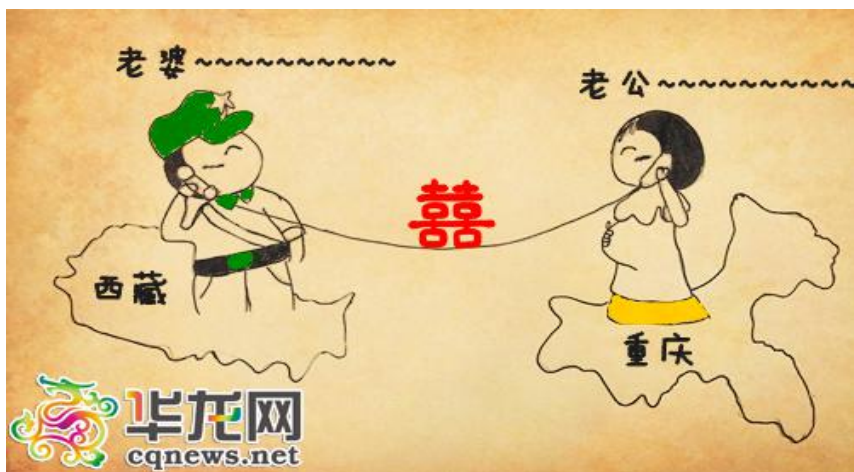


1 网络通信概述

1. 什么是网络





说明

- 网络就是一种辅助双方或者多方能够连接在一起的工具
- 如果没有网络可想单机的世界是多么的孤单

单机游戏（不能和远在他乡的朋友一起玩）





2. 使用网络的目的

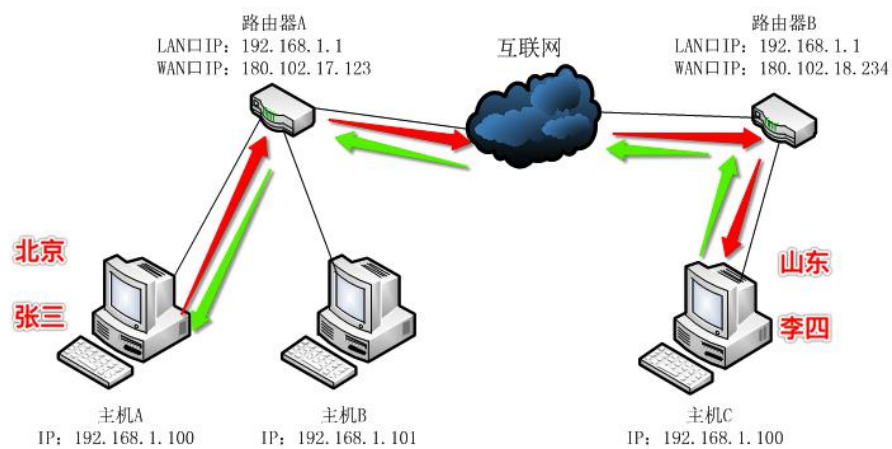
就是为了联通多方然后进行通信用的，即把数据从一方传递给另外一方

前面的学习编写的程序都是单机的，即不能和其他电脑上的程序进行通信

为了让在不同的电脑上运行的软件，之间能够互相传递数据，就需要借助网络的功能

小总结

- 使用网络能够把多方链接在一起，然后可以进行数据传递
- 所谓的网络编程就是，让在不同的电脑上的软件能够进行数据传递，即进程之间的通信



2 ip 地址

1. 什么是地址

The image shows a YTO Express shipping label with the following fields and information:

- 圆通速递 YTO EXPRESS** logo and website: www.yto.net.cn
- VIP** label
- Barcode** with number: V2 4062 5467
- 始发地 (DEPARTURE)** fields: 省 (Province), 市(县) (City), 区(镇) (Town), 邮政编码 (POSTAL CODE)
- 收件人姓名 (TO)** and **单位名称 (COMPANY NAME)**
- 收件地址 (ADDRESS)** fields: 省 (Province), 市(县) (City), 区(镇) (Town), 邮政编码 (POSTAL CODE)
- 城市 (CITY)**
- 重量 (WEIGHT)** in kg and **体积 (VOLUME)** in L, with dimensions: 长 (L), 宽 (W), 高 (H) in CM
- 付款方式 (MEANS OF PAYMENT)**: 现金 (CASH), 协议结算 (AGREEMENT)
- 资费 (CHARGE)** and **加急费 (HASTY FEE)**
- 包装费 (PACKAGING FEE)**
- 费用总计 (TOTAL)**
- 代收人签名 (AUTHORIZED SIGNATURE)** and **证件号 (ID NO.)**
- 备注 (REMARKS)**



发送 存草稿 查词典 取消

发件人

收件人

抄送

密送

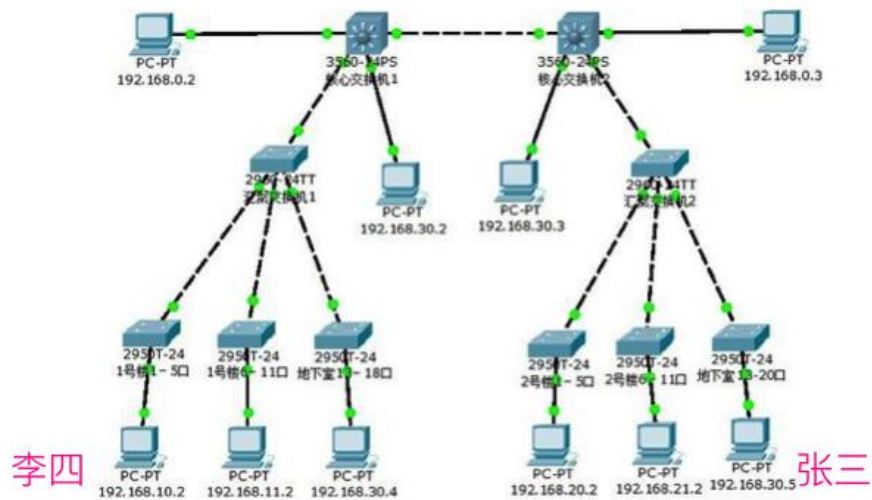
主题

添加附件 - 批量上传附件 - 网盘附件

B **I** **U** **A^A** **A_↓** | 图片 截图 表情 信纸 签名

地址就是用来标记地点的

2. ip 地址的作用



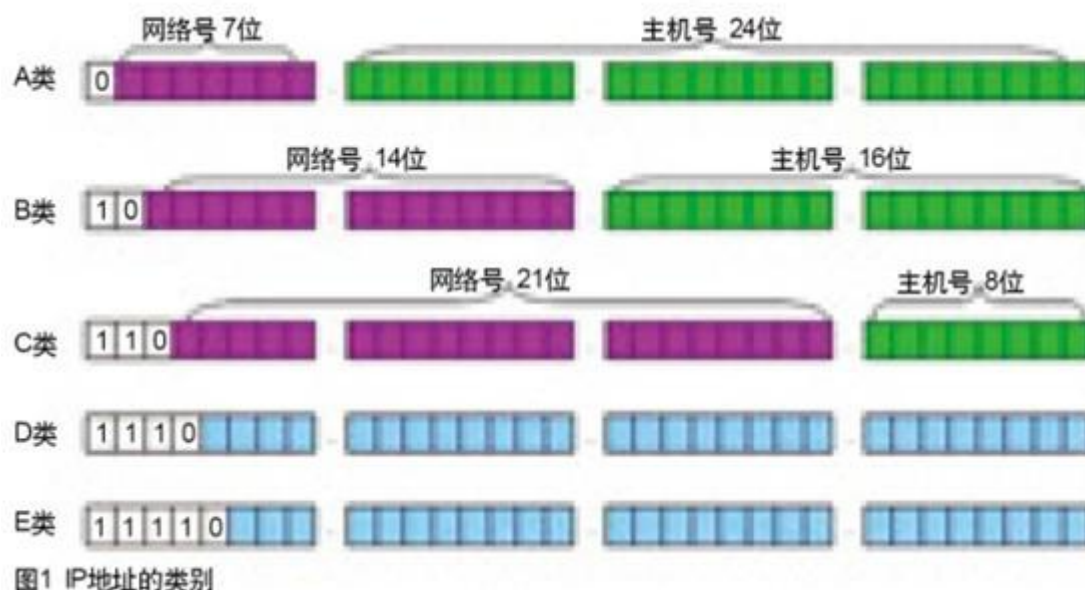
怎么传过去？

to:张三
content: 来吃晚饭

ip 地址：用来在网络中标记一台电脑，比如 192.168.1.1；在本地局域网上是唯一的。

3. ip 地址的分类（了解）

每一个 IP 地址包括两部分：网络地址和主机地址



3.1 A 类 IP 地址

一个 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

二进制表示为：00000001 00000000 00000000 00000001 - 01111110 11111111 11111111 11111110

可用的 A 类网络有 126 个，每个网络能容纳 1677214 个主机

3.2 B 类 IP 地址

一个 B 类 IP 地址由 2 个字节的网络地址和 2 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

二进制表示为：10000000 00000001 00000000 00000001 - 10111111 11111111 11111111 11111110

可用的 B 类网络有 16384 个，每个网络能容纳 65534 主机

3.3 C 类 IP 地址

一个 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

二进制表示为: 11000000 00000000 00000001 00000001 - 11011111 11111111
11111110 11111110

C 类网络可达 2097152 个，每个网络能容纳 254 个主机

3.4 D 类地址用于多点广播

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（Multicast）中

多点广播地址用来一次寻址一组计算机 s 地址范围 224.0.0.1-239.255.255.254

3.5 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

3.6 私有 ip

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，也就是属于私网 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

3.7 注意

IP 地址 127. 0. 0. 1~127. 255. 255. 255 用于回路测试，

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。

3 Linux 命令(ping, ifconfig)

查看或配置网卡信息: ifconfig

如果, 我们只是敲: ifconfig, 它会显示所有网卡的信息:

```
python@ubuntu:/home$ ifconfig
ens33  Link encap:以太网 硬件地址 00:0c:29:2b:0b:2b
       inet 地址:192.168.1.107 广播:192.168.1.255 掩码:255.255.255.0
       inet6 地址: fe80::e48b:c9c7:ea2c:78b2/64 Scope:Link
       UP BROADCAST RUNNING MULTICAST MTU:1500 跃点数:1
       接收数据包:22515 错误:0 丢弃:0 过载:0 帧数:0
       发送数据包:3830 错误:0 丢弃:0 过载:0 载波:0
       碰撞:0 发送队列长度:1000
       接收字节:7333381 (7.3 MB) 发送字节:967383 (967.3 KB)

lo      Link encap:本地环回
       inet 地址:127.0.0.1 掩码:255.0.0.0
       inet6 地址: ::1/128 Scope:Host
       UP LOOPBACK RUNNING MTU:65536 跃点数:1
       接收数据包:2924 错误:0 丢弃:0 过载:0 帧数:0
       发送数据包:2924 错误:0 丢弃:0 过载:0 载波:0
       碰撞:0 发送队列长度:1
       接收字节:855307 (855.3 KB) 发送字节:855307 (855.3 KB)
```

网卡名 (指向 ens33)

ip地址 (指向 192.168.1.107)

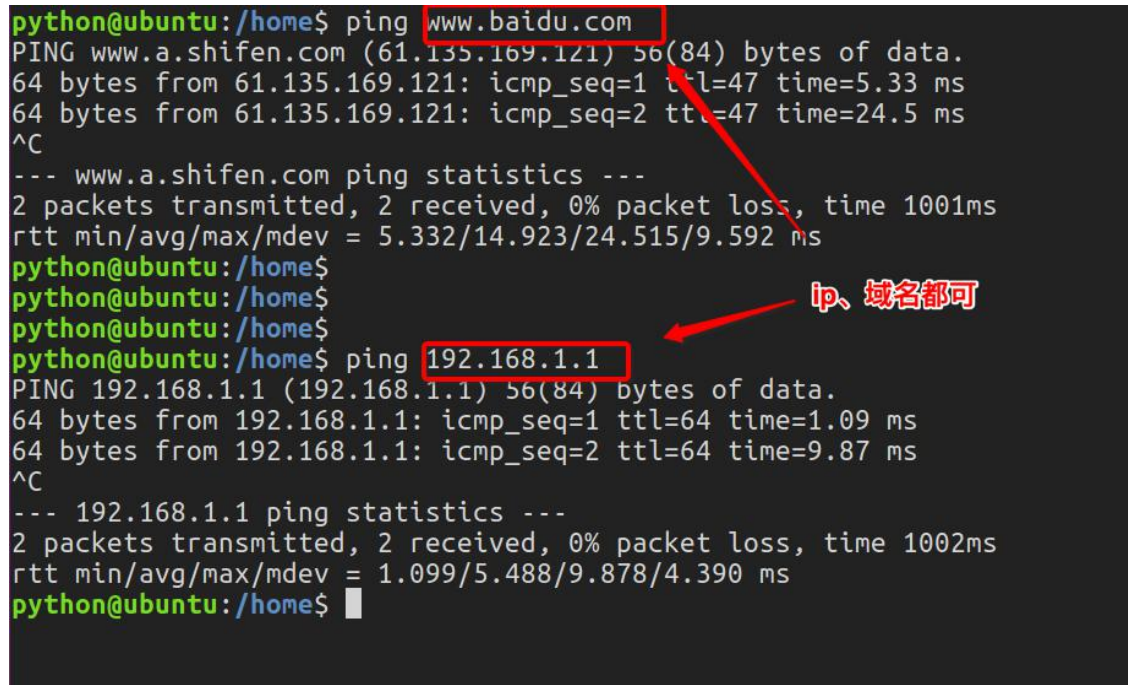
```
python@ubuntu:/home$ ifconfig ens33 192.168.1.108
SIOCSIFADDR: 不允许的操作
SIOCSIFFLAGS: 不允许的操作
python@ubuntu:/home$
python@ubuntu:/home$
python@ubuntu:/home$ sudo ifconfig ens33 192.168.1.108 修改ens33
python@ubuntu:/home$ 权限 的ip
python@ubuntu:/home$ ifconfig
ens33  Link encap:以太网 硬件地址 00:0c:29:2b:0b:2b
       inet 地址:192.168.1.108 广播:192.168.1.255 掩码:255.255.255.0
       inet6 地址: fe80::e48b:c9c7:ea2c:78b2/64 Scope:Link
       UP BROADCAST RUNNING MULTICAST MTU:1500 跃点数:1
       接收数据包:22550 错误:0 丢弃:0 过载:0 帧数:0
       发送数据包:3857 错误:0 丢弃:0 过载:0 载波:0
       碰撞:0 发送队列长度:1000
       接收字节:7336241 (7.3 MB) 发送字节:970854 (970.8 KB)

lo      Link encap:本地环回
       inet 地址:127.0.0.1 掩码:255.0.0.0
       inet6 地址: ::1/128 Scope:Host
       UP LOOPBACK RUNNING MTU:65536 跃点数:1
       接收数据包:2964 错误:0 丢弃:0 过载:0 帧数:0
       发送数据包:2964 错误:0 丢弃:0 过载:0 载波:0
       碰撞:0 发送队列长度:1
       接收字节:864523 (864.5 KB) 发送字节:864523 (864.5 KB)
```

测试远程主机连通性：ping

通常用 ping 来检测网络是否正常

```
python@ubuntu:/home$ ping www.baidu.com
PING www.a.shifen.com (61.135.169.121) 56(84) bytes of data.
64 bytes from 61.135.169.121: icmp_seq=1 ttl=47 time=5.33 ms
64 bytes from 61.135.169.121: icmp_seq=2 ttl=47 time=24.5 ms
^C
--- www.a.shifen.com ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 5.332/14.923/24.515/9.592 ms
python@ubuntu:/home$
python@ubuntu:/home$
python@ubuntu:/home$
python@ubuntu:/home$ ping 192.168.1.1
PING 192.168.1.1 (192.168.1.1) 56(84) bytes of data.
64 bytes from 192.168.1.1: icmp_seq=1 ttl=64 time=1.09 ms
64 bytes from 192.168.1.1: icmp_seq=2 ttl=64 time=9.87 ms
^C
--- 192.168.1.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 1.099/5.488/9.878/4.390 ms
python@ubuntu:/home$
```

