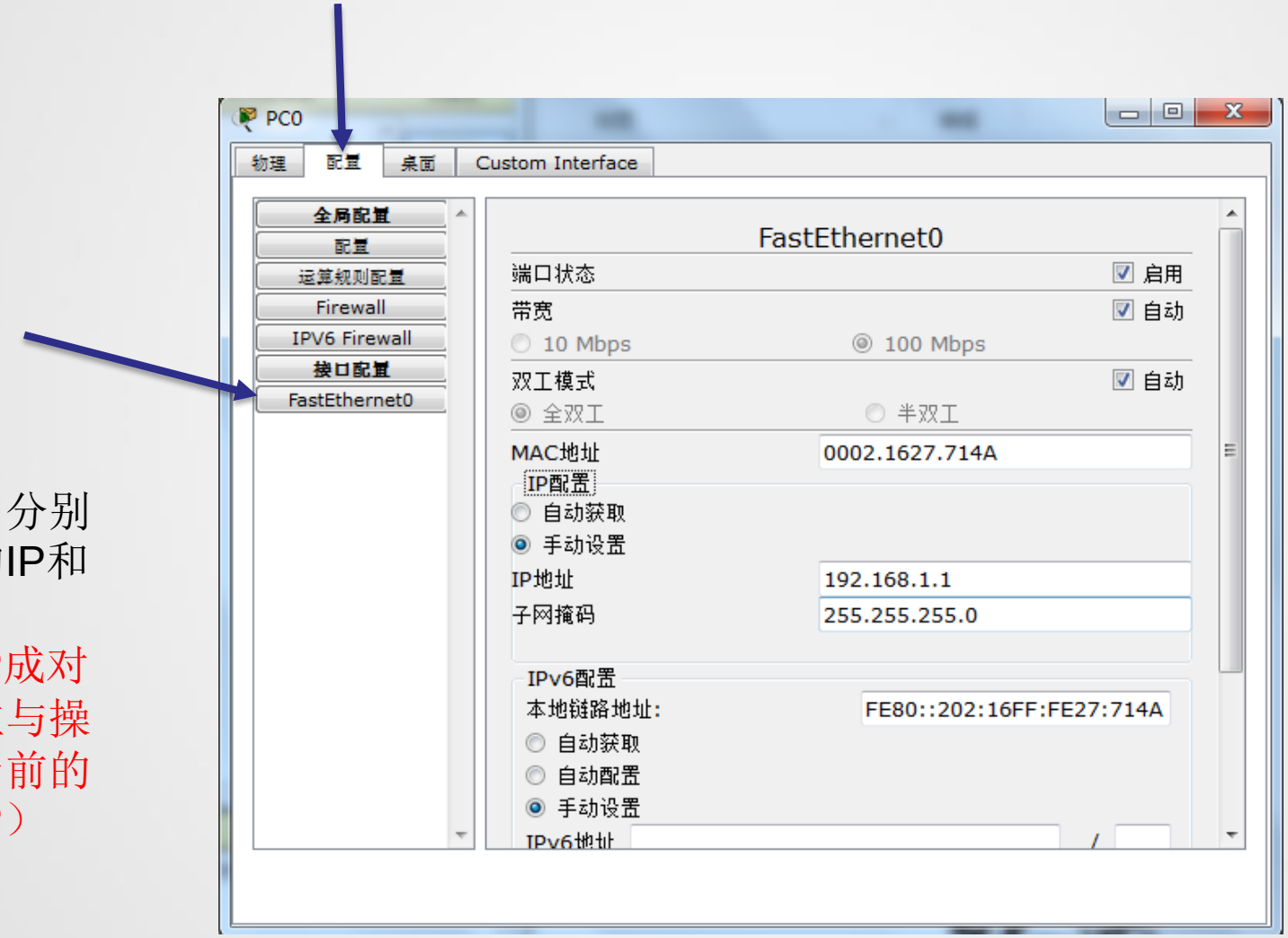
- Packet Tracer 是由Cisco(思科)公司发布的一个辅助学习工具，为学习思科网络课程的初学者去设计、 配置、 排除网络故障提供了网络模拟环境（不用买硬件）
- 可以提供数据包在网络中行进的详细处理过程
  观察网络实时运行情况
  （辅助学习网络通信过程）