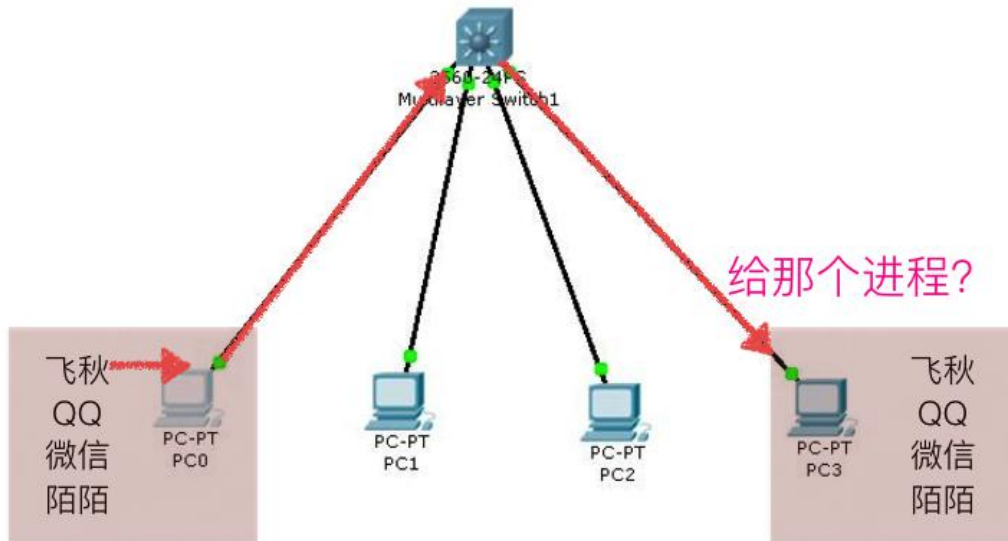


路由查看

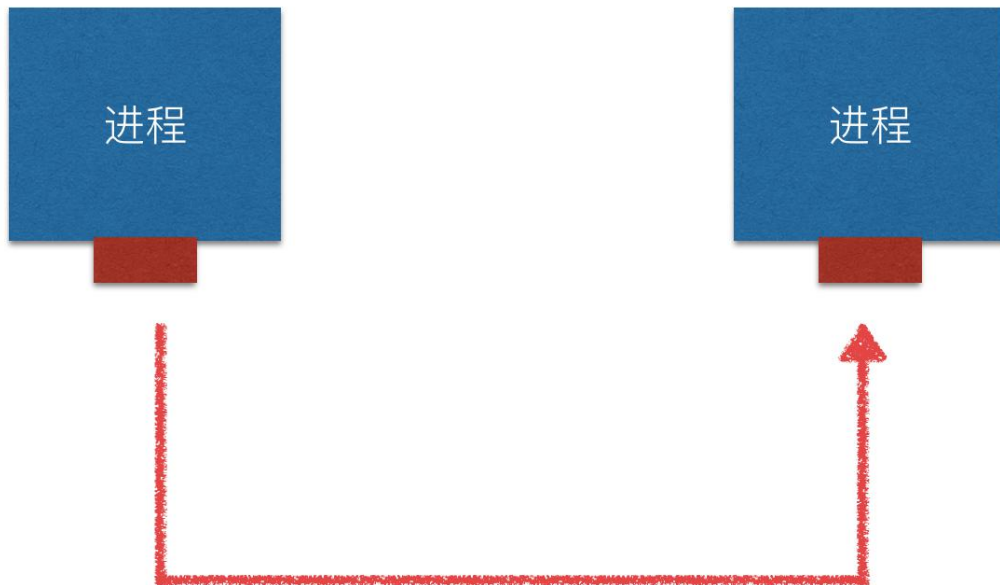
route 可以查看路由

4 端口

1. 什么是端口



端口就好一个房子的门，是出入这间房子的必经之路。



如果一个程序需要收发网络数据，那么就需要有这样的端口

在 linux 系统中，端口可以有 65536 (2 的 16 次方) 个之多！

既然有这么多，操作系统为了统一管理，所以进行了编号，这就是端口号

2. 端口号

端口是通过端口号来标记的，端口号只有整数，范围是从 0 到 65535

注意：端口数不一样的*nix 系统不一样，还可以手动修改

3. 端口是怎样分配的

端口号不是随意使用的，而是按照一定的规定进行分配。

端口的分类标准有好几种，我们这里不做详细讲解，只介绍一下知名端口和动态端口

3.1 知名端口（Well Known Ports）

知名端口是众所周知的端口号，范围从 0 到 1023

80 端口分配给 HTTP 服务

21 端口分配给 FTP 服务

可以理解为，一些常用的功能使用的号码是估计的，好比 电话号码 110、10086、10010 一样



一般情况下，如果一个程序需要使用知名端口的需要有 root 权限

3.2 动态端口（Dynamic Ports）

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程序或应用程序需要网络通信时，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。

当这个程序关闭时，同时也就释放了所占用的端口号

3.3 怎样查看端口及谁使用了端口？

- 用“**netstat -an**”查看端口状态
- **sudo lsof -i [tcp/udp]:2425** 必须是 root 才能查看
sudo lsof -i tcp:22 查看哪一个进程用了这个端口
- **ps -elf |grep udp_server** 查看某个进程是否还在

4. 小总结

端口有什么用呢？我们知道，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。需要注意的是，端口并不是一一对应的。比如你的电脑作为客户机访问一台 WWW 服务器时，WWW 服务器使用“80”端口与你的电脑通信，但你的电脑则可能使用“3457”这样的端口。

5 socket 简介

1. 不同电脑上的进程之间如何通信

首要解决的问题是如何唯一标识一个进程，否则通信无从谈起！

在 1 台电脑上可以通过进程号（PID）来唯一标识一个进程，但是在网络中这是行不通的。

其实 TCP/IP 协议族已经帮我们解决了这个问题，网络层的“ip 地址”可以唯一标识网络中的主机，而传输层的“协议+端口”可以唯一标识主机中的应用进程（进程）。

这样利用协议，ip 地址，端口就可以标识网络的进程了，网络中的进程通信就可以利用这个标志与其它进程进行交互

注意：

- 所谓进程指的是：运行的程序以及运行时用到的资源这个整体称之为进程（在讲解多任务编程时进行详细讲解）
- 所谓进程间通信指的是：运行的程序之间的数据共享
- 后面课程中会详细说到，像网络层等知识，不要着急

2. 什么是 socket

socket(简称 套接字) 是进程间通信的一种方式，它与其他进程间通信的一个主要不同是：

它能实现不同主机间的进程间通信，我们网络上各种各样的服务大多都是基于 Socket 来完成通信的

例如我们每天浏览网页、QQ 聊天、收发 email 等等



微信



米聊



易信



陌陌



来往



旺信



百度



傲游



UC



谷歌



火狐



搜狗



QQ



360



欧朋



海豚

3. 创建 socket

在 Python 中使用 socket 模块的函数 `socket` 就可以完成：

```
import socket
socket.socket(AddressFamily, Type)
```

说明：

函数 `socket.socket` 创建一个 socket，该函数带有两个参数：

- **Address Family:** 可以选择 `AF_INET`（用于 Internet 进程间通信）或者 `AF_UNIX`（用于同一台机器进程间通信），实际工作中常用 `AF_INET`
- **Type:** 套接字类型，可以是 `SOCK_STREAM`（流式套接字，主要用于 TCP 协议）或者 `SOCK_DGRAM`（数据报套接字，主要用于 UDP 协议）

创建一个 tcp socket（tcp 套接字）

```
import socket

# 创建 tcp 的套接字
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# ...这里是使用套接字的功能（省略）...

# 不用的时候，关闭套接字
s.close()
```

创建一个 udp socket（udp 套接字）

```
import socket

# 创建 udp 的套接字
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# ...这里是使用套接字的功能（省略）...

# 不用的时候，关闭套接字
s.close()
```

说明

- 套接字使用流程 与 文件的使用流程很类似
 - a. 创建套接字
 - b. 使用套接字收/发数据

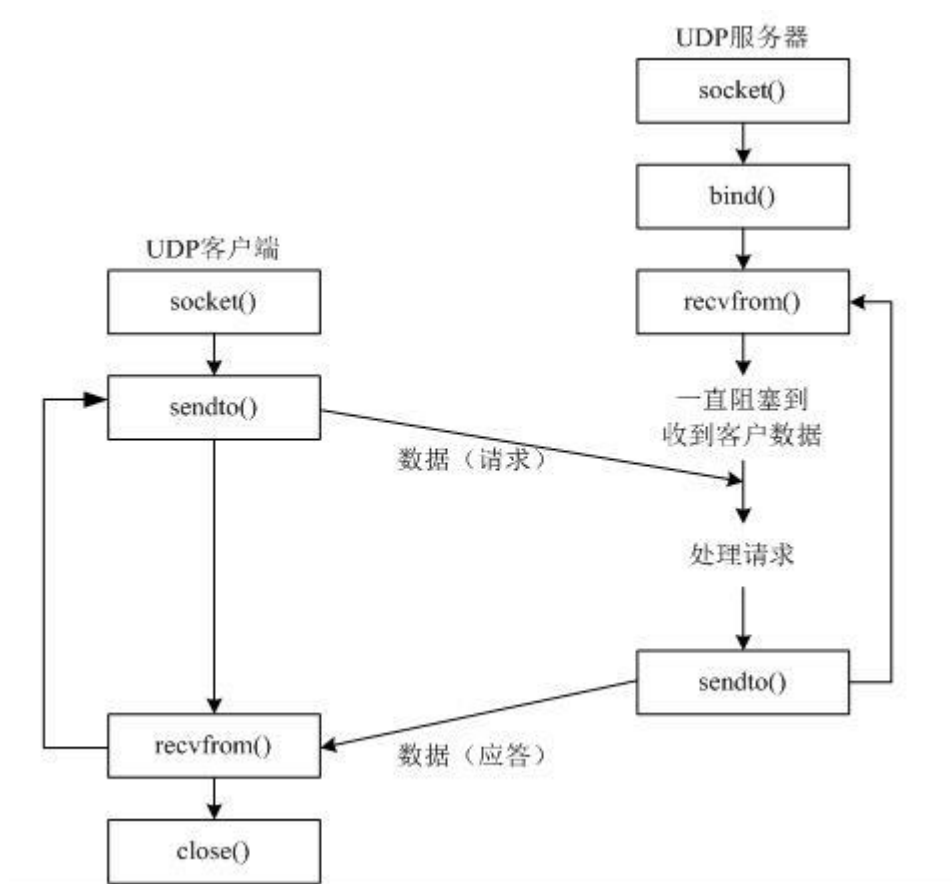
c. 关闭套接字

6 udp 网络程序-发送、接收数据

1. udp 网络程序-发送数据

创建一个基于 udp 的网络程序流程很简单，具体步骤如下：

2. 创建客户端套接字
3. 发送/接收数据
4. 关闭套接字



代码如下：

```
#coding=utf-8
```

```
from socket import *
```

```
# 1. 创建 udp 套接字
```

```
udp_socket = socket(AF_INET, SOCK_DGRAM)
```

```
# 2. 准备接收方的地址
# '192.168.1.103'表示目的 ip 地址
# 8080 表示目的端口
dest_addr = ('192.168.1.103', 8080) # 注意 是元组, ip 是字符串, 端口是数字

# 3. 从键盘获取数据
send_data = input("请输入要发送的数据:")

# 4. 发送数据到指定的电脑上的指定程序中
udp_socket.sendto(send_data.encode('utf-8'), dest_addr)

# 5. 关闭套接字
udp_socket.close()
```

运行现象:

在 Ubuntu 中运行脚本:



```
python@ubuntu:~/Desktop$ python3 01.py
请输入要发送的数据:nihao
```

2. udp 网络程序-发送、接收数据（客户端）

```
#coding=utf-8
```

```
from socket import *
```

```
# 1. 创建 udp 套接字
udp_socket = socket(AF_INET, SOCK_DGRAM)

# 2. 准备接收方的地址
dest_addr = ('192.168.236.129', 8080)

# 3. 从键盘获取数据
send_data = input("请输入要发送的数据:")

# 4. 发送数据到指定的电脑上
udp_socket.sendto(send_data.encode('utf-8'), dest_addr)

# 5. 等待接收对方发送的数据
recv_data = udp_socket.recvfrom(1024) # 1024 表示本次接收的最大字节数
```

```
# 6. 显示对方发送的数据
# 接收到的数据 recv_data 是一个元组
# 第 1 个元素是对方发送的数据
# 第 2 个元素是对方的 ip 和端口
print(recv_data[0].decode('gbk'))
print(recv_data[1])
```

```
# 7. 关闭套接字
udp_socket.close()
```

python 脚本:



```
python@ubuntu:~/Desktop$ python3 02.py
请输入要发送的数据:你好 啊
可以的
('192.168.236.129', 8080)
python@ubuntu:~/Desktop$
```

出错信息:

bind 使用, 如果端口被占用

OSError: [Errno 98] Address already in use

7 udp 绑定信息

1. udp 网络程序-端口问题

- 会变的端口号

重新运行多次脚本, 会发现客户端端口改变了

说明：

- 每重新运行一次网络程序，上图中红圈中的数字，不一样的原因在于，这个数字标识这个网络程序，当重新运行时，如果没有确定到底用哪个，**系统默认会随机分配**
- 记住一点：这个网络程序在运行的过程中，这个就唯一标识这个程序，所以如果其他电脑上的网络程序如果想要向此程序发送数据，那么就需要向这个数字（即端口）标识的程序发送即可

2. udp 绑定信息---服务器端

<1>. 绑定信息

一般情况下，在一台电脑上运行的网络程序有很多，为了不与其他网络程序占用同一个端口号，往往在编程中，udp 的端口号一般不绑定

但是如果需要做成一个服务器端的程序的话，是需要绑定的，想想看这又是为什么呢？

如果报警电话每天都在变，想必世界就会乱了，所以一般服务性的程序，往往需要一个固定的端口号，这就是所谓的端口绑定



<2>. 绑定示例

```
#coding=utf-8
```

```
from socket import *
```

```
# 1. 创建套接字
```

```
udp_socket = socket(AF_INET, SOCK_DGRAM)
```

```
# 2. 绑定本地的相关信息，如果一个网络程序不绑定，则系统会随机分配
```

```
local_addr = ('', 7788) # ip 地址和端口号，ip 一般不用写，表示本机的任何一个 ip
```

```
udp_socket.bind(local_addr)
```


3. 等待接收对方发送的数据

```
recv_data = udp_socket.recvfrom(1024) # 1024 表示本次接收的最大字节数
```

4. 显示接收到的数据

```
print(recv_data[0].decode('utf8'))
```

5. 关闭套接字

```
udp_socket.close()
```

运行结果:

我是女生

(**'192.168.3.28', 51404**)

<3>. 总结

- 一个 udp 网络程序，可以不绑定，此时操作系统会随机进行分配一个端口，如果重新运行此程序端口可能会发生变化
- 一个 udp 网络程序，也可以绑定信息（ip 地址，端口号），如果绑定成功，那么操作系统用这个端口号来进行区别收到的网络数据是否是此进程的
端口占用怎么办？
这里讲解 kill 命令

8 网络通信过程(简单版)

发送手机方



购买手机方



说明

- 网络通信过程中，之所需要 ip、port 等，就是为了能够将一个复杂的通信过程进行任务划分，从而保证数据准确无误的传递

python3 编码转换

str->bytes:encode 编码

bytes->str:decode 解码

字符串通过编码成为字节码，字节码通过解码成为字符串。

```
>>> text = '我是文本'
>>> text
'我是文本'
>>> print(text)
我是文本
>>> bytesText = text.encode()
>>> bytesText
b'\xe6\x88\x91\xe6\x98\xaf\xe6\x96\x87\xe6\x9c\xac'
>>> print(bytesText)
```

```
b'\xe6\x88\x91\xe6\x98\xaf\xe6\x96\x87\xe6\x9c\xac'
>>> type(text)
<class 'str'>
>>> type(bytesText)
<class 'bytes'>
>>> textDecode = bytesText.decode()
>>> textDecode
'我是文本'
>>> print(textDecode)
我是文本
```

其中 `decode()` 与 `encode()` 方法可以接受参数，其声明分别为：

```
bytes.decode(encoding="utf-8", errors="strict")
str.encode(encoding="utf-8", errors="strict")
```

其中的 `encoding` 是指在解码编码过程中使用的编码(此处指“编码方案”是名词)，`errors` 是指错误的处理方案。

详细的可以参照官方文档：

- `str.encode()`
- `bytes.decode()`

9 应用：udp 聊天器

说明

- 在一个电脑中编写 1 个程序，有 2 个功能

- 1.获取键盘数据，并将其发送给对方
- 2.接收数据并显示
- 并且功能数据进行选择以上的 2 个功能调用

要求

5. 实现上述程序

参考代码

```
import socket
```

```
def send_msg(udp_socket):
    """获取键盘数据，并将其发送给对方"""
    # 1. 从键盘输入数据
    msg = input("\n 请输入要发送的数据:")
    # 2. 输入对方的 ip 地址
    dest_ip = input("\n 请输入对方的 ip 地址:")
    # 3. 输入对方的 port
    dest_port = int(input("\n 请输入对方的 port:"))
    # 4. 发送数据
    udp_socket.sendto(msg.encode("utf-8"), (dest_ip, dest_port))

def recv_msg(udp_socket):
    """接收数据并显示"""
    # 1. 接收数据
    recv_msg = udp_socket.recvfrom(1024)
    # 2. 解码
    recv_ip = recv_msg[1]
    recv_msg = recv_msg[0].decode("utf-8")
    # 3. 显示接收到的数据
    print(">>>%s:%s" % (str(recv_ip), recv_msg))

def main():
    # 1. 创建套接字
    udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    # 2. 绑定本地信息
    udp_socket.bind("", 7890)
    while True:
        # 3. 选择功能
        print("="*30)
        print("1:发送消息")
        print("2:接收消息")
        print("="*30)
```

```
op_num = input("请输入要操作的功能序号:")

# 4. 根据选择调用相应的函数
if op_num == "1":
    send_msg(udp_socket)
elif op_num == "2":
    recv_msg(udp_socket)
else:
    print("输入有误，请重新输入...")

if __name__ == "__main__":
    main()
```

想一想

- 以上的程序如果选择了接收数据功能，并且此时没有数据，程序会堵塞在这，那么怎样才能让这个程序收发数据一起进行呢？别着急，学习完多任务知识之后就解决了 $O(n_n)O...$

10 TCP 简介

TCP 介绍

TCP 协议，传输控制协议（英语：**Transmission Control Protocol**，缩写为**TCP**）是一种面向连接的、可靠的、基于字节流的传输层通信协议，由 IETF 的 RFC 793 定义。

TCP 通信需要经过创建连接、数据传送、终止连接三个步骤。

TCP 通信模型中，在通信开始之前，一定要先建立相关的链接，才能发送数据，类似于生活中，"打电话"



TCP 特点

1. 面向连接

通信双方必须先建立连接才能进行数据的传输，双方都必须为该连接分配必要的系统内核资源，以管理连接的状态和连接上的传输。

双方间的数据传输都可以通过这一个连接进行。

完成数据交换后，双方必须断开此连接，以释放系统资源。

这种连接是一对一的，因此 TCP 不适用于广播的应用程序，基于广播的应用程序请使用 UDP 协议。

2. 可靠传输

1) TCP 采用发送应答机制

TCP 发送的每个报文段都必须得到接收方的**应答**才认为这个 TCP 报文段传输成功

2) 超时重传

发送端发出一个报文段之后就启动定时器，如果在定时时间内没有收到应答就重新发送这个报文段。

TCP 为了保证不发生丢包，就给每个包一个**序列号**，同时序列号也保证了传送到接收端实体的包的**按序接收**。然后接收端实体对已成功收到的包发回一个相应的确认（ACK）；如果发送端实体在合理的往返时延（RTT）内未收到确认，那么对应的数据包就被假设为已丢失将会被进行重传。