2台电脑连网

- 添加两台电脑
- 连接
- 设置网络
- ping

2台电脑连网



单击终端设备，分别设置两台终端的IP和子网掩码

子网掩码：与IP成对出现，进行按位与操作后可以得到当前的网络号（哪类IP）

2台电脑连网

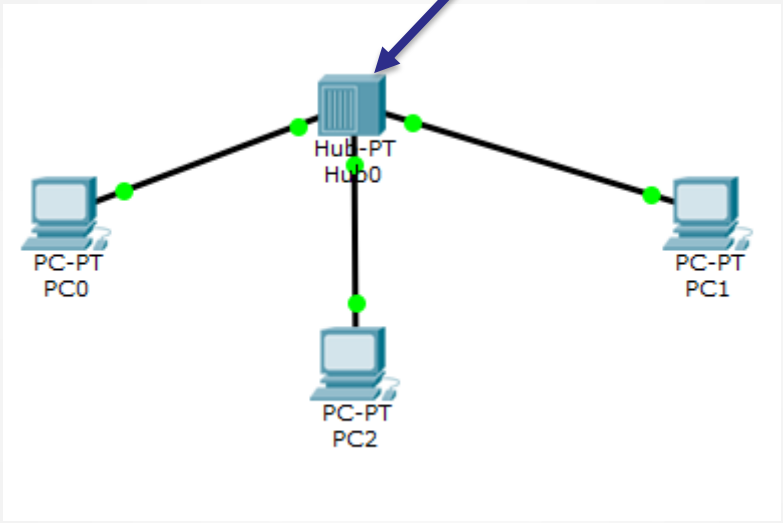
再单击其中一台终端，尝试IP地址能否ping通

（换一个网段还能否ping通？）

通过集线器连网

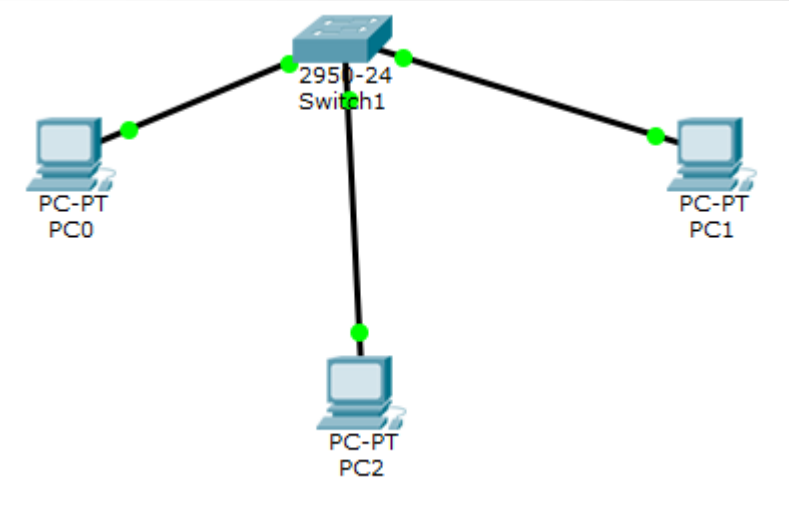
- 添加三台电脑
- 添加集线器
  （为什么不能连到之前的通路上？）
- 连网
- 设置网络
- ping

集线器类似于USB扩展口：用来链接多台电脑



通过交换机连网

- 集线器(HUB)是计算机网络中连接多个计算机或其他设备的连接设备，是对网络进行集中管理的最小单元。英文Hub就是中心的意思，像树的主干一样，它是各分支的汇集点。HUB是一个共享设备，主要提供信号放大和中转的功能，它把一个端口接收的所有信号向所有端口分发出去
- 交换机(Switch)是一种基于MAC（网卡的硬件地址）识别，能完成封装转发数据包功能的网络设备。交换机可以“学习”MAC地址，并把其存放在内部地址表中，通过在数据帧的始发者和目标接收者之间建立临时的交换路径，使数据帧直接由源地址到达目的地址
-



通过交换机连网

- 例如一个8口hub， 当端口1上的机器要给端口8上的机器发数据
- 端口1上hub上有没有数据在传输，如果没有，端口1就跳出来向hub上喊：“我有数据包要给端口8，请端口8听到后回话”
- 这个数据被以广播的方式发送到hub上的其余7个口上，每端口都会接到这样的数据包，然后端口2——端口7会发一则消息给1：“我不是端口8”
- 与此同时端口8会发消息给1：“我是端口8，找我吗？”端口1收到上述消息后，会和端口8进行确认，然后他们建立传输链接，完成数据转发。
- 等如果端口1在发送寻找断口8的消息后，没有得到相应，那它还会接着发这个消息，直到收到端口8的回答。等端口1和端口8完整数据转发后，假设他们还要进行通讯，那么hub上还会重复以上的过程
- 一个数据，需要送达所有的端口，这不但增加了数据转发的时间，而且hub往往会给网络带来广播风暴（广播数据充斥网络无法处理，并占用大量网络带宽，导致正常业务不能运行）

通过交换机连网

- 相同的工作交换机就不用这么麻烦
- 假设端口1和端口8从没有通信过，那么开始的时候，他们的工作和hub一样，端口1要在交换机上找端口8，一旦端口8返回确认信息，那再端口1上就会生成1个和端口8的地址对应表
- 这个表里面有所有和端口1通过信的端口，一旦有了这地址对应表，那在以后端口1要和端口8通讯，可以直接送达，而且其他的端口也不会知道他们之间正在转发数据，这样加快了数据转发时间，并且避免了广播风暴
  （ping之后点模拟测试以上规则）