3) 错误校验

TCP 用一个校验和函数来检验数据是否有错误；在发送和接收时都要计算校验和。

4) 流量控制和阻塞管理

滑动窗口：控制双方的发送接收速度

流量控制用来避免主机发送得过快而使接收方来不及完全收下。

TCP 与 UDP 的不同点

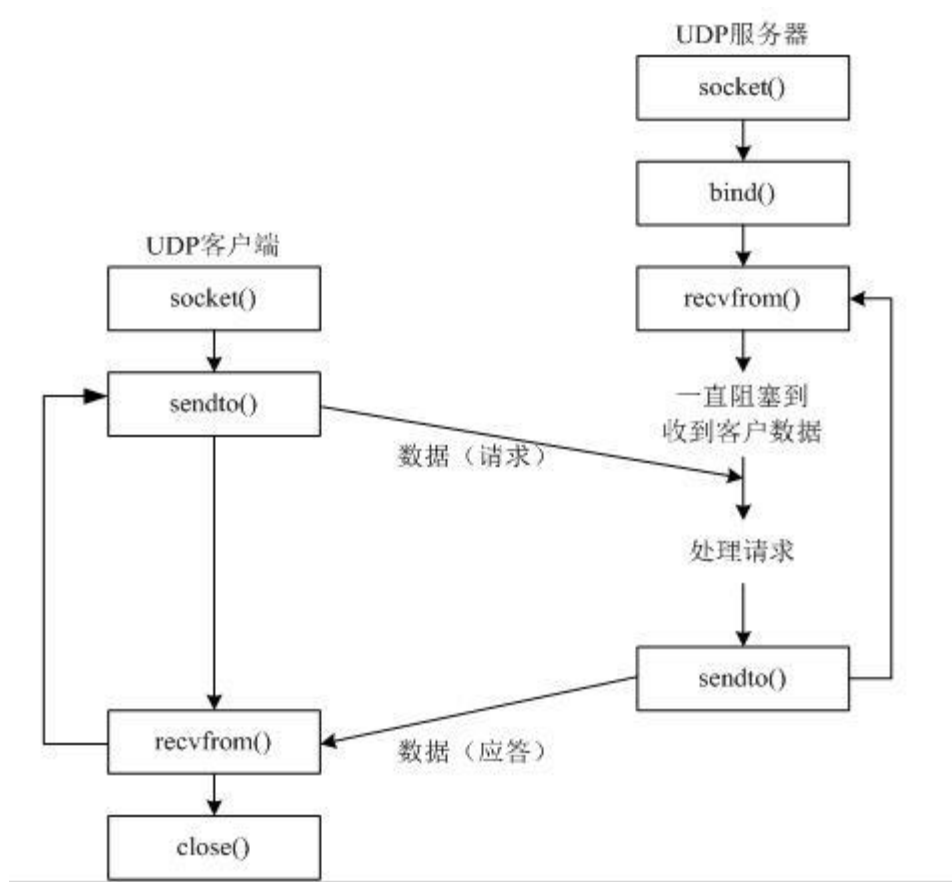
- 面向连接（确认有创建三方交握，连接已创建才作传输。）
- 有序数据传输
- 重发丢失的数据包
- 舍弃重复的数据包

- 无差错的数据传输
- 阻塞/流量控制

udp 通信模型

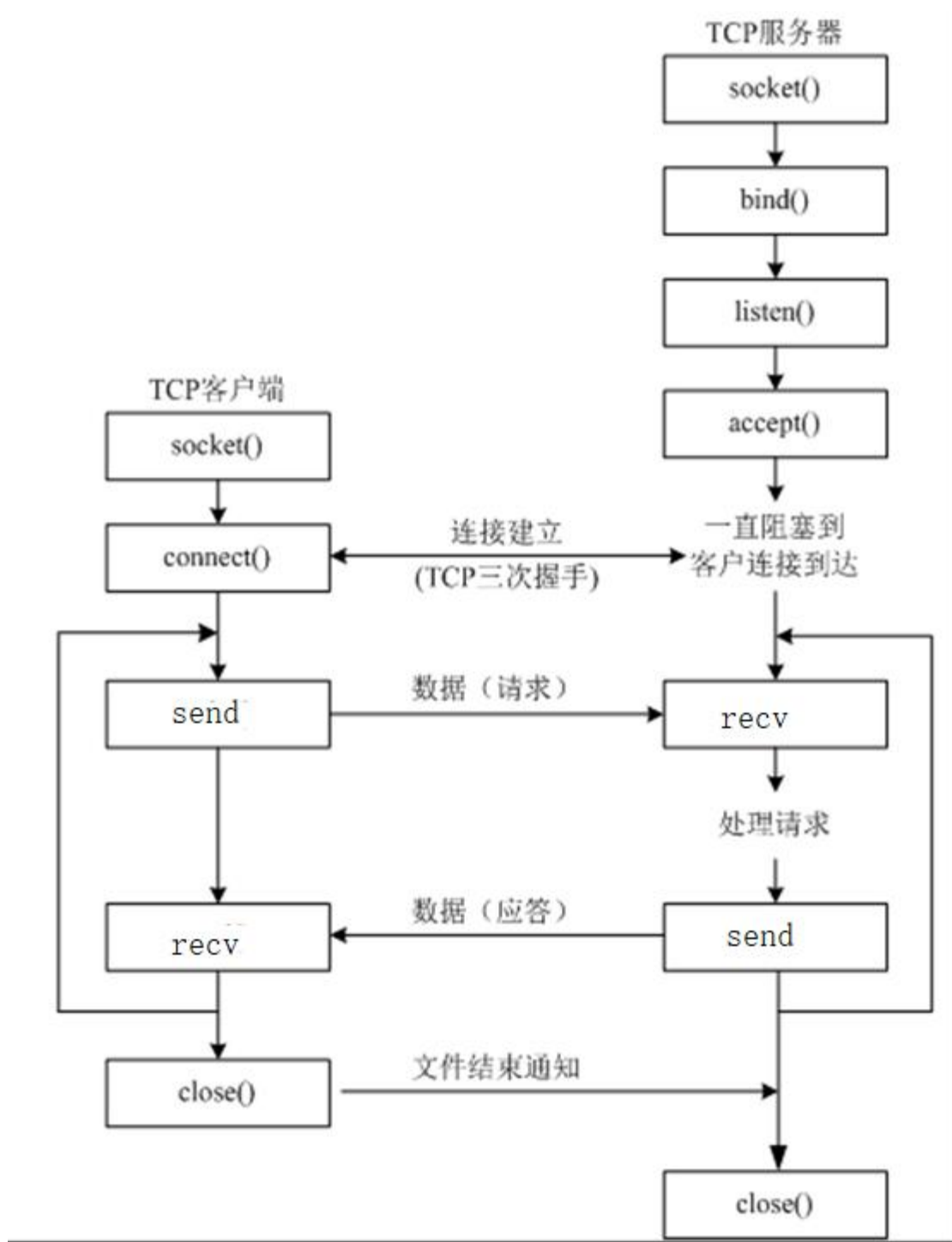
udp 通信模型中，在通信开始之前，不需要建立相关的链接，只需要发送数据即可，类似于生活中，"写信"





TCP 通信模型

udp 通信模型中，在通信开始之前，一定要先建立相关的链接，才能发送数据，类似于生活中，"打电话"



2 tcp 客户端

tcp 客户端，并不是像之前一个段子：一个顾客去饭馆吃饭，这个顾客要点菜，就问服务员咱们饭店有客户端么，然后这个服务员非常客气的说道：先生 我们饭店不用客户端，我们直接送到您的餐桌上

如果，不学习网络的知识是不是 说不定也会发生那样的笑话，哈哈

所谓的服务器端：就是提供服务的一方，而客户端，就是需要被服务的一方

tcp 客户端构建流程

tcp 的客户端要比服务器端简单很多，如果说服务器端是需要自己买手机、查手机卡、设置铃声、等待别人打电话流程的话，那么客户端就只需要找一个电话亭，拿起电话拨打即可，流程要少很多

示例代码：

```
from socket import *

# 创建 socket
tcp_client_socket = socket(AF_INET, SOCK_STREAM)

# 目的信息
server_ip = input("请输入服务器 ip:")
server_port = int(input("请输入服务器 port:"))

# 链接服务器
tcp_client_socket.connect((server_ip, server_port))

# 提示用户输入数据
send_data = input("请输入要发送的数据: ")

tcp_client_socket.send(send_data.encode("gbk"))

# 接收对方发送过来的数据，最大接收 1024 个字节
recvData = tcp_client_socket.recv(1024)
print('接收到的数据为:', recvData.decode('gbk'))

# 关闭套接字
tcp_client_socket.close()
```


运行流程:

<1>tcp 客户端

请输入服务器 ip:10.10.0.47

请输入服务器 port:8080

请输入要发送的数据: 你好啊

接收到的数据为: 我很好, 你呢

3 tcp 服务器

生活中的电话机

如果想让别人能更够打通咱们的电话获取相应服务的话, 需要做以下几件事情:

1. 买个手机
2. 插上手机卡
3. 设计手机为正常接听状态 (即能够响铃)
4. 静静的等着别人拨打

tcp 服务器

如同上面的电话机过程一样, 在程序中, 如果想要完成一个 tcp 服务器的功能, 需要的流程如下:

1. socket 创建一个套接字
2. bind 绑定 ip 和 port
3. listen 使套接字变为可以被动链接
4. accept 等待客户端的连接
5. recv/send 接收发送数据

一个很简单的 tcp 服务器如下:

```
from socket import *
```

```
# 创建 socket
```

```
tcp_server_socket = socket(AF_INET, SOCK_STREAM)
```

```
# 本地信息
address = ('', 7788)

# 绑定
tcp_server_socket.bind(address)

# 使用 socket 创建的套接字默认的属性是主动的，使用 listen 将其变为被动的，这样
# 就可以接收别人的链接了
tcp_server_socket.listen(128)

# 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专门为这个客户端服务
# client_socket 用来为这个客户端服务
# tcp_server_socket 就可以省下来专门等待其他新客户端的链接
client_socket, clientAddr = tcp_server_socket.accept()

# 接收对方发送过来的数据
recv_data = client_socket.recv(1024) # 接收 1024 个字节
print('接收到的数据为:', recv_data.decode('gbk'))

# 发送一些数据到客户端
client_socket.send("thank you !".encode('gbk'))

# 关闭为这个客户端服务的套接字，只要关闭了，就意味着不能再为这个客户端服务
# 了，如果还需要服务，只能再次重新连接
client_socket.close()
```

运行流程：

<1>tcp 服务器

接收到的数据为：你好吗

4 tcp 注意点

1. tcp 服务器需要绑定，否则客户端找不到这个服务器
2. tcp 客户端不绑定，因为是主动链接服务器，所以只要确定好服务器的 ip、port 等信息就好，本地客户端可以随机
3. tcp 服务器中通过 listen 可以将 socket 创建出来的主动套接字变为被动的，这是做 tcp 服务器时必须要做的
4. 当客户端需要链接服务器时，就需要使用 connect 进行链接，udp 是不需要链接的而是直接发送，但是 tcp 必须先链接，只有链接成功才能通信

5. 当一个 tcp 客户端连接服务器时，服务器端会有 1 个新的套接字，这个套接字用来标记这个客户端，单独为这个客户端服务
6. listen 后的套接字是被动套接字，用来接收新的客户端的连接请求的，而 accept 返回的新套接字是标记这个新客户端的
7. 关闭 listen 后的套接字意味着被动套接字关闭了，会导致新的客户端不能够链接服务器，但是之前已经链接成功的客户端正常通信。
8. 关闭 accept 返回的套接字意味着这个客户端已经服务完毕
9. 当客户端的套接字调用 close 后，服务器端会 recv 解堵塞，并且返回的长度为 0，因此服务器可以通过返回数据的长度来区别客户端是否已经下线

5 案例:文件下载器

服务器 参考代码如下:

```
from socket import *
import sys

def get_file_content(file_name):
    """获取文件的内容"""
    try:
        with open(file_name, "rb") as f:
            content = f.read()
        return content
    except:
        print("没有下载的文件:%s" % file_name)

def main():

    if len(sys.argv) != 2:
```

```
        print("请按照如下方式运行: python3 xxx.py 7890")
        return
    else:
        # 运行方式为 python3 xxx.py 7890
        port = int(sys.argv[1])

    # 创建 socket
    tcp_server_socket = socket(AF_INET, SOCK_STREAM)
    # 本地信息
    address = ('', port)
    # 绑定本地信息
    tcp_server_socket.bind(address)
    # 将主动套接字变为被动套接字
    tcp_server_socket.listen(128)

    while True:
        # 等待客户端的链接, 即为这个客户端发送文件
        client_socket, clientAddr = tcp_server_socket.accept()
        # 接收对方发送过来的数据
        recv_data = client_socket.recv(1024) # 接收 1024 个字节
        file_name = recv_data.decode("utf-8")
        print("对方请求下载的文件名为:%s" % file_name)
        file_content = get_file_content(file_name)
        # 发送文件的数据给客户端
        # 因为获取打开文件时是以 rb 方式打开, 所以 file_content 中的数据已经是
        # 二进制的格式, 因此不需要 encode 编码
        if file_content:
            client_socket.send(file_content)
        # 关闭这个套接字
        client_socket.close()

    # 关闭监听套接字
    tcp_server_socket.close()

if __name__ == "__main__":
    main()
```

客户端 参考代码如下:

```
from socket import *
```

```
def main():
```

```
    # 创建 socket
```

```
tcp_client_socket = socket(AF_INET, SOCK_STREAM)

# 目的信息
server_ip = input("请输入服务器 ip:")
server_port = int(input("请输入服务器 port:"))

# 链接服务器
tcp_client_socket.connect((server_ip, server_port))

# 输入需要下载的文件名
file_name = input("请输入要下载的文件名: ")

# 发送文件下载请求
tcp_client_socket.send(file_name.encode("utf-8"))

# 接收对方发送过来的数据，最大接收 1024 个字节（1K）
recv_data = tcp_client_socket.recv(1024)
# print('接收到的数据为:', recv_data.decode('utf-8'))
# 如果接收到数据再创建文件，否则不创建
if recv_data:
    with open("[接收]"+file_name, "wb") as f:
        f.write(recv_data)

# 关闭套接字
tcp_client_socket.close()

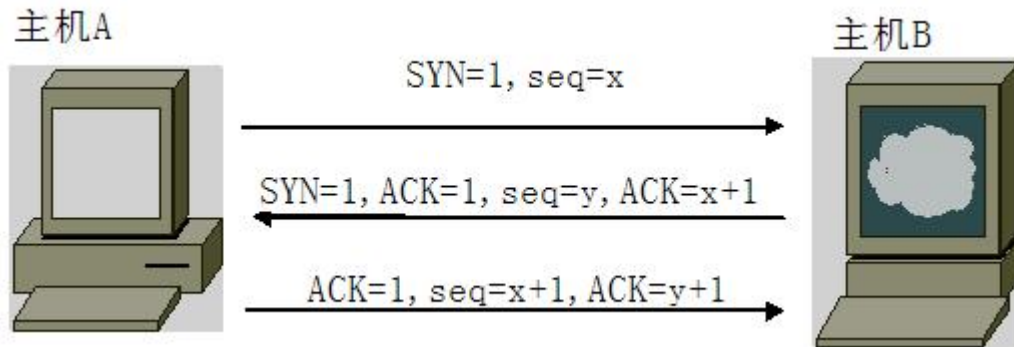
if __name__ == "__main__":
    main()
```

基本协议设计

Client send 文件名 ---》recv 接文件内容---》内容写入文件

Server recv 接文件名---》读文件内容---》send 发文件内容

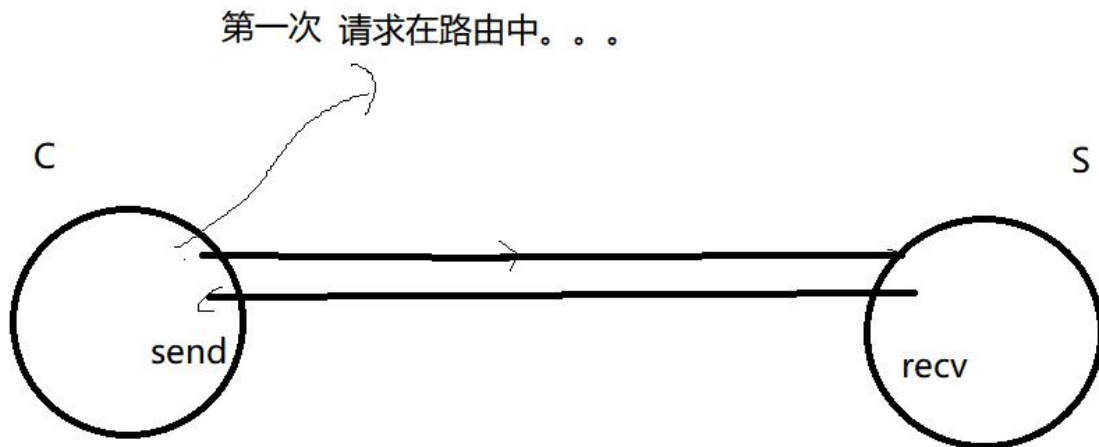
6 tcp 的 3 次握手



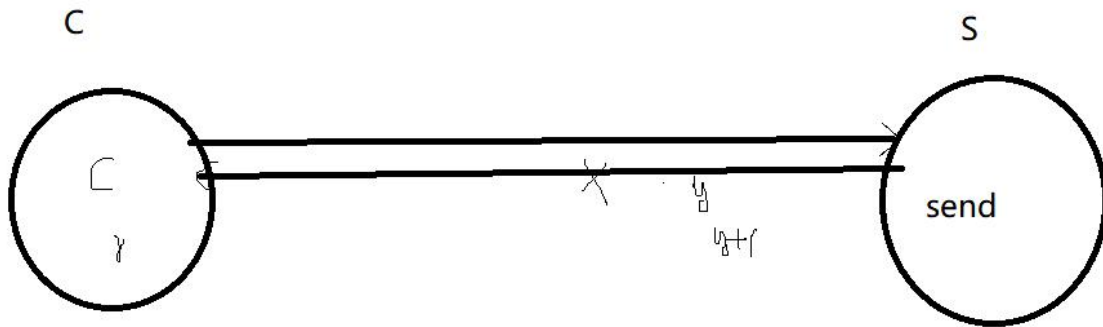
比如是 A(client)机要连到 B(server)机，结果发送的连接信息由于某种原因没有到达 B 机；