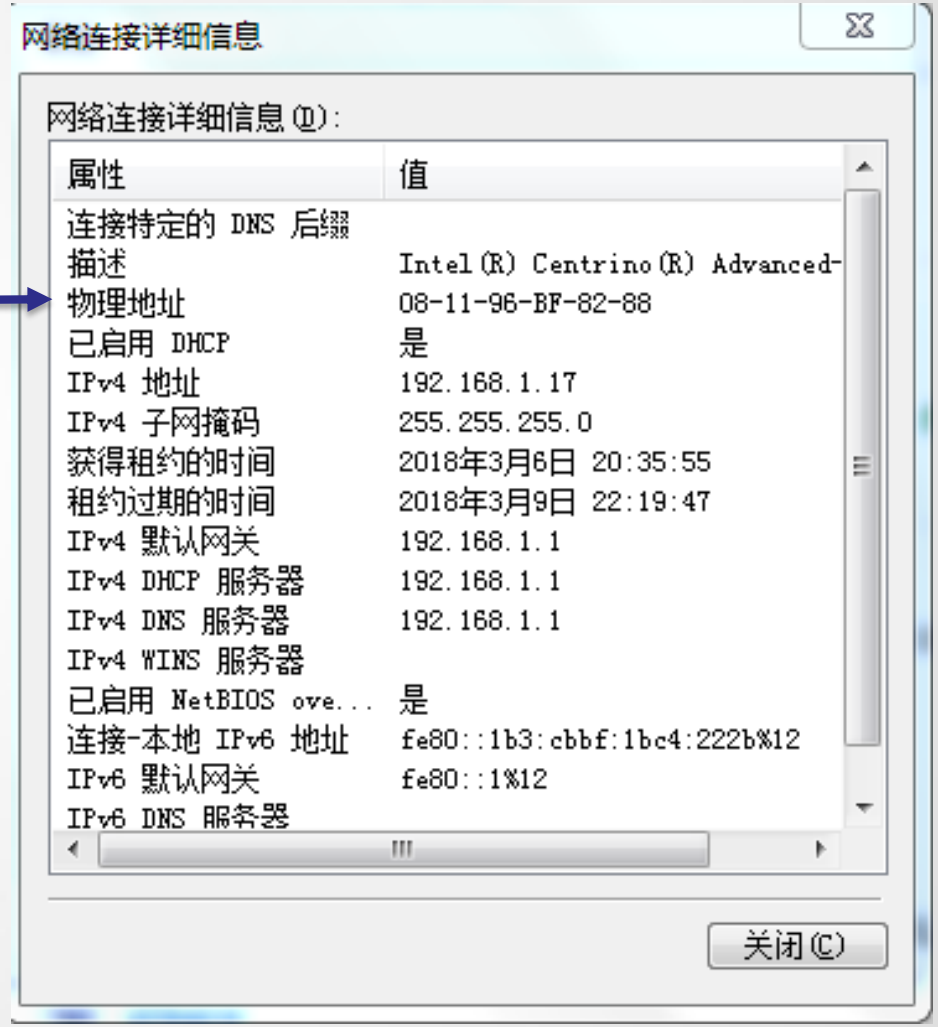
通过路由器连网

- 两台电脑之间能通信的前提是什么？
  - 在同一网段
- 多台电脑之间为什么不能把网线剪开连在一起？
  - 数据是通过电信号来传输的，多个电信号同时传输会出错
- 集线器hub有什么作用？
  - 链接多台电脑组成小型局域网
- 集线器和交换机的区别？
  - 集线器收到的所有数据包都会以广播形式发送
  - 交换机可以智能学习，如果已经通信过的设备之间，可以直接通信

通过路由器连网

物理地址（实际地址）：由网络设备制造商生产时写在硬件内部
IP地址与MAC地址在计算机里都是以二进制表示的，IP地址是32位的，而MAC地址则是48位的（6个字节）
如：08:00:20:0A:8C:6D就是一个MAC地址，其中前3组16进制数08:00:20代表网络硬件制造商的编号，它由IEEE（电气与电子工程师协会）分配
而后3组16进制数0A:8C:6D代表该制造商所制造的某个网络产品（如网卡）的系列号

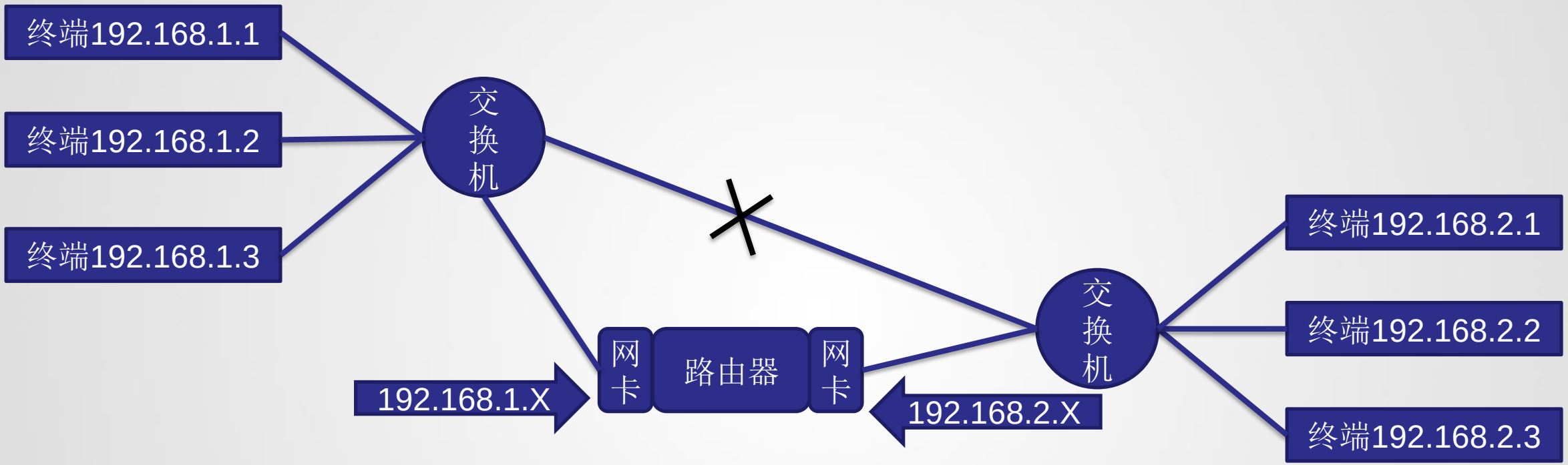
MAC地址在世界是惟一的
（可以直接理解为网卡的序列号）
通过IP地址、端口号、MAC地址　保证了数据的稳定传输