于是，A 机又发了一次，结果这次 B 收到了，于是就发信息回来，两机就连接。传完东西后，断开。

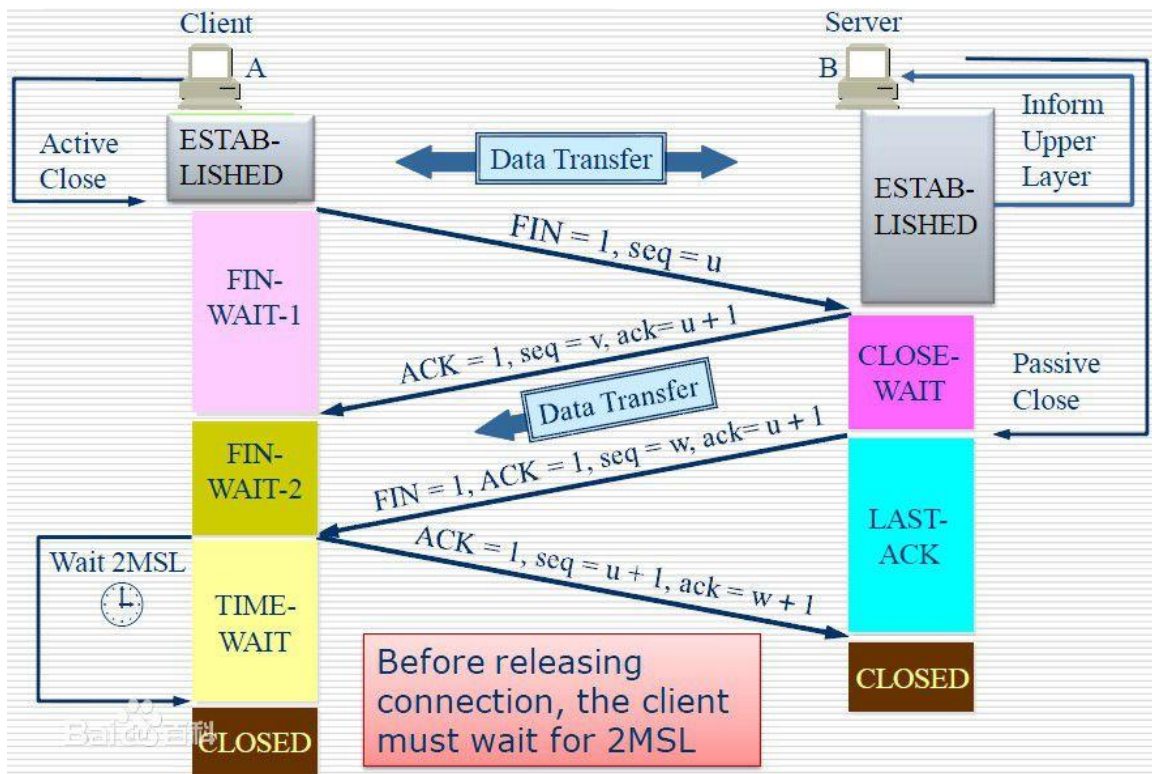
结果这时候，原先没有到达的连接信息突然又传到了 B 机，于是 B 机发信息给 A，然后 B 机就以为和 A 连上了，这个时候 B 机就在等待 A 传东西过去。永远的等待在 `recv` 接口上



考虑计算机 C 和 S 之间的通信，假定 C 给 S 发送一个连接请求分组，S 收到了这个分组，并发送了确认应答分组。按照两次握手的协定，S 认为连接已经成功地建立了，可以开始发送数据分组。可是，C 在 S 的应答分组在传输中被丢失的情况下，将不知道 S 是否已准备好，不知道 S 建议什么样的序列号，C 甚至怀疑 S 是否收到自己的连接请求分组。在这种情况下，C 认为连接还未建立成功，将忽略 S 发来的任何数据分组，只等待连接确认应答分组。而 S 在发出的数据分组超时后，重复发送同样的分组。这样就形成了死锁



7 tcp 的 4 次挥手

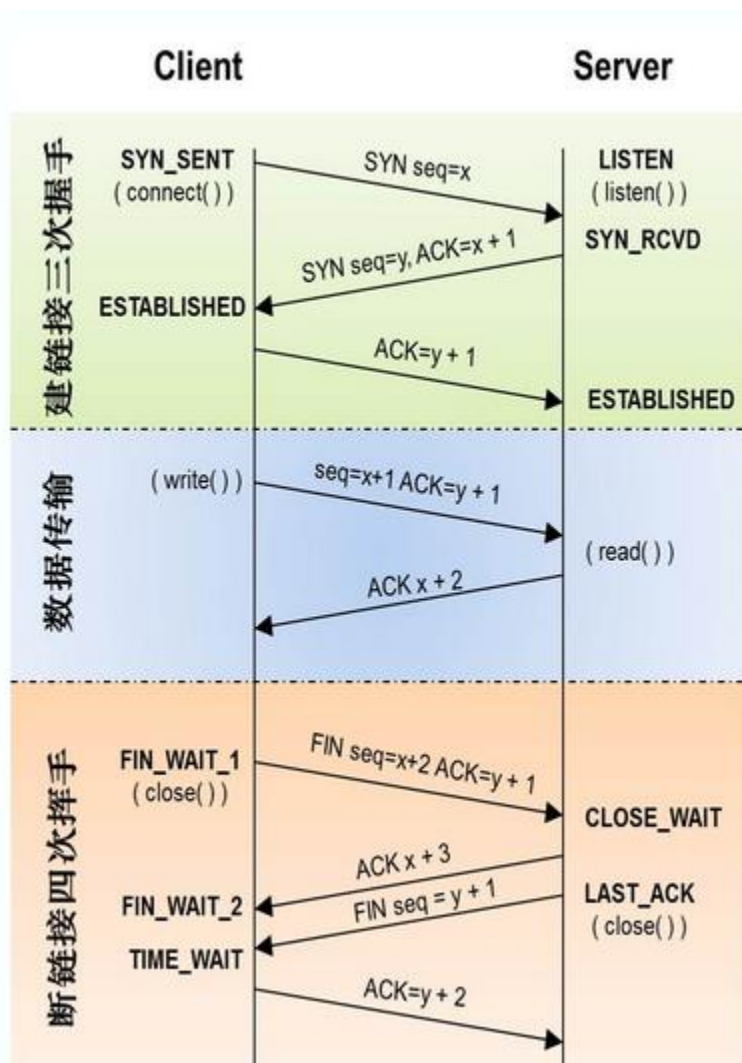


有 4 个缓冲区，所以需要 4 个缓冲区

8 tcp 长连接和短连接

TCP 在真正的读写操作之前，server 与 client 之间必须建立一个连接，当读写操作完成后，双方不再需要这个连接时它们可以释放这个连接，连接的建立通过三次握手，释放则需要四次握手，所以说每个连接的建立都是需要资源消耗和时间消耗的。

TCP 通信的整个过程，如下图：



1. TCP 短连接

模拟一种 TCP 短连接的情况：

1. client 向 server 发起连接请求
2. server 接到请求，双方建立连接
3. client 向 server 发送消息
4. server 回应 client
5. 一次读写完成，此时双方任何一个都可以发起 close 操作

在步骤 5 中，一般都是 client 先发起 close 操作。当然也不排除有特殊的情况。

从上面的描述看，短连接一般只会在 client/server 间传递一次读写操作！

2. TCP 长连接

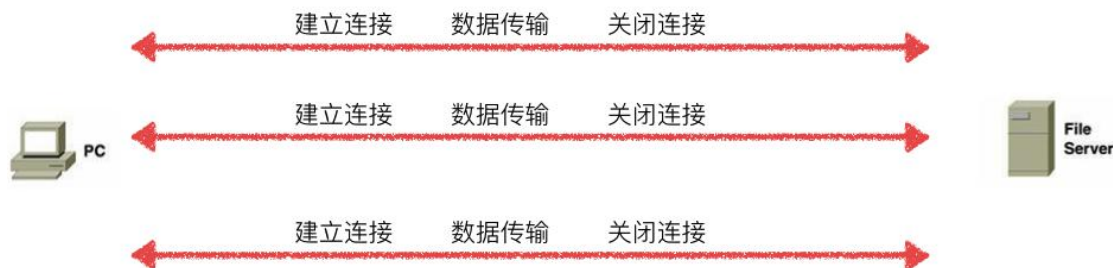
再模拟一种长连接的情况:

1. client 向 server 发起连接
2. server 接到请求, 双方建立连接
3. client 向 server 发送消息
4. server 回应 client
5. 一次读写完成, 连接不关闭
6. 后续读写操作...
7. 长时间操作之后 client 发起关闭请求

3. TCP 长/短连接操作过程

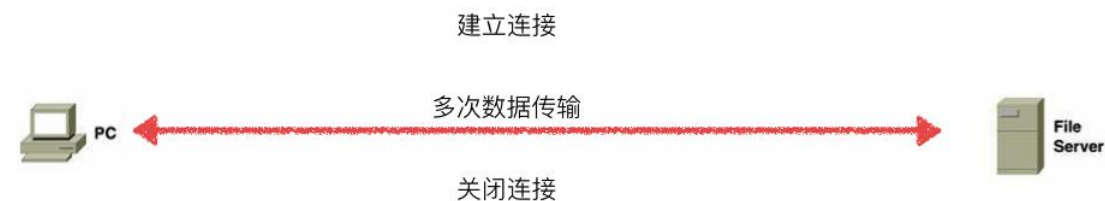
3.1 短连接的操作步骤是:

建立连接——数据传输——关闭连接...建立连接——数据传输——关闭连接



3.2 长连接的操作步骤是:

建立连接——数据传输...（保持连接）...数据传输——关闭连接



4. TCP 长/短连接的优点和缺点

- 长连接可以省去较多的 TCP 建立和关闭的操作, 减少浪费, 节约时间。

对于频繁请求资源的客户来说，较适用长连接。

- **client** 与 **server** 之间的连接如果一直不关闭的话，会存在一个问题，

随着客户端连接越来越多，**server** 早晚有扛不住的时候，这时候 **server** 端需要采取一些策略，

如关闭一些长时间没有读写事件发生的连接，这样可以避免一些恶意连接导致 **server** 端服务受损；

如果条件再允许就可以以客户端机器为颗粒度，限制每个客户端的最大长连接数，

这样可以完全避免某个蛋疼的客户端连累后端服务。

- 短连接对于服务器来说管理较为简单，存在的连接都是有用的连接，不需要额外的控制手段。
- 但如果客户请求频繁，将在 **TCP** 的建立和关闭操作上浪费时间和带宽。

5. TCP 长/短连接的应用场景

- 长连接多用于操作频繁，点对点的通讯，而且连接数不能太多情况。

每个 **TCP** 连接都需要三次握手，这需要时间，如果每个操作都是先连接，再操作的话那么处理速度会降低很多，所以每个操作完后都不断开，再次处理时直接发送数据包就 **OK** 了，不用建立 **TCP** 连接。

例如：数据库的连接用长连接，如果用短连接频繁的通信会造成 **socket** 错误，

而且频繁的 **socket** 创建也是对资源的浪费。

- 而像 **WEB** 网站的 **http** 服务一般都用短链接，因为长连接对于服务端来说会耗费一定的资源，

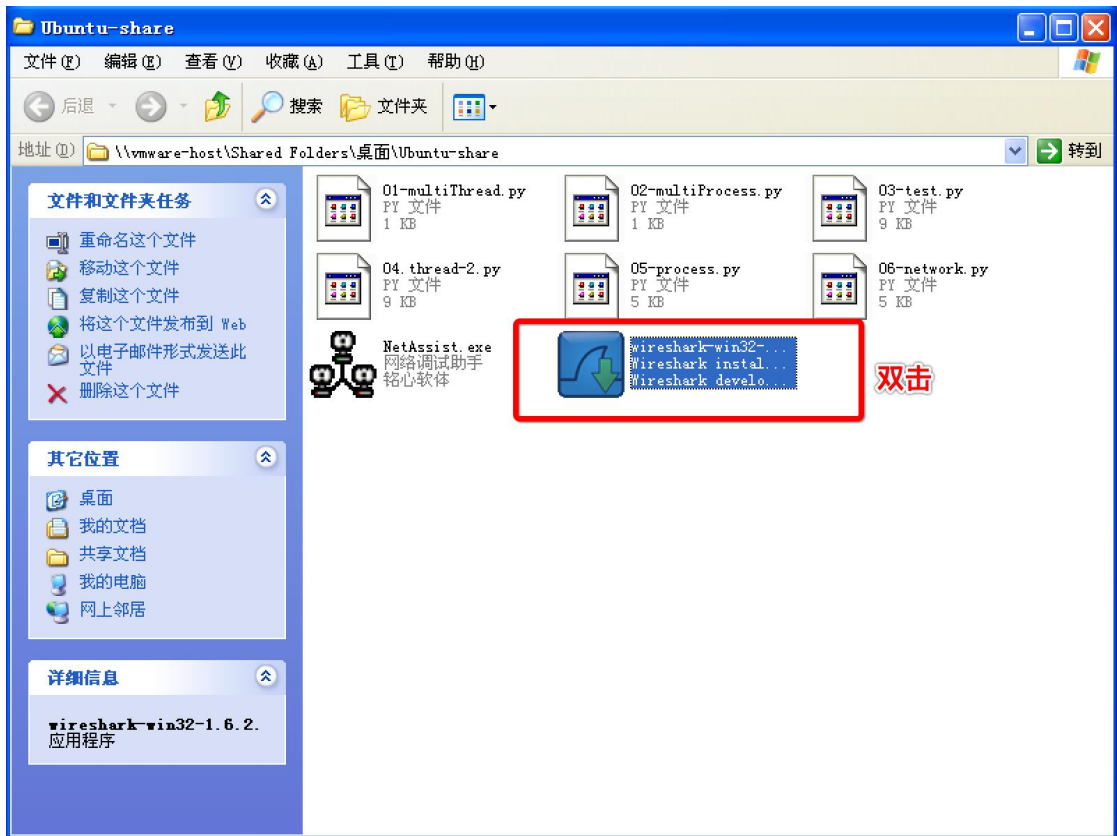
而像 **WEB** 网站这么频繁的成千上万甚至上亿客户端的连接用短连接会更省一些资源，