通过路由器连网

- 路由器：确定一条路径的设备，路由器是连接因特网中用来链接网络号不同的、不同的网络，相当于中间人各局域网、广域网的设备，它会根据信道的情况自动选择和设定路由，以最佳路径，按前后顺序发送信号的设备。
- 路由器的一个作用是连通不同的网络，另一个作用是选择信息传送的线路
- 选择通畅快捷的近路，能大大提高通信速度，减轻网络系统通信负荷，节约网络系统资源，提高网络系统畅通率（举例：怎么去柏林）

通过路由器连网



同一个局域网当中的终端之间进行通讯的基础是处于同一个网段中，一个路由器至少有两个网卡，能够链接不同网段的网络使之可以通信（了解即可）

TCP

TCP：传输控制协议（使用情况多于udp）

稳定：保证数据一定能收到

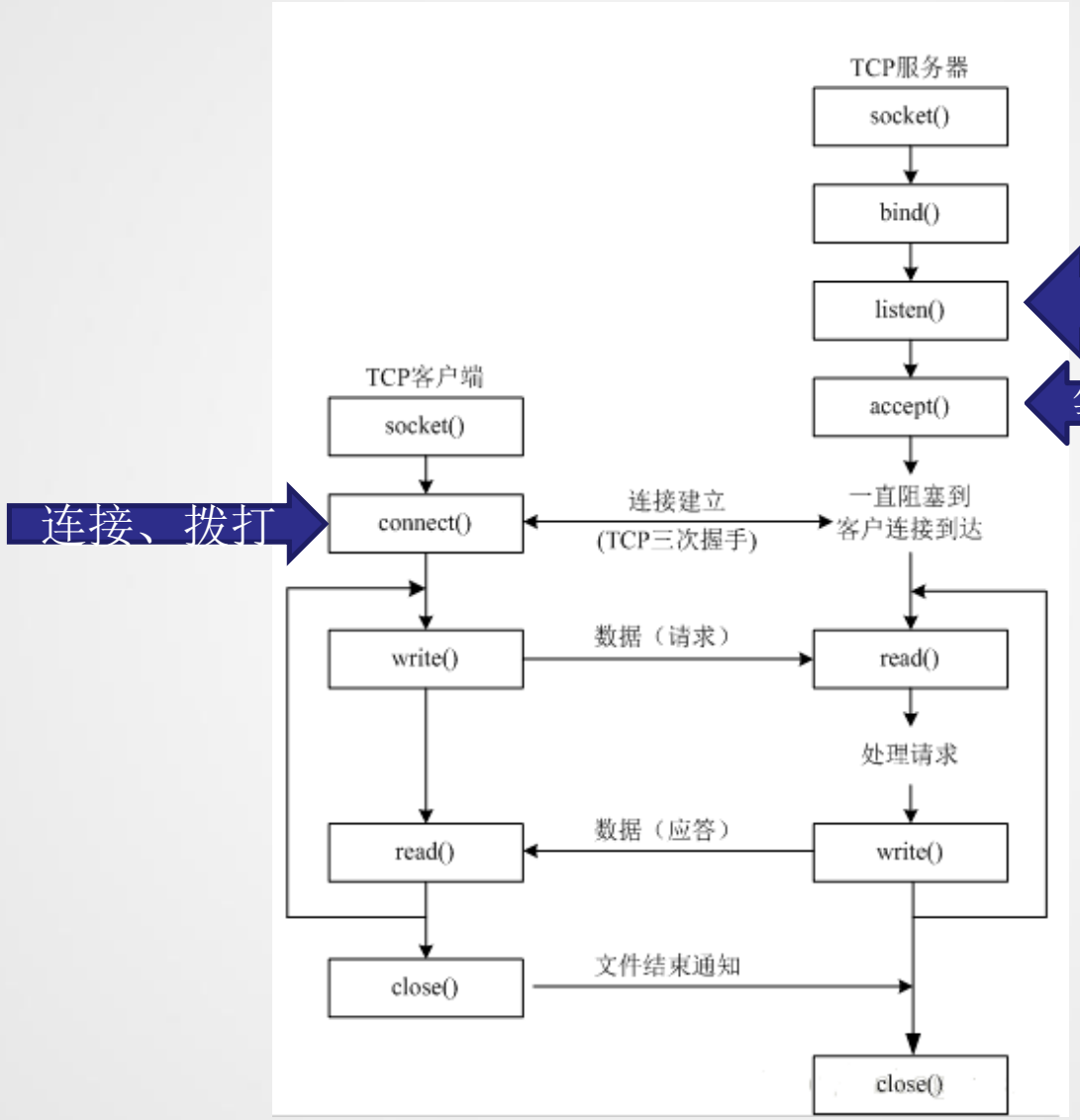
相对UDP会慢一点

web服务器一般都使用TCP（银行转账，稳定比快要重要）

TCP通信模型：

在通信之前，必须先等待建立链接

TCP



设置最大连接数

等待接受socket连接

连接、拨打

TCP的三次握手

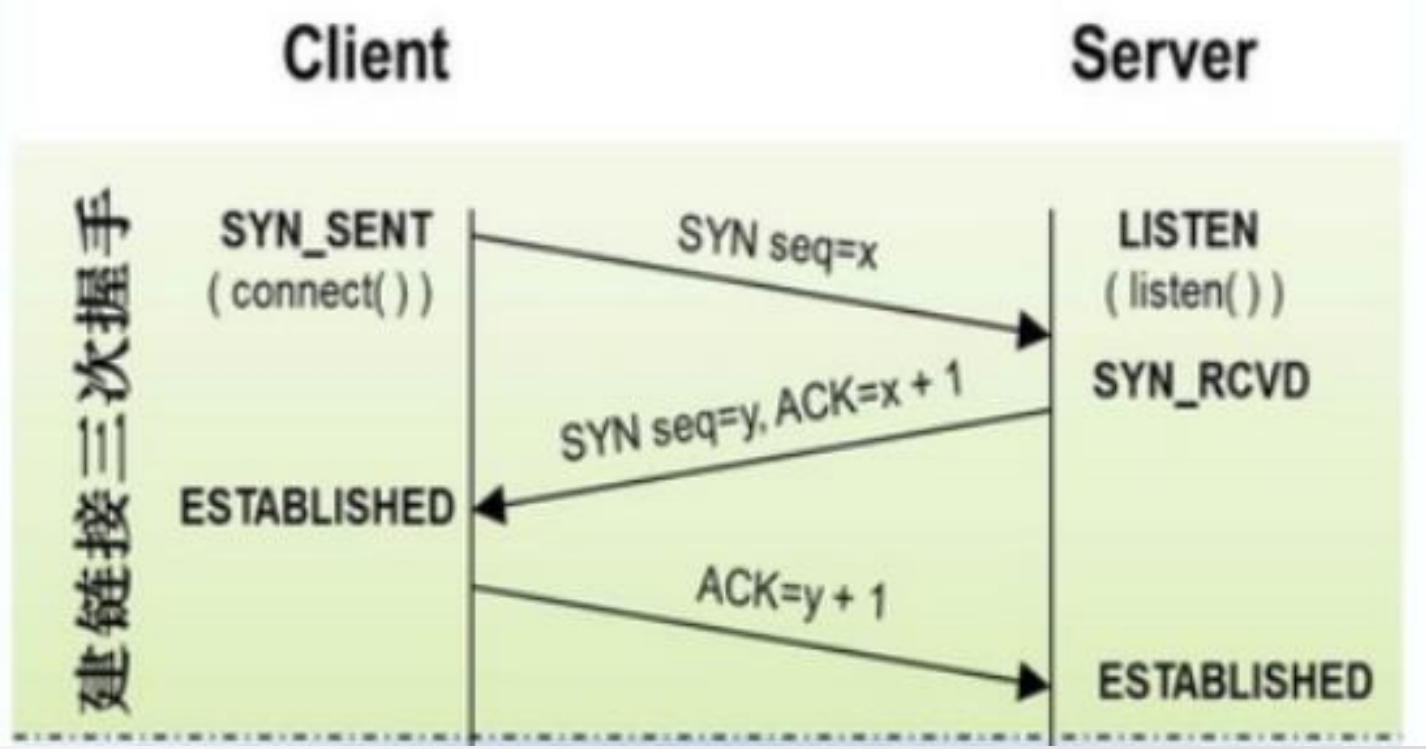
第一次握手：建立连接时，客户端发送SYN（请求同步）包到服务器，并进入SYN_SENT（请求连接）状态，等待服务器确认

第二次握手：服务器收到syn包，必须确认客户的SYN（x+1），同时自己也发送一个SYN包（syn=y），即SYN+ACK包，此时服务器进入SYN_RECV(SYN派遣)状态

第三次握手：客户端收到服务器的SYN＋ACK包，向服务器发送确认包ACK(y+1)，此包发送完毕，客户端和服务器进入ESTABLISHED（TCP连接成功）状态，完成三次握手。客户端与服务器才正式开始传送数据