如果用长连接，而且同时有成千上万的用戶，如果每个用戶都占用一个连接的话，

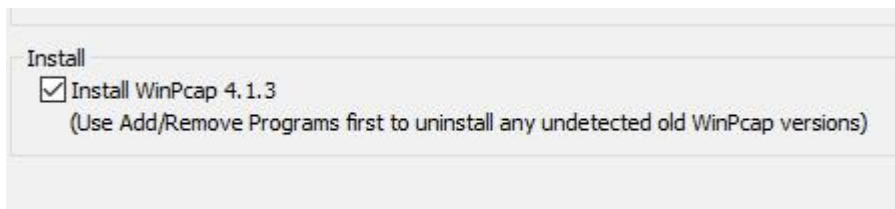
那可想而知吧。所以并发量大，但每个用戶无需频繁操作情况下需用短连好。

9 wireshark 抓包工具使用

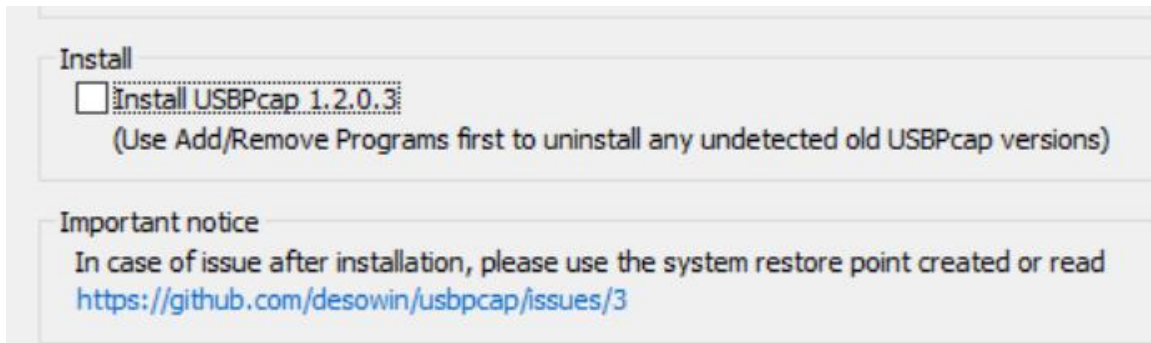
1. 安装 wireshark



安装时注意下面一步一定要打勾

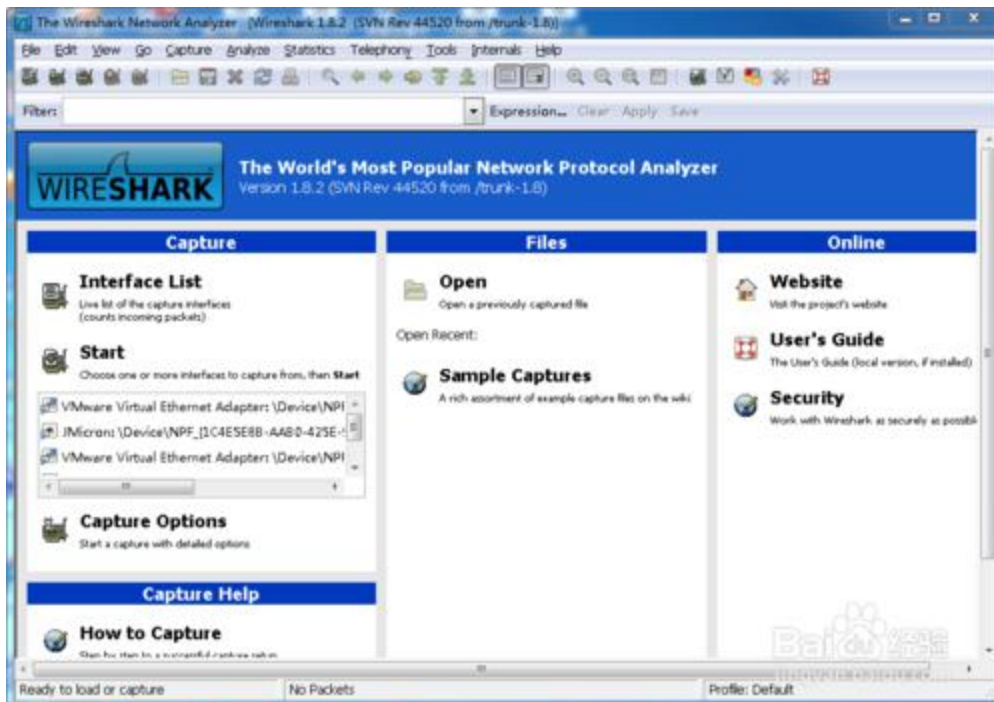


Usb 的这个不需要打勾



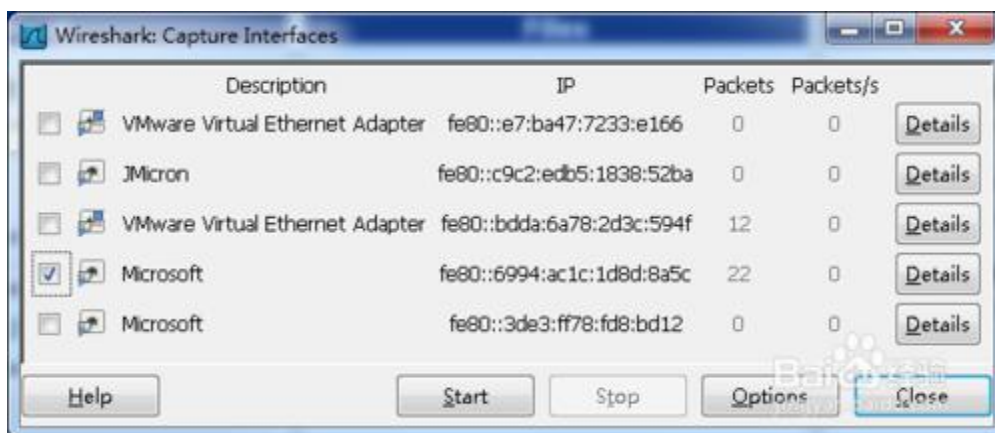
2 抓包工具 wireshark 的使用

开始界面

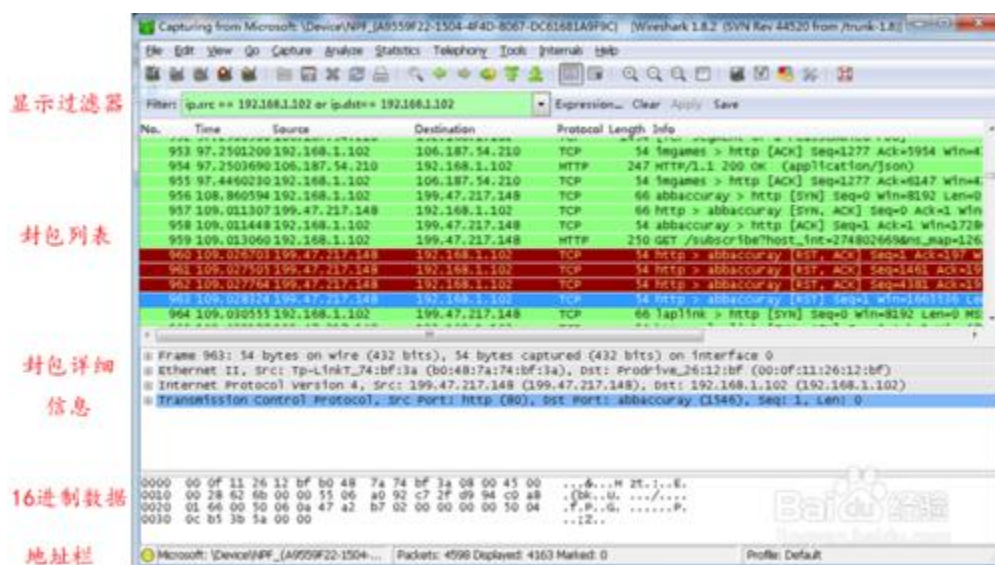


wireshark 是捕获机器上的某一块网卡的网络包，当你的机器上有多块网卡的时候，你需要选择一个网卡。

点击 Caputre->Interfaces.. 出现下面对话框，选择正确的网卡。然后点击"Start"按钮, 开始抓包

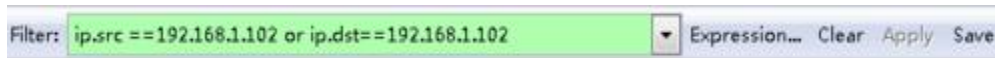


Wireshark 窗口介绍



WireShark 主要分为这几个界面

1. Display Filter(显示过滤器)， 用于过滤
2. Packet List Pane(封包列表)， 显示捕获到的封包， 有源地址和目标地址， 端口号。 颜色不同， 代表
3. Packet Details Pane(封包详细信息), 显示封包中的字段
4. Dissector Pane(16 进制数据)
5. Miscellanous(地址栏， 杂项)



使用过滤是非常重要的，初学者使用 wireshark 时，将会得到大量的冗余信息，在几千甚至几万条记录中，以至于很难找到自己需要的部分。搞得晕头转向。

过滤器会帮助我们在大量的数据中迅速找到我们需要的信息。

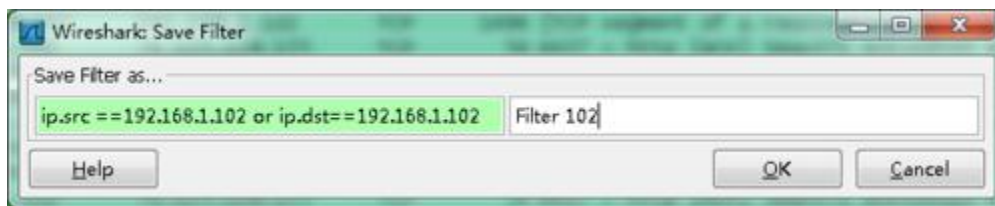
过滤器有两种，

一种是显示过滤器，就是主界面上那个，用来在捕获的记录中找到所需要的记录

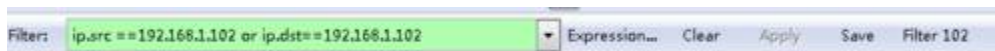
一种是捕获过滤器，用来过滤捕获的封包，以免捕获太多的记录。在 Capture -> Capture Filters 中设置

保存过滤

在 Filter 栏上，填好 Filter 的表达式后，点击 Save 按钮，取个名字。比如"Filter 102",



Filter 栏上就多了个"Filter 102" 的按钮。



过滤表达式的规则

表达式规则

1. 协议过滤

比如 TCP，只显示 TCP 协议。

2. IP 过滤

比如 ip.src == 192.168.1.102 显示源地址为 192.168.1.102，

ip.dst == 192.168.1.102, 目标地址为 192.168.1.102

3. 端口过滤

tcp.port == 80, 端口为 80 的

tcp.srcport == 80, 只显示 TCP 协议的源端口为 80 的。

4. Http 模式过滤

http.request.method=="GET", 只显示 HTTP GET 方法的。

5. 逻辑运算符为 AND/ OR

常用的过滤表达式

过滤表达式	用途
http	只查看 HTTP 协议的记录
ip.src ==192.168.1.102 or ip.dst==192.168.1.102	源地址或者目标地址是 192.168.1.102

封包列表(Packet List Pane)

封包列表的面板中显示, 编号, 时间戳, 源地址, 目标地址, 协议, 长度, 以及封包信息。你可以看到不同的协议用了不同的颜色显示。

你也可以修改这些显示颜色的规则, View -> Coloring Rules.

No.	Time	Source	Destination	Protocol	Length	Info
265	15.8906110	192.168.1.102	74.125.128.156	TCP	66	[TCP] Seq=3772
266	15.8921780	74.125.128.156	192.168.1.102	TCP	1484	[TCP] Seq=3772
267	15.8921780	192.168.1.102	74.125.128.156	TCP	66	[TCP] Seq=3772
268	15.8926100	74.125.128.156	192.168.1.102	TCP	610	[TCP] Seq=3772
269	15.8926140	192.168.1.102	74.125.128.156	TCP	66	[TCP] Seq=3772
270	16.5576320	114.80.142.90	192.168.1.102	HTTP	264	HTTP/1.0 304 Not Modified
271	16.5680360	192.168.1.102	180.168.255.118	DNS	76	Standard query 0x30ee A www.blogjava.net
272	16.5685810	192.168.1.102	180.168.255.118	DNS	75	Standard query 0xd4be A www.cppblog.com
273	16.5695380	192.168.1.102	180.168.255.118	DNS	75	Standard query 0xba0a A www.hujiang.com
274	16.7500800	192.168.1.102	114.80.142.90	TCP	34	8561 > http [ACK] Seq=2094 Ack=421 win=16860
275	16.8942490	114.80.142.90	192.168.1.102	HTTP	264	[TCP] Seq=2094
276	16.8943460	192.168.1.102	114.80.142.90	TCP	66	[TCP] Seq=2094
277	17.0615280	180.168.255.118	192.168.1.102	DNS	91	Standard query response 0xd4be A 61.155.169.1
278	17.0637590	192.168.1.102	180.168.255.118	DNS	77	Standard query 0x7272 A www.hujiangfish.com
279	17.0661740	180.168.255.118	192.168.1.102	DNS	169	Standard query response 0xb4c1 CNAME www.hujiang.com
280	17.0683610	192.168.1.102	180.168.255.118	DNS	74	Standard query 0xa990 A www.hj13.com
281	17.0690520	180.168.255.118	192.168.1.102	DNS	92	Standard query response 0x0a16 A 114.80.142.90
282	17.0733540	192.168.1.102	180.168.255.118	DNS	71	Standard query 0x0a16 A h1ng.39.net
283	17.1430580	180.168.255.118	192.168.1.102	DNS	175	Standard query response 0x7272 CNAME www.hujiang.com
284	17.1455140	192.168.1.102	180.168.255.118	DNS	75	Standard query 0x5493 A down.admin5.com

封包详细信息 (Packet Details Pane)

这个面板是我们最重要的, 用来查看协议中的每一个字段。

各行信息分别为

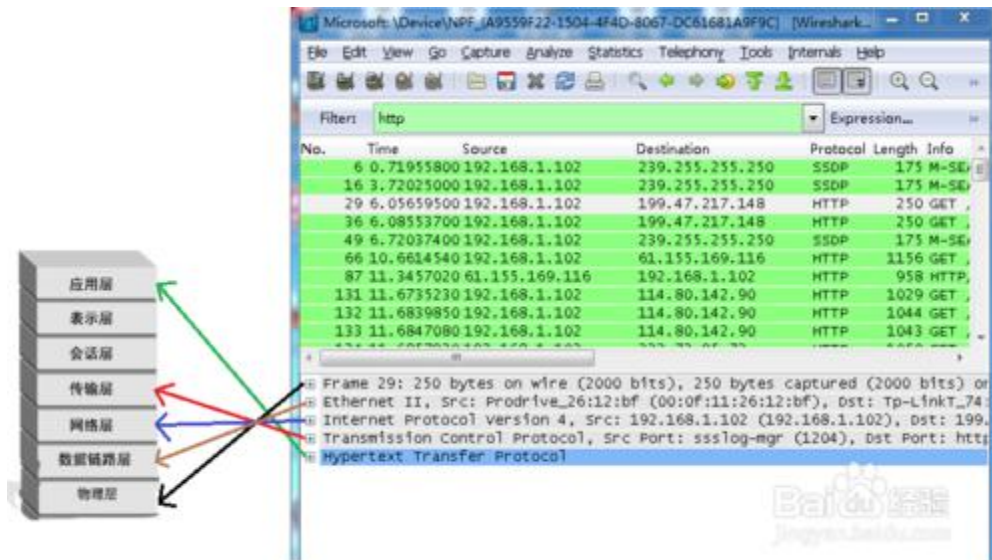
Frame: 物理层的数据帧概况

Ethernet II: 数据链路层以太网帧头部信息

Internet Protocol Version 4: 互联网层 IP 包头部信息

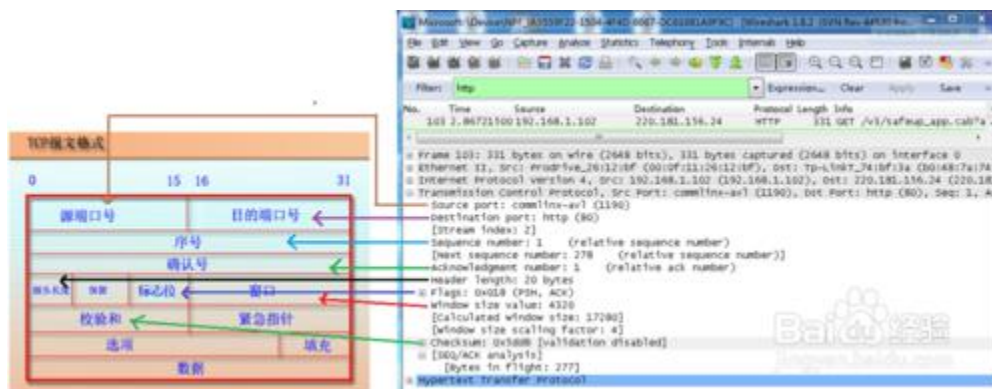
Transmission Control Protocol: 传输层 T 的数据段头部信息，此处是 TCP

Hypertext Transfer Protocol: 应用层的信息，此处是 HTTP 协议



TCP 包的具体内容

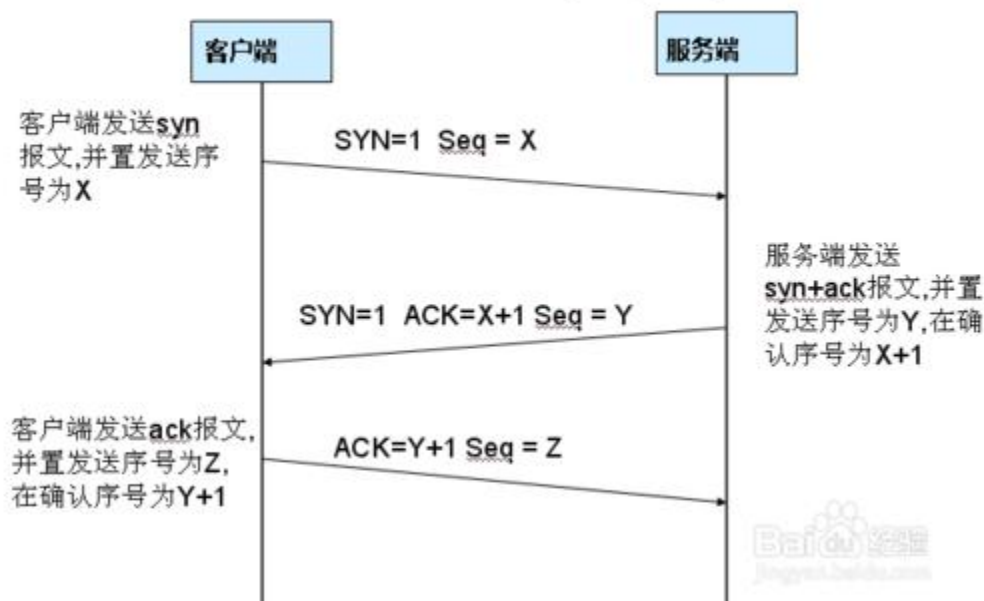
从下图可以看到 wireshark 捕获到的 TCP 包中的每个字段。



看到这，基本上对 wireshak 有了初步了解，现在我们看一个 TCP 三次握手的实例

三次握手过程为

TCP 三次握手

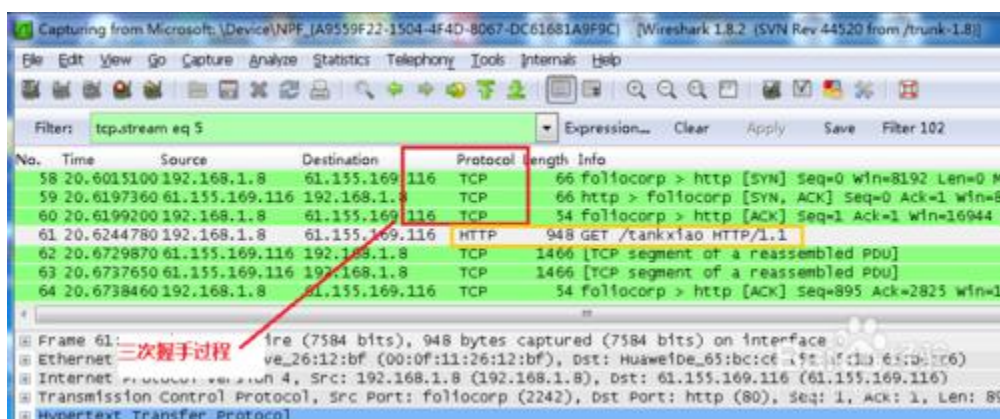


这图我都看过很多遍了，这次我们用 wireshark 实际分析下三次握手的过程。

打开 wireshark, 打开浏览器输入 `http://www.cr173.com`

在 wireshark 中输入 `http` 过滤，然后选中 `GET /tankxiao HTTP/1.1` 的那条记录，右键然后点击 "Follow TCP Stream",