理想状态下，TCP连接一旦建立，在通信双方中的任何一方主动关闭连接之前，TCP 连接都将被一直保持下去

SYN：请求同步
ACK：确认同步



TCP服务器

- 在tcp传输过程中，如果有一方收到了对方的数据，一定会发送一个ACK确认包给发送方

TCP的四次挥手

- 断开一个TCP连接则需要“四次挥手”
  - 第一次挥手：主动关闭方调用close，会发送一个长度为0的数据包以及FIN（结束标志）用来关闭主动方到被动关闭方的数据传送，告诉被动关闭方：我已经不会再给你发数据了，但是，此时主动关闭方还可以接受数据
  - 第二次挥手：被动关闭方收到FIN包后，发送一个ACK给对方，确认序号为收到序号+1
  - 第三次挥手：被动关闭方发送一个FIN，用来关闭被动关闭方到主动关闭方的数据传送，也就是告诉主动关闭方，我的数据也发送完了，不会再给你发数据了
  - 第四次挥手：主动关闭方收到FIN后，发送一个ACK给被动关闭方，确认序号为收到序号+1，至此，完成四次挥手

TCP服务器

- 长连接：三次握手四次挥手之间分多次传递完所有数据（优酷看视频、在线游戏），长时间占用某个套接字
- 短连接：三次握手四次挥手之间传递少部分数据，多次握手挥手才传递完所有数据（浏览器），短时间占用

- tcp服务器流程如下：
  1. socket创建一个套接字
  2. bind绑定ip和port
  3. listen设置最大连接数，收到连接请求后，这些请求需要排队，如果队列满，就拒绝请求
  4. accept等待客户端的链接、接收连接请求
  5. recv/send接收发送数据

TCP服务器

- ```
  from socket import *
  tcpSerSocket = socket(AF_INET, SOCK_STREAM)
  address = ( ‘’ , 7788)
  tcpSerSocket.bind(address)
  tcpSerSocket.listen(5)#设置最大连接数
  ```
- ```
  newSocket, clientAddr = tcpSerSocket.accept()
  # 如果有新的客户端来链接服务器， 那么就产生一个新的套接字
  # newSocket用来为这个客户端服务（10086小妹）
  # tcpSerSocket就可以省下来等待其他新客户端的链接
  # 接收对方发送过来的数据， 最大接收1024个字节
  recvData = newSocket.recv(1024)  #接收tcp数据
  # 发送一些数据到客户端
  newSocket.send(“thank you !”)   #发送tcp数据
  # 关闭为这个客户端服务的套接字， 只要关闭了， 就意味着为不能再为这个客户端服务了
  newSocket.close()
  # 关闭监听套接字， 只要这个套接字关闭了， 就意味着整个程序不能再接收任何新的客户端的连接
  tcpSerSocket.close()
  ```

- 使用网络调试助手测试代码

TCP客户端

```
from socket import *
clientSocket = socket(AF_INET, SOCK_STREAM)

serAddr = ('192.168.1.17', 7788)
#链接服务器
clientSocket.connect(serAddr)

clientSocket.send(b"hello")
recvData = clientSocket.recv(1024)
print("接收到的数据为：", recvData)
clientSocket.close()
```

单进程服务器（每次只能服务一个客户端）

```
from socket import *
serSocket = socket(AF_INET, SOCK_STREAM)
localAddr = ('',7788)
serSocket.bind(localAddr)
serSocket.listen(5)
while True:
    print("主进程等待新客户端")
    newSocket,destAddr = serSocket.accept()
    print("主进程接下来负责处理",str(destAddr))
    try:
        while True:
            recvData = newSocket.recv(1024)
            if len(recvData)>0: #如果收到的客户端数据长度为0，代表客户端已经调用close（）下线
                print("接收到", str(destAddr),recvData)
            else:
                print("%s-客户端已关闭" %str(destAddr))
                break
    finally:
        newSocket.close()
serSocket.close()
```

并发服务器

serSocket.setsockopt(SOL_SOCKET,SO_REUSEADDR,1）

重新设置套接字选项，重复使用绑定的信息

当有一个有相同本地地址和端口的socket1处于TIME_WAIT状态时，而你启动的程序的socket2要占用该地址和端口，你的程序就要用到SO_REUSEADDR选项。

多进程服务器（同时为多个客户端服务）

```
from socket import *
from multiprocessing import *
from time import sleep
# 处理客户端的请求并为其服务
def dealWithClient(newSocket,destAddr):
    while True:
        recvData = newSocket.recv(1024)
        if len(recvData)>0:
            print('recv[%s]:%s'%(str(destAddr), recvData))
        else:
            print('[%s]客户端已经关闭'%str(destAddr))
            break
    newSocket.close()

def main():
    serSocket = socket(AF_INET, SOCK_STREAM)
    serSocket.setsockopt(SOL_SOCKET, SO_REUSEADDR , 1)
    localAddr = ('', 7788)
    serSocket.bind(localAddr)
    serSocket.listen(5)
```

```
try:
    while True:
        print('-----主进程，， 等待新客户端的到来------')
        newSocket,destAddr = serSocket.accept()
        print('-----主进程，， 接下来创建一个新的进程负责数据处理')
        client = Process(target=dealWithClient, args=(newSocket,destAddr))
        client.start()
        #因为已经向子进程中copy了一份（引用），
并且父进程中这个套接字也没有用处了
        #所以关闭
        newSocket.close()
finally:
    #当为所有的客户端服务完之后再进行关闭，
        表示不再接收新的客户端的链接
    serSocket.close()
if __name__ == '__main__':
    main()
```