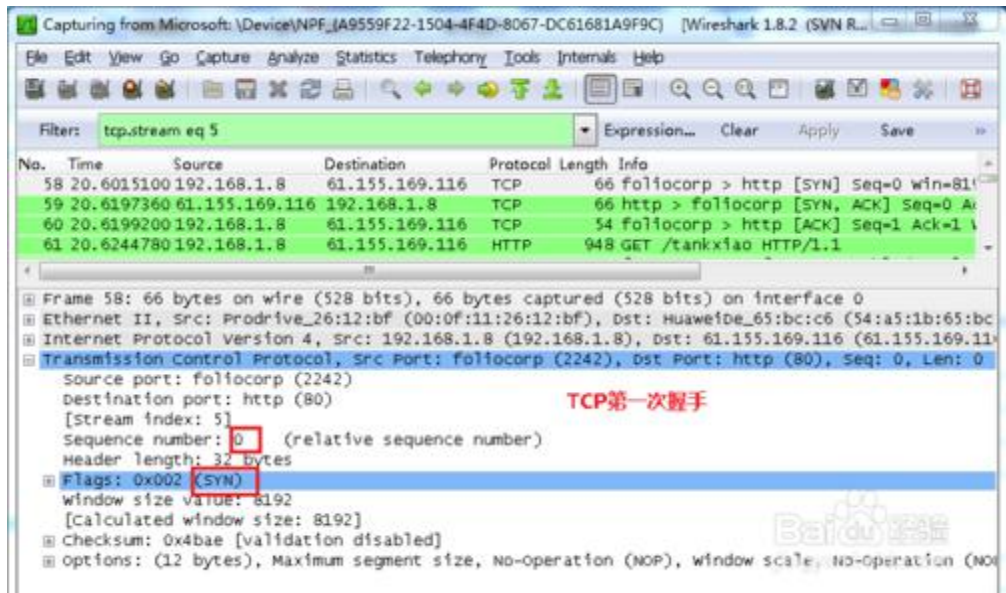
这样做的目的是为了得到与浏览器打开网站相关的数据包，将得到如下图



图中可以看到 wireshark 截获到了三次握手的三个数据包。第四个包才是 HTTP 的，这说明 HTTP 的确是使用 TCP 建立连接的。

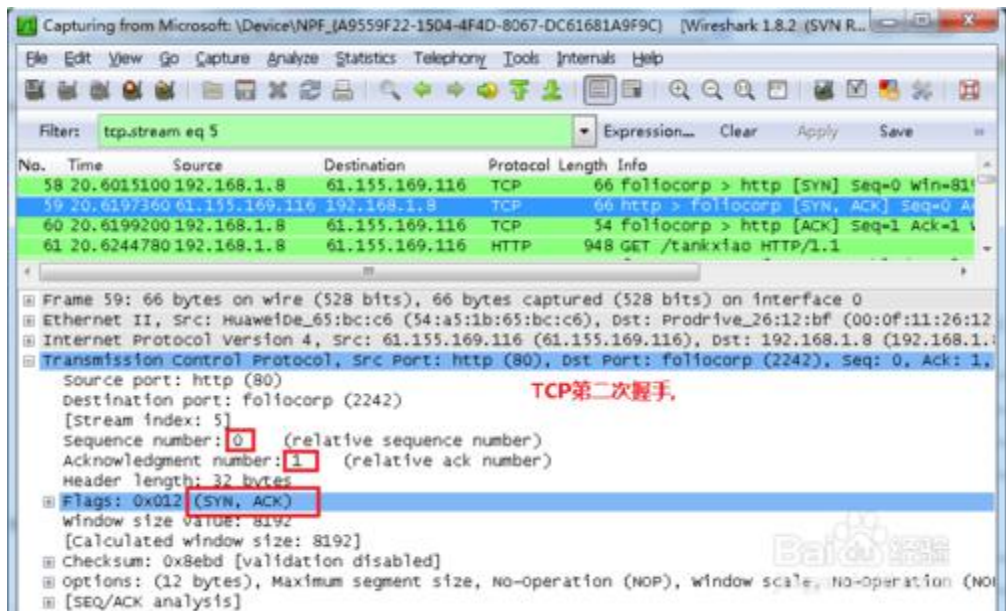
第一次握手数据包

客户端发送一个 TCP，标志位为 SYN，序列号为 0，代表客户端请求建立连接。
如下图



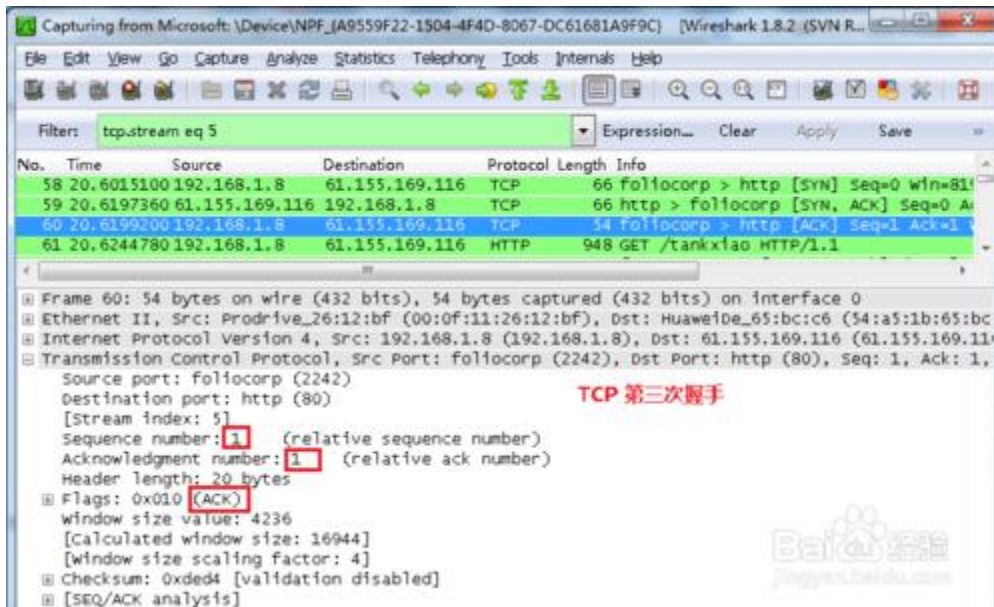
第二次握手的数据包

服务器发回确认包, 标志位为 SYN,ACK. 将确认序号(Acknowledgement Number)设置为客户的 ISN 加 1 以.即 $0+1=1$, 如下图



第三次握手的数据包

客户端再次发送确认包(ACK) SYN 标志位为 0,ACK 标志位为 1.并且把服务器发来 ACK 的序号字段+1,放在确定字段中发送给对方.并且在数据段放写 ISN 的+1, 如下图:



就这样通过了 TCP 三次握手，建立了连接

通过 TcpDump 命令也可以抓包

10 tcp-ip 简介

作为新时代标杆的我们，已经离不开手机、离不开网络，对于互联网大家可能耳熟能详，但是计算机网络的出现比互联网要早很多

1. 什么是协议



有的说英语，有的说中文，有的说德语，说同一种语言的人可以交流，不同的语言之间就不行了

为了解决不同种族人之间的语言沟通障碍，现规定国际通用语言是英语，这就是一个规定，这就是协议

2. 计算机网络沟通用什么

现在的生活中，不同的计算机只需要能够联网（有线无线都可以）那么就可以相互进行传递数据



那么不同种类之间的计算机到底是怎么进行数据传递的呢？

就像说不同语言的人沟通一样，只要有一种大家都认可都遵守的协议即可，那么这个计算机都遵守的网络通信协议叫做 TCP/IP 协议

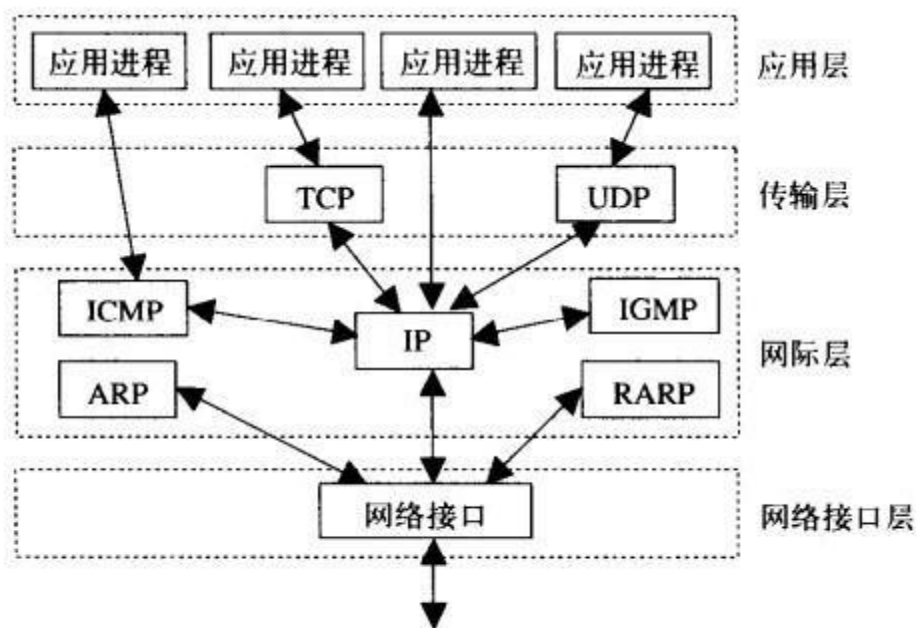
3. TCP/IP 协议(族)

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

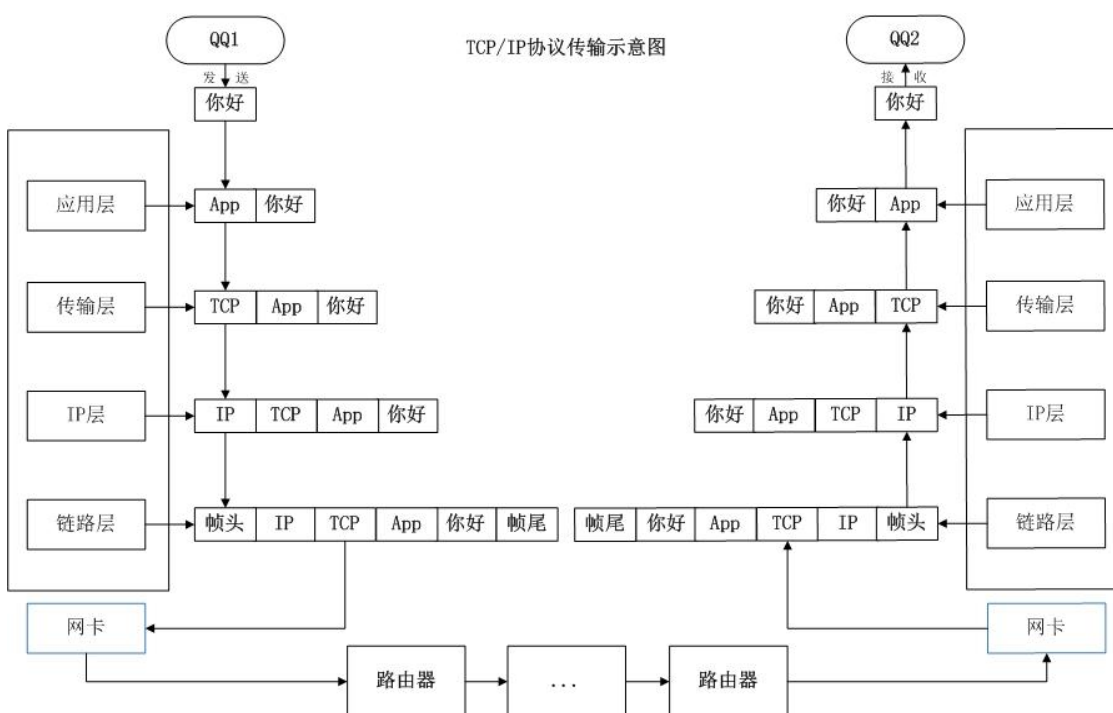
为了把全世界的所有不同类型的计算机都连接起来，就必须规定一套全球通用的协议，为了实现互联网这个目标，互联网协议族（Internet Protocol Suite）就是通用协议标准。

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简称 TCP/IP 协议(族)

常用的网络协议如下图所示：



TCP/IP协议族中各协议之间的关系

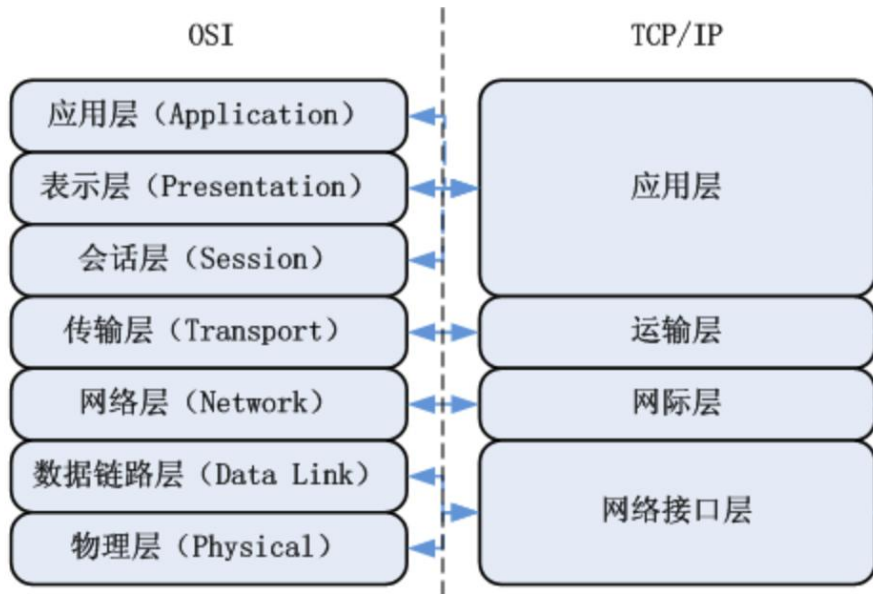


说明:

网际层也称为: 网络层

网络接口层也称为: 链路层

另外一套标准



11 TCP/IP 协议详解

1.1. TCP/IP 协议概述

协议 protocol: 通信双方必须遵循的规矩 由 iso 规定 rpc 文档

osi 参考模型: (应-表-会-传-网-数-物)

➔ 应用层 表示层 会话层 传输层 网络层 数据链路层 物理层

tcp/ip 模型 4 层:

应用层 {http 超文本传输协议 ftp 文件传输协议 telnet 远程登录 ssh 安全外壳协议 smtp 简单邮件发送 pop3 收邮件}

传输层 {tcp 传输控制协议, udp 用户数据包协议}

网络层 {ip 网际互联协议 icmp 网络控制消息协议 igmp 网络组管理协议}

网络接口层 {arp 地址转换协议, rarp 反向地址转换协议, mpls 多协议标签交换}

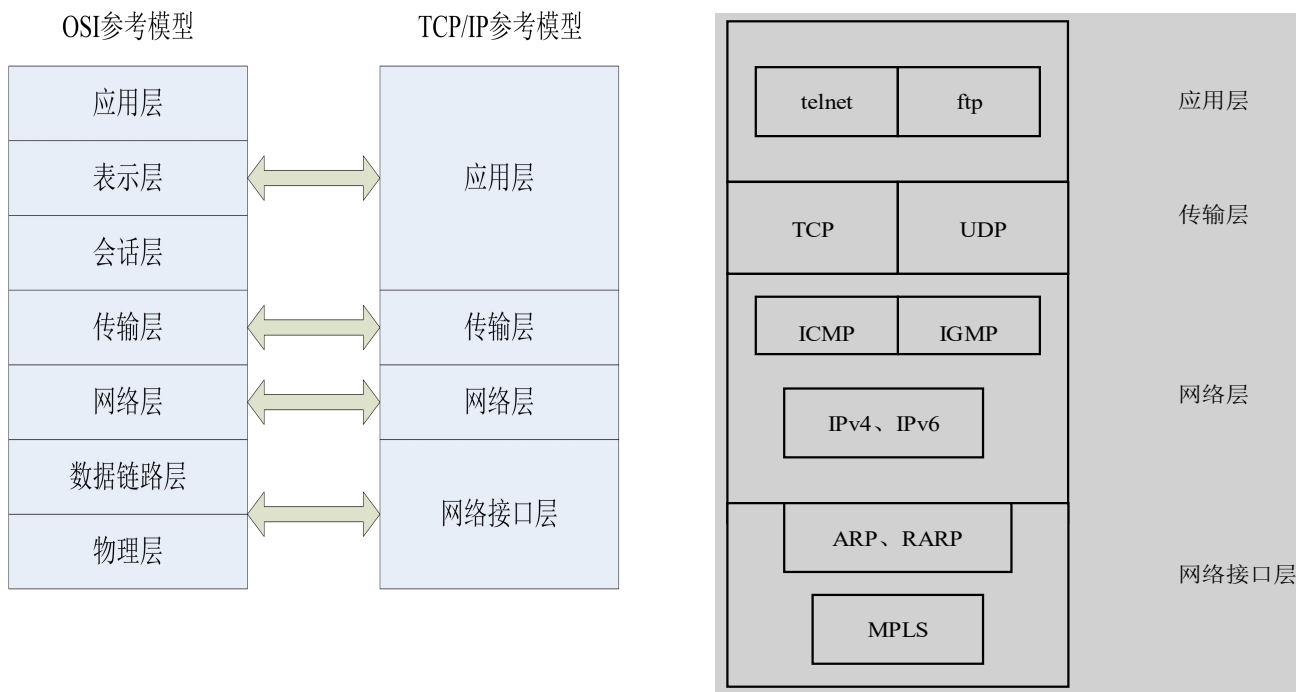
TCP 协议: 传输控制协议 面向连接的协议 能保证传输安全可靠 速度慢 (有 3 次握手)

UDP 协议: 用户数据包协议 非面向连接 速度快 不可靠

通常是 ip 地址后面跟上端口号: ip 用来定位主机 port 区别应用 (进程)

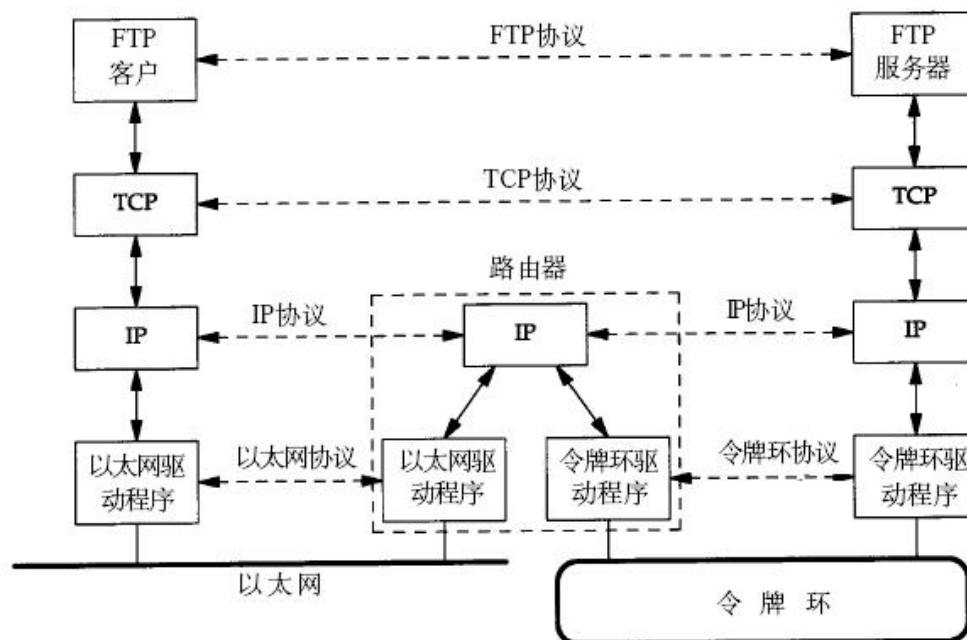
http 的端口号 80 ssh-->22 telnet-->23 ftp-->21 用户自己定义的通常要大于 1024