多线程服务器（耗费的资源比多进程小一些）

```
from socket import *
from threading import Thread
from time import sleep
# 处理客户端的请求并执行事情
def dealWithClient(newSocket,destAddr):
    while True:
        recvData = newSocket.recv(1024)
        if len(recvData)>0:
            print('recv[%s]:%s'%(str(destAddr), recvData))
        else:
            print('[%s]客户端已经关闭'%str(destAddr))
            break
    newSocket.close()
def main():
    serSocket = socket(AF_INET, SOCK_STREAM)
    serSocket.setsockopt(SOL_SOCKET, SO_REUSEADDR , 1)
    localAddr = ('', 7788)
    serSocket.bind(localAddr)
    serSocket.listen(5)
```

多线程服务器（耗费的资源比多进程小一些）

```
try:
    while True:
        print('-----主进程，， 等待新客户端的到来------')
        newSocket,destAddr = serSocket.accept()
        print('主进程接下来创建一个新的线程负责处理 ', str(destAddr)))
        client = Thread(target=dealWithClient, args=(newSocket,destAddr))
        client.start()
        #因为线程中共享这个套接字， 如果关闭了会导致这个套接字不可用，
        #但是此时在线程中这个套接字可能还在收数据， 因此不能关闭
        #newSocket.close()
finally:
    serSocket.close()
if __name__ == '__main__':
    main()
```

socketsever

可以使用socketserver来创建socket用来简化并发服务器

socketserver可以实现和多个客户端通信（实现并发处理多个客户端请求的Socket服务端）
它是在socket的基础上进行了一层封装，也就是说底层还是调用的socket

服务器接受客户端连接请求——》实例化一个请求处理程序——》根据服务器类和请求处理程序类，调用处理方法。

例如：

基本请求程序类（BaseRequestHandler）调用方法 handle 。此方法通过属性 self.request 来访问客户端套接字

socketsever

创建socketserver的基本步骤：

首先import socketserver

创建一个请求处理类，继承 BaseRequestHandler 并且重写父类中的 handle()
在handle（）中处理和客户端所有的交互，建立链接时会自动执行handle方法

```
socketserver.TCPServer.allow_reuse_address = True  # 允许地址（端口）重用
```

实例化 TCPServer对象 ，将服务器IP/端口号和请求处理类传给 TCPServer

```
server = socketserver.ThreadingTCPServer(ip_port,MyServer)
   # 对 socketserver.ThreadingTCPServer 类实例化对象，将ip地址，端口号以及自己定义的类名传入，并返回一个对象
#多线程：ThreadingTCPServer
#多进程：ForkingTCPServer -only in Linux
```

对象执行serve_forever方法，开启服务端（handle_request()只处理一个请求）

```
server.serve_forever()
 #处理多个请求，永远执行
```

socketsever

服务端：

```
import socketserver

# 自定义类来实现通信循环
class MyServer(socketserver.BaseRequestHandler):
    # 必须写入handle方法，建立链接时会自动执行handle方法
    def handle(self):
        while True:
            data = self.request.recv(1024)
            # handle 方法通过属性 self.request 来访问客户端套接字
            print('->client:', data)
            self.request.send(data.upper())

socketserver.TCPServer.allow_reuse_address = True
server = socketserver.ThreadingTCPServer(('127.0.0.1', 8080), MyServer)
server.serve_forever()
```

socketsever

客户端：

```
import socket

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

while True:
    client.send('hello'.encode('utf-8'))
    data = client.recv(1024)
    print(data)
```

远程执行命令subprocess

- **Python可以使用subprocess下的Popen类中的封装的方法来执行命令**
  - 构造方法 popen() 创建popen类的实例化对象

  - obj = Subprocess.Popen(data,shell=True,stdout=subprocess.PIPE,stderr=subprocess.PIPE)
    - data 命令内容
    - shell = True 命令解释器，相当于调用cmd 执行指定的命令
    - stdout 正确结果丢到管道中
    - stderr 错了丢到另一个管道中
    - PIPE 将结果转移到当前进程

  - stdout.read() 可以获取命令执行的结果
  - 指定结果后会将执行结果封装到指定的对象中
  - 然后通过对象.stdout.read()获取执行命令的结果，如果不定义stdout会将结果进行标准输出

远程执行命令subprocess

```
import subprocess
obj = subprocess.Popen('dir',
                       shell=True,
                       stdout=subprocess.PIPE,
                       stderr=subprocess.PIPE,
                       )

print(obj.stdout.read().decode('gbk'))  # 正确命令
print(obj.stderr.read().decode('gbk'))  #错误命令
```

结果的编码是以当前所在的系统为准的，如果是windows，那么res.stdout.read()读出的就是GBK编码的，在接收端需要用GBK解码

远程执行命令subprocess

- 通过**socket**远程调用执行命令并返回结果：服务端

```
import socket
import subprocess
phone = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
phone.bind(('127.0.0.1', 8080))
phone.listen(5)
while 1:
    conn, client_addr = phone.accept()
    print(client_addr)
    while 1:
        try:
            cmd = conn.recv(1024)
            ret = subprocess.Popen(cmd.decode('utf-8'), shell=True, stdout=subprocess.PIPE, stderr=subprocess.PI
            correct_msg = ret.stdout.read()
            error_msg = ret.stderr.read()
            conn.send(correct_msg + error_msg)
        except ConnectionResetError:
            print("服务结束")
            break
    conn.close()
phone.close()
```

远程执行命令subprocess

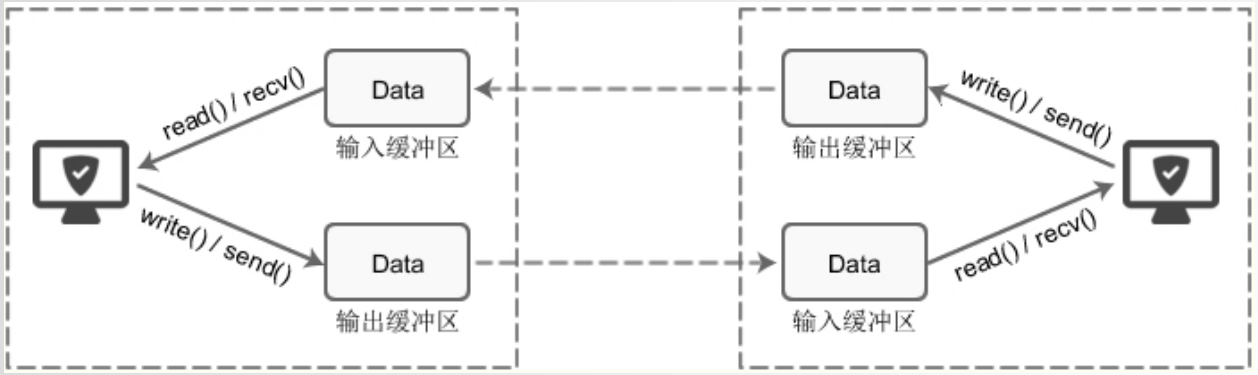
- 客户端

```
import socket
phone = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
phone.connect(('127.0.0.1', 8080))
while 1:
    cmd = input('>>>')
    if cmd == "zaijian":
        break
    phone.send(cmd.encode('utf-8'))
    from_server_data = phone.recv(1024)
    print(from_server_data.decode('gbk'))
phone.close()


#提出沾包问题
```

解决沾包问题

TCP协议是面向流的协议，容易出现粘包问题



不管是recv还是send都不是直接接收对方的数据（不是一个send一定对应一个recv），而是操作自己的缓存区（产生沾包的根本原因）
例如基于tcp的套接字客户端往服务端上传数据，发送时数据内容是按照一段一段的字节流发送的，
在接收方看了，根本不知道该文件的字节流从何处开始，在何处结束

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。

而UDP是面向消息的协议，每个UDP段都是一条消息，应用程序必须以消息为单位提取数据，
不能一次提取任意字节的数据，这一点和TCP是很不同的

只有TCP有粘包现象，UDP永远不会粘包

解决沾包问题

粘包不一定会发生

如果发生了：1.可能是在客户端已经粘了

2.客户端没有粘，可能是在服务端粘了

客户端粘包：

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，数据量很小，TCP优化算法会当做一个包发出去，产生粘包）

服务端粘包

接收方没能及时接收缓冲区的包（或没有接收完），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）

客户端沾包

```
# server端：
import socket
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(('127.0.0.1', 9904))
server.listen(5)
conn, addr = server.accept()
res1 = conn.recv(1024)
print('第一次', res1)
res2 = conn.recv(1024)
print('第二次', res2)
```

```
# client端：
import socket
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(('127.0.0.1', 9904))
client.send('hello'.encode('utf-8'))
client.send('world'.encode('utf-8'))
```

打印结果：

第一次 b'helloworld'

第二次 b''

不合适的解决方案：

send时加上时间间隔，虽然可以解决，但是会影响效率。不可取。

服务端沾包

```
# server端：
import socket
server =socket.socket(socket.AF_INET ,socket.SOCK_STREAM)
server.bind(('127.0.0.1' ,9904))
server.listen(5)

conn,addr =server.accept()
res1 = conn.recv(2)  # 第一没有接收完整
print('第一次' ,res1)
res2 =conn.recv(10)# 第二次会接收旧数据，再收取新的
print('第二次', res2)
```

不合适的解决方案：
提升recv的接收数量的上限。 不可取
因为没有上限。8G数据，一次接收8G撑爆内存了

打印结果：
第一次 b'he'
第二次 b'lloworld'

```
# client端：
import socket
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(('127.0.0.1', 9904))

client.send('hello'.encode('utf-8'))
client.send('world'.encode('utf-8'))
```

如何解决

- 问题的根源在于：接收端不知道发送端将要传送的字节流的长度
  - 所以解决粘包的方法就是发送端在发送数据前，发一个头文件包，告诉发送的字节流总大小，然后接收端来一个死循环接收完所有数据
  - 使用struct模块可以用于将Python的值根据格式符，转换为固定长度的字符串（byte类型）
  - struct模块中最重要的三个函数是pack(), unpack(), calcsize()
    - pack(fmt, v1, v2, ...)　　按照给定的格式(fmt)，把数据封装成字符串(实际上是类似于c结构体的字节流)
    - unpack(fmt, string)　　按照给定的格式(fmt)解析字节流string，返回解析出来的tuple
    - calcsize(fmt)　　计算给定的格式(fmt)占用多少字节的内存

如何解决

- 解决方法见参考代码（test/test1）