1.2. OSI 参考模型及 TCP/IP 参考模型



TCP/IP 协议族的每一层的作用：

- 网络接口层：负责将二进制流转换为数据帧，并进行数据帧的发送和接收。要注意的是数据帧是独立的网络信息传输单元。
- 网络层：负责将数据帧封装成 IP 数据报，并运行必要的路由算法。
- 传输层：负责端对端之间的通信会话连接和建立。传输协议的选择根据数据传输方式而定。
- 应用层：负责应用程序的网络访问，这里通过端口号来识别各个不同的进程。



跨路由通信

链路层有以太网、令牌环网等标准，链路层负责网卡设备的驱动、帧同步（即从网线上检测到什么信号算作新帧的开始）、冲突检测（如果检测到冲突就自动重发）、数据差错校验等工作。交换机是工作在链路层的网络设备，可以在不同的链路层网络之间转发数据帧（比如十兆以太网和百兆以太网之间、以太网和令牌环网之间），由于不同链路层的帧格式不同，交换机要将进来的数据包拆掉链路层首部重新封装之后再转发。

网络层的 IP 协议是构成 Internet 的基础。Internet 上的主机通过 IP 地址来标识，Internet 上有大量路由器负责根据 IP 地址选择合适的路径转发数据包，数据包从 Internet 上的源主机到目的主机往往要经过十多个路由器。路由器是工作在第三层的网络设备，同时兼有交换机的功能，可以在不同的链路层接口之间转发数据包，因此路由器需要将进来的数据包拆掉网络层和链路层两层首部并重新封装。IP 协议不保证传输的可靠性，数据包在传输过程中可能丢失，可靠性可以在上层协议或应用程序中提供支持。

网络层负责点到点（ptop, point-to-point）的传输（这里的“点”指主机或路由器），而传输层负责端到端（etoe, end-to-end）的传输（这里的“端”指源主机和目的主机）。传输层可选择 TCP 或 UDP 协议。

以太网驱动程序首先根据以太网首部中的“上层协议”字段确定该数据帧的有效载荷（payload，指除去协议首部之外实际传输的数据）是 IP、ARP 还是 RARP 协议的数据报，然后交给相应的协议处理。假如是 IP 数据报，IP 协议再根据 IP 首部中的“上层协议”字段确定该数据报的有效载荷是 TCP、UDP、ICMP 还是 IGMP，然后交给相应的协议处理。假如是 TCP 段或 UDP 段，TCP 或 UDP 协议再

根据 TCP 首部或 UDP 首部的“端口号”字段确定应该将应用层数据交给哪个用户进程。IP 地址是标识网络中不同主机的地址，而端口号就是同一台主机上标识不同进程的地址，IP 地址和端口号合起来标识网络中唯一的进程。

虽然 IP、ARP 和 RARP 数据报都需要以太网驱动程序来封装成帧，但是从功能上划分，ARP 和 RARP 属于链路层，IP 属于网络层。虽然 ICMP、IGMP、TCP、UDP 的数据都需要 IP 协议来封装成数据报，但是从功能上划分，ICMP、IGMP 与 IP 同属于网络层，TCP 和 UDP 属于传输层。

TCP/IP 协议族的每一层协议的相关注解：

- ARP：（地址转换协议）用于获得同一物理网络中的硬件主机地址。是设备通过自己知道的 IP 地址来获得自己不知道的物理地址的协议。

- RARP：反向地址转换协议（RARP：Reverse Address Resolution Protocol）反向地址转换协议（RARP）允许局域网的物理机器从网关服务器的 ARP 表或者缓存上请求其 IP 地址。网络管理员在局域网网关路由器里创建一个表以映射物理地址（MAC）和与其对应的 IP 地址。当设置一台新的机器时，其 RARP 客户机程序需要向路由器上的 RARP 服务器请求相应的 IP 地址。假设在路由表中已经设置了一个记录，RARP 服务器将会返回 IP 地址给机器，此机器就会存储起来以便日后使用。RARP 可以使用于以太网、光纤分布式数据接口及令牌环 LAN

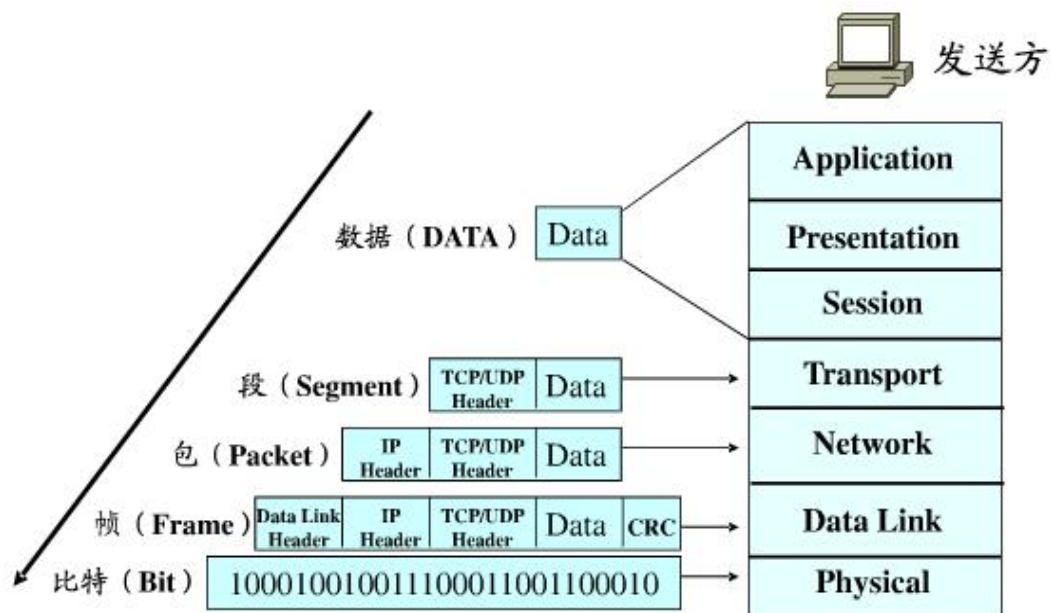
- IP：（网际互联协议）负责在主机和网络之间寻址和路由数据包。

- ICMP：（网络控制消息协议）用于发送报告有关数据包的传送错误的协议。Ping 命令所使用的协议

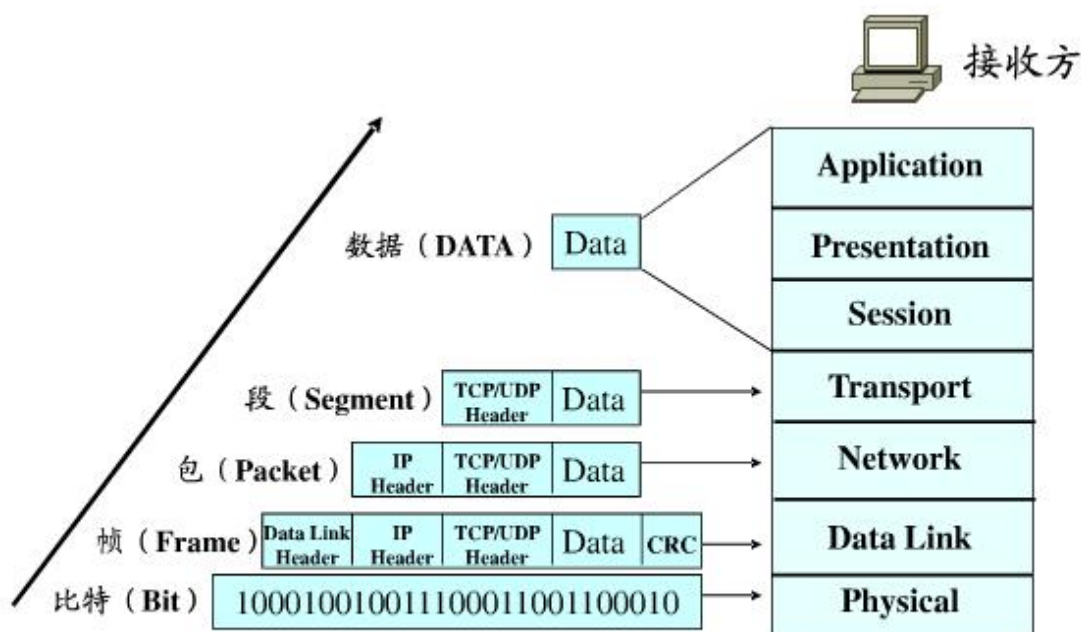
- IGMP：（网络组管理协议）被 IP 主机用来向本地多路广播路由器报告主机组成员的协议。主机与本地路由器之间使用 Internet 组管理协议（IGMP，Internet Group Management Protocol）来进行组播组成员信息的交互。

- TCP：（传输控制协议）为应用程序提供可靠的通信连接。适合于一次传输大批数据的情况。并适用于要求得到响应的应用程序。

- UDP：（用户数据包协议）提供了无连接通信，且不对传送包进行可靠的保证。适合于一次传输少量数据。



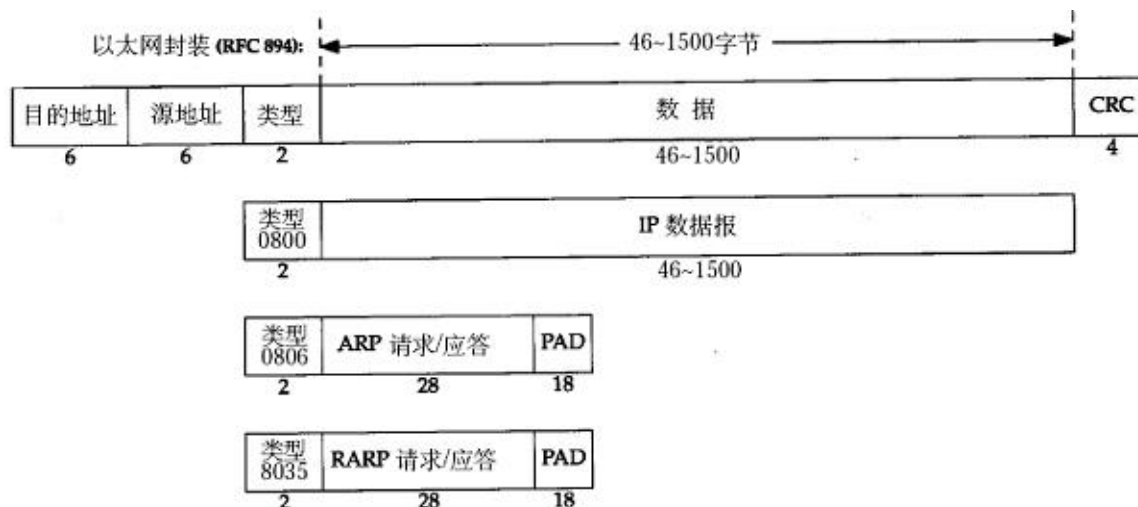
□ 数据封装过程是在不同的层次对数据打上相应的标识



□ 数据解封装过程是在不同的层次对数据去掉相应的标识

1.3. 以太网帧格式

以太网的帧格式如下所示：



以太网帧格式

其中的源地址和目的地址是指网卡的硬件地址（也叫 MAC 地址），长度是 48 位，是在网卡出厂时固化的。可在 shell 中使用 `ifconfig` 命令查看，“HWaddr 00:15:F2:14:9E:3F”部分就是硬件地址。协议字段有三种值，分别对应 IP、ARP、RARP。帧尾是 CRC 校验码。

以太网帧中的数据长度规定最小 46 字节，最大 1500 字节，ARP 和 RARP 数据包的长度不够 46 字节，要在后面补填充位。最大值 1500 称为以太网的最大传输单元（MTU），不同的网络类型有不同的 MTU，如果一个数据包从以太网路由到拨号链路上，数据包长度大于拨号链路的 MTU，则需要对数据包进行分片

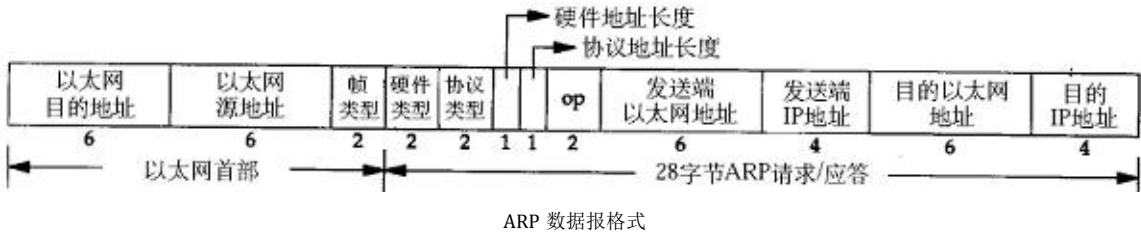
（fragmentation）。`ifconfig` 命令输出中也有“MTU:1500”。注意，MTU 这个概念指数据帧中有效载荷的最大长度，不包括帧头长度。

1.4. ARP 数据报格式

在网络通讯时，源主机的应用程序知道目的主机的 IP 地址和端口号，却不知道目的主机的硬件地址，而数据包首先是被网卡接收到再去处理上层协议的，如果接收到的数据包的硬件地址与本机不符，则直接丢弃。因此在通讯前必须获得目的主机的硬件地址。ARP 协议就起到这个作用。源主机发出 ARP 请求，询问“IP 地址是 192.168.0.1 的主机的硬件地址是多少”，并将这个请求广播到本地网段（以太网帧首部的硬件地址填 FF:FF:FF:FF:FF:FF 表示广播），目的主机接收到广播的 ARP 请求，发现其中的 IP 地址与本机相符，则发送一个 ARP 应答数据包给源主机，将自己的硬件地址填写在应答包中。

每台主机都维护一个 ARP 缓存表，可以用 `arp -a` 命令查看。缓存表中的表项有过期时间（一般为 20 分钟），如果 20 分钟内没有再次使用某个表项，则该表项失效，下次还要发 ARP 请求来获得目的主机的硬件地址。想一想，为什么表项要有过期时间而不是一直有效？

ARP 数据报的格式如下所示：



源 MAC 地址、目的 MAC 地址在以太网首部和 ARP 请求中各出现一次，对于链路层为以太网的情况是多余的，但如果链路层是其它类型的网络则有可能是必要的。硬件类型指链路层网络类型，1 为以太网，协议类型指要转换的地址类型，0x0800 为 IP 地址，后面两个地址长度对于以太网地址和 IP 地址分别为 6 和 4（字节），op 字段为 1 表示 ARP 请求，op 字段为 2 表示 ARP 应答。

看一个具体的例子。

请求帧如下（为了清晰在每行的前面加了字节计数，每行 16 个字节）：

以太网首部（14 字节）

0000: ff ff ff ff ff 00 05 5d 61 58 a8 08 06

ARP 帧（28 字节）

0000: 00 01

0010: 08 00 06 04 00 01 00 05 5d 61 58 a8 c0 a8 00 37

0020: 00 00 00 00 00 00 c0 a8 00 02

填充位（18 字节）

0020: 00 77 31 d2 50 10

0030: fd 78 41 d3 00 00 00 00 00 00 00 00

以太网首部：目的主机采用广播地址，源主机的 MAC 地址是 00:05:5d:61:58:a8，上层协议类型 0x0806 表示 ARP。

ARP 帧：硬件类型 0x0001 表示以太网，协议类型 0x0800 表示 IP 协议，硬件地址（MAC 地址）长度为 6，协议地址（IP 地址）长度为 4，op 为 0x0001 表示请求目的主机的 MAC 地址，源主机 MAC 地址为 00:05:5d:61:58:a8，源主机 IP 地址为 c0 a8 00 37（192.168.0.55），目的主机 MAC 地址全 0 待填写，目的主机 IP 地址为 c0 a8 00 02（192.168.0.2）。

由于以太网规定最小数据长度为 46 字节，ARP 帧长度只有 28 字节，因此有 18 字节填充位，填充位的内容没有定义，与具体实现相关。

应答帧如下：

以太网首部

0000: 00 05 5d 61 58 a8 00 05 5d a1 b8 40 08 06

ARP 帧

0000: 00 01

0010: 08 00 06 04 00 02 00 05 5d a1 b8 40 c0 a8 00 02

0020: 00 05 5d 61 58 a8 c0 a8 00 37

填充位

0020: 00 77 31 d2 50 10

0030: fd 78 41 d3 00 00 00 00 00 00 00 00 00

以太网首部：目的主机的 MAC 地址是 00:05:5d:61:58:a8，源主机的 MAC 地址是 00:05:5d:a1:b8:40，上层协议类型 0x0806 表示 ARP。

ARP 帧：硬件类型 0x0001 表示以太网，协议类型 0x0800 表示 IP 协议，硬件地址（MAC 地址）长度为 6，协议地址（IP 地址）长度为 4，op 为 0x0002 表示应答，源主机 MAC 地址为 00:05:5d:a1:b8:40，源主机 IP 地址为 c0 a8 00 02（192.168.0.2），目的主机 MAC 地址为 00:05:5d:61:58:a8，目的主机 IP 地址为 c0 a8 00 37（192.168.0.55）。

拓展：代理 ARP，免费 ARP

思考题：如果源主机和目的主机不在同一网段，ARP 请求的广播帧无法穿过路由器，源主机如何与目的主机通信？

1.5. IP 段格式



IP 数据报格式

版本号 (Version)：长度 4 比特。标识目前采用的 IP 协议的版本号。一般的值为 0100 (IPv4)，0110 (IPv6)

IP 包头长度 (Header Length)：长度 4 比特。这个字段的作用是为了描述 IP 包头的长度，因为在 IP 包头中有变长的可选部分。该部分占 4 个 bit 位，单位为 32bit (4 个字节)，即本区域值=IP 头部长度 (单位为 bit) / (8*4)，因此，一个 IP 包头的长度最长为“1111”，即 15*4=60 个字节。IP 包头最小长度为 20 字节。

服务类型 (Type of Service)：长度 8 比特。8 位按位被如下定义 PPP DTRCO

PPP：定义包的优先级，取值越大数据越重要

- 000 普通 (Routine)
- 001 优先的 (Priority)
- 010 立即的发送 (Immediate)
- 011 闪电式的 (Flash)
- 100 比闪电还闪电式的 (Flash Override)
- 101 CRI/TIC/ECP(找不到这个词的翻译)
- 110 网间控制 (Internetwork Control)
- 111 网络控制 (Network Control)

D 时延: 0:普通 1:延迟尽量小
T 吞吐量: 0:普通 1:流量尽量大
R 可靠性: 0:普通 1:可靠性尽量大
M 传输成本: 0:普通 1:成本尽量小
O 最后一位被保留, 恒定为 0

IP 包总长 (Total Length) : 长度 16 比特。以字节为单位计算的 IP 包的长度 (包括头部和数据), 所以 IP 包最大长度 65535 字节。

标识符 (Identifier) : 长度 16 比特。该字段和 Flags 和 Fragment Offset 字段联合使用, 对较大的上层数据包进行分段 (fragment) 操作。路由器将一个包拆分后, 所有拆分开的小包被标记相同的值, 以便目的端设备能够区分哪个包属于被拆分开的一个包的一部分。

标记 (Flags) : 长度 3 比特。该字段第一位不使用。第二位是 DF (Don't Fragment) 位, DF 位设为 1 时表明路由器不能对该上层数据包分段。如果一个上层数据包无法在不分段的情况下进行转发, 则路由器会丢弃该上层数据包并返回一个错误信息。第三位是 MF (More Fragments) 位, 当路由器对一个上层数据包分段, 则路由器会在除了最后一个分段的 IP 包的包头中将 MF 位设为 1。

片偏移 (Fragment Offset) : 长度 13 比特。表示该 IP 包在该组分片包中位置, 接收端靠此来组装还原 IP 包。片偏移量, 13 位, 指出较长的分组在分片后, 某段在原分组的相对位置。也就是说相对原分组数据段的起点, 该片从何处开始。段偏移以 8 字节为偏移单位。这就是, 每个分片的长度一定是 8 字节 (64 位) 的整数倍

生存时间 (TTL) : 长度 8 比特。当 IP 包进行传送时, 先会对该字段赋予某个特定的值。当 IP 包经过每一个沿途的路由器的时候, 每个沿途的路由器会将 IP 包的 TTL 值减少 1。如果 TTL 减少为 0, 则该 IP 包会被丢弃。这个字段可以防止由于路由环路而导致 IP 包在网络中不停被转发。

- 1, TTL 的作用是限制 IP 数据包在计算机网络中的存在的时间。TTL 的最大值是 255, TTL 的一个推荐值是 64。
- 2, 虽然 TTL 从字面上翻译, 是可以存活的时间, 但实际上 TTL 是 IP 数据包在计算机网络中可以转发的最大跳数。
- 3, TTL 字段由 IP 数据包的发送者设置, 在 IP 数据包从源到目的整个转发路径上, 每经过一个路由器, 路由器都会修改这个 TTL 字段值, 具体的做法是把该 TTL 的值减 1, 然后再将 IP 包转发出去。

4, 如果在 IP 包到达目的 IP 之前, TTL 减少为 0, 路由器将会丢弃收到的 TTL =0 的 IP 包并向 IP 包的发送者发送 ICMP time exceeded 消息。

TTL 的主要作用是避免 IP 包在网络中的无限循环和收发, 节省了网络资源, 并能使 IP 包的发送者能收到告警消息。,

5, TTL 是由发送主机设置的, 以防止数据包不断在 IP 互联网络上永不终止地循环。转发 IP 数据包时, 要求路由器至少将 TTL 减小 1。

协议 (Protocol) : 长度 8 比特。标识了上层所使用的协议。

以下是比较常用的协议号:

- 1 ICMP
- 2 IGMP
- 6 TCP
- 17 UDP
- 88 IGRP
- 89 OSPF

头部校验 (Header Checksum) : 长度 16 位。用来做 IP 头部的正确性检测, 但不包含数据部分。因为每个路由器要改变 TTL 的值, 所以路由器会为每个通过的数据包重新计算这个值。

起源和目标地址 (Source and Destination Addresses) : 这两个地址都是 32 比特。标识了这个 IP 包的起源和目标地址。要注意除非使用 NAT, 否则整个传输的过程中, 这两个地址不会改变。

至此, IP 包头基本的 20 字节已介绍完毕, 此后部分属于可选项, 不是必须的部分。

可选项 (Options) : 这是一个可变长的字段。该字段属于可选项, 主要用于测试, 由起源设备根据需要改写。可选项包含以下内容:

松散源路由 (Loose source routing) : 给出一连串路由器接口的 IP 地址。IP 包必须沿着这些 IP 地址传送, 但是允许在相继的两个 IP 地址之间跳过多个路由器。

严格源路由 (Strict source routing) : 给出一连串路由器接口的 IP 地址。IP 包必须沿着这些 IP 地址传送, 如果下一跳不在 IP 地址表中则表示发生错误。

路由记录 (Record route) : 当 IP 包离开每个路由器的时候记录路由器的出站接口的 IP 地址。

时间戳 (Timestamps) : 当 IP 包离开每个路由器的时候记录时间。

填充（Padding）：因为 IP 包头长度（Header Length）部分的单位为 32bit，所以 IP 包头的长度必须为 32bit 的整数倍。因此，在可选项后面，IP 协议会填充若干个 0，以达到 32bit 的整数倍

想一想，前面讲了以太网帧中的最小数据长度为 46 字节，不足 46 字节的要用填充字节补上，那么如何界定这 46 字节里前多少个字节是 IP、ARP 或 RARP 数据报而后面是填充字节？

1.6. 路由(route)

路由（名词）

数据包从源地址到目的地址所经过的路径，由一系列路由节点组成。

路由（动词）

某个路由节点为数据包选择投递方向的选路过程。

1.6.1. 路由器工作原理

路由器（Router）是连接因特网中各局域网、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。

传统地，路由器工作于 OSI 七层协议中的第三层，其主要任务是接收来自一个网络接口的数据包，根据其中所含的目的地址，决定转发到下一个目的地址。因此，路由器首先得在转发路由表中查找它的目的地址，若找到了目的地址，就在数据包的帧格前添加下一个 MAC 地址，同时 IP 数据包头的 TTL（Time To Live）域也开始减数，并重新计算校验和。当数据包被送到输出端口时，它需要按顺序等待，以便被传送到输出链路上。

路由器在工作时能够按照某种路由通信协议查找设备中的路由表。如果到某一特定节点有一条以上的路径，则基本预先确定的路由准则是选择最优（或最经济）的传输路径。由于各种网络段和其相互连接情况可能会因环境变化而变化，因此路由情况的信息一般也按所使用的路由信息协议的规定而定时更新。

网络中，每个路由器的基本功能都是按照一定的规则来动态地更新它所保持的路由表，以便保持路由信息的有效性。为了便于在网络间传送报文，路由器总是先按照预定的规则把较大的数据分解成适当大小的数据包，再将这些数据包分别通过相同或不同路径发送出去。当这些数据包按先后秩序到达目的地后，再把分解的数据包按照一定顺序包装成原有的报文形式。路由器的分层寻址功能是路由器的重要功能之一，该功能可以帮助具有很多节点站的网络来存储寻址信息，同时还能在网络间截获发送到远地网段的报文，起转发作用；选择最合理的路由，引导通信也是路由器基本功能；多协议路由器还可以连接使用不同通信协议的网络段，成为不同通信协议网络段之间的通信平台。

路由和交换之间的主要区别就是交换发生在 OSI 参考模型第二层（数据链路层），而路由发生在第三层，即网络层。这一区别决定了路由和交换在移动信息的过程中需使用不同的控制信息，所以两者实现各自功能的方式是不同的。

1.6.2. 路由表(Routing Table)

在计算机网络中，路由表或称路由择域信息库（RIB）是一个存储在路由器或者联网计算机中的电子表格（文件）或类数据库。路由表存储着指向特定网络地址的路径。

路由条目

路由表中的一行，每个条目主要由目的网络地址、子网掩码、下一跳地址、发送接口四部分组成，如果要发送的数据包的目的网络地址匹配路由表中的某一行，就按规定的接口发送到下一跳地址。

缺省路由条目

路由表中的最后一行，主要由下一跳地址和发送接口两部分组成，当目的地址与路由表中其它行都不匹配时，就按缺省路由条目规定的接口发送到下一跳地址。

路由节点

一个具有路由能力的主机或路由器，它维护一张路由表，通过查询路由表来决定向哪个接口发送数据包。

1.6.3. 以太网交换机工作原理

以太网交换机是基于以太网传输数据的交换机，以太网采用共享总线型传输媒体方式的局域网。以太网交换机的结构是每个端口都直接与主机相连，并且一般都工作在全双工方式。交换机能同时连通许多对端口，使每一对相互通信的主机都能像独占通信媒体那样，进行无冲突地传输数据。

以太网交换机工作于 OSI 网络参考模型的第二层（即数据链路层），是一种基于 MAC（Media Access Control，介质访问控制）地址识别、完成以太网数据帧转发的网络设备。

1.6.4. hub 工作原理

集线器实际上就是中继器的一种，其区别仅在于集线器能够提供更多的端口服务，所以集线器又叫多口中继器。

集线器功能是随机选出某一端口的设备，并让它独占全部带宽，与集线器的上联设备（交换机、路由器或服务器等）进行通信。从 Hub 的工作方式可以看出，它在网络中只起到信号放大和重发作用，其目的是扩大网络的传输范围，而不具备

信号的定向传送能力，是一个标准的共享式设备。其次是 Hub 只与它的上联设备(如上层 Hub、交换机或服务器)进行通信，同层的各端口之间不会直接进行通信，而是通过上联设备再将信息广播到所有端口上。由此可见，即使是在同一 Hub 的不同两个端口之间进行通信，都必须经过两步操作：

第一步是将信息上传到上联设备；

第二步是上联设备再将该信息广播到所有端口上。

Hub 功能非常弱，目前已经几乎消失殆尽（都购买不着），现在都是 4 口交换机，8 口交换机，或者是无线路由器。

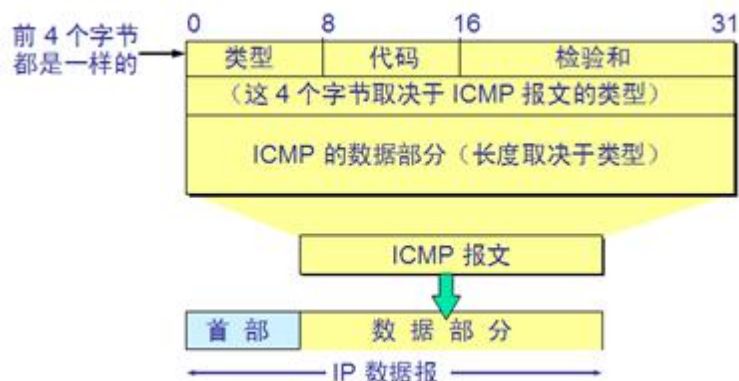
1.7. ICMP 协议

ICMP 是（Internet Control Message Protocol）Internet 控制报文协议。它是 TCP/IP 协议族的一个子协议，用于在 IP 主机、路由器之间传递控制消息。**控制消息是指网络通不通、主机是否可达、路由是否可用等网络本身的消息。**这些控制消息虽然并不传输用户数据，但是对于用户数据的传递起着重要的作用

TICMP 协议是一种面向无连接的协议，用于传输出错报告控制信息。它是一个非常重要的协议，它对于网络安全具有极其重要的意义。

它是 TCP/IP 协议族的一个子协议，属于网络层协议，主要用于在主机与路由器之间传递控制信息，包括报告错误、交换受限控制和状态信息等。当遇到 IP 数据无法访问目标、IP 路由器无法按当前的传输速率转发数据包等情况时，会自动发送 ICMP 消息。ICMP 报文在 IP 帧结构的首部协议类型字段（Protocol 8bit)的值=1.

如下图所示，ICMP 包有一个 8 字节长的包头，其中前 4 个字节是固定的格式，包含 8 位类型字段，8 位代码字段和 16 位的校验和；后 4 个字节根据 ICMP 包的类型而取不同的值。



ICMP 提供一致易懂的出错报告信息。发送的出错报文返回到发送原数据的设备，因为只有发送设备才是出错报文的逻辑接受者。发送设备随后可根据 ICMP 报

文确定发生错误的类型，并确定如何才能更好地重发失败的数据包。但是 ICMP 唯一的功能是报告问题而不是纠正错误，纠正错误的任务由发送方完成。

我们在网络中经常会使用到 ICMP 协议，比如我们经常使用的用于检查网络不通的 Ping 命令（Linux 和 Windows 中均有），这个“Ping”的过程实际上就是 ICMP 协议工作的过程。还有其他的网络命令如跟踪路由的 Tracert 命令也是基于 ICMP 协议的。

TYPE	CODE	Description	Query	Error
0	0	Echo Reply——回显应答（Ping 应答）	x	
3	0	Network Unreachable——网络不可达		x
3	1	Host Unreachable——主机不可达		x
3	2	Protocol Unreachable——协议不可达		x
3	3	Port Unreachable——端口不可达		x
3	4	Fragmentation needed but no frag. bit set——需要进行分片但设置不分片比特		x
3	5	Source routing failed——源站选路失败		x
3	6	Destination network unknown——目的网络未知		x
3	7	Destination host unknown——目的主机未知		x
3	8	Source host isolated (obsolete)——源主机被隔离（作废不用）		x
3	9	Destination network administratively prohibited——目的网络被强制禁止		x
3	10	Destination host administratively prohibited——目的主机被强制禁止		x
3	11	Network unreachable for TOS——由于服务类型 TOS，网络不可达		x
3	12	Host unreachable for TOS——由于服务类型 TOS，主机不可达		x
3	13	Communication administratively prohibited by filtering——由于过滤，通信被强制禁止		x
3	14	Host precedence violation——主机越权		x

3	15	Precedence cutoff in effect——优先中止生效		x
4	0	Source quench——源端被关闭（基本流控制）		
5	0	Redirect for network——对网络重定向		
5	1	Redirect for host——对主机重定向		
5	2	Redirect for TOS and network——对服务类型和网络重定向		
5	3	Redirect for TOS and host——对服务类型和主机重定向		
8	0	Echo request——回显请求（Ping 请求）	x	
9	0	Router advertisement——路由器通告		
10	0	Route solicitation——路由器请求		
11	0	TTL equals 0 during transit——传输期间生存时间为 0		x
11	1	TTL equals 0 during reassembly——在数据报组装期间生存时间为 0		x
12	0	IP header bad (catchall error)——坏的 IP 首部（包括各种差错）		x
12	1	Required options missing——缺少必需的选项		x
13	0	Timestamp request (obsolete)——时间戳请求（作废不用）	x	
14		Timestamp reply (obsolete)——时间戳应答（作废不用）	x	
15	0	Information request (obsolete)——信息请求（作废不用）	x	
16	0	Information reply (obsolete)——信息应答（作废不用）	x	
17	0	Address mask request——地址掩码请求	x	
18	0	Address mask reply——地址掩码应答		

1.8. TCP 协议

(1) 概述

TCP 是 TCP/IP 体系中面向连接的运输层协议，它提供全双工和可靠交付的服务。它采用许多机制来确保端到端结点之间的可靠数据传输，如采用序列号、确认重传、滑动窗口等。

首先，TCP 要为所发送的每一个报文段加上序列号，保证每一个报文段能被接收方接收，并只被正确的接收一次。

其次，TCP 采用具有重传功能的积极确认技术作为可靠数据流传输服务的基础。这里“确认”是指接收端在正确收到报文段之后向发送端回送一个确认

(ACK) 信息。发送方将每个已发送的报文段备份在自己的缓冲区里，而且在收到相应的确认之前是不会丢弃所保存的报文段的。“积极”是指发送方在每一个报文段发送完毕的同时启动一个定时器，加入定时器的定时期满而关于报文段的确认信息还没有达到，则发送方认为该报文段已经丢失并主动重发。为了避免由于网络延时引起迟到的确认和重复的确认，TCP 规定在确认信息中捎带一个报文段的序号，使接收方能正确的将报文段与确认联系起来。

最后，采用可变长的滑动窗口协议进行流量控制，以防止由于发送端与接收端之间的不匹配而引起的数据丢失。这里所采用的滑动窗口协议与数据链路层的滑动窗口协议在工作原理上完全相同，唯一的区别在于滑动窗口协议用于传输层是为了在端对端节点之间实现流量控制，而用于数据链路层是为了在相邻节点之间实现流量控制。TCP 采用可变长的滑动窗口，使得发送端与接收端可根据自己的 CPU 和数据缓存资源对数据发送和接收能力来进行动态调整，从而灵活性更强，也更合理。

(2) 三次握手协议

在利用 TCP 实现源主机和目的主机通信时，目的主机必须同意，否则 TCP 连接无法建立。为了确保 TCP 连接的成功建立，TCP 采用了一种称为三次握手的方式，三次握手方式使得“序号/确认号”系统能够正常工作，从而使它们的序号达成同步。如果三次握手成功，则连接建立成功，可以开始传送数据信息。

其三次握手分别为：

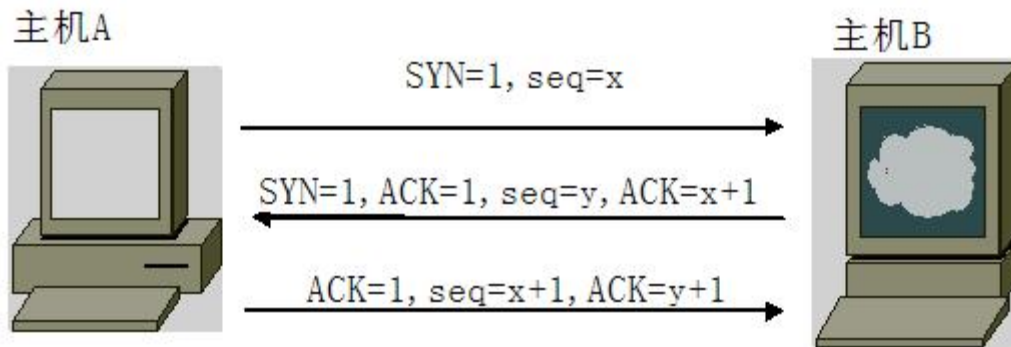
1) 源主机 A 的 TCP 向主机 B 发送连接请求报文段，其首部中的 SYN（同步）标志位应置为 1，表示想跟目标主机 B 建立连接，进行通信，并发送一个同步序号 X（例：SEQ=100）进行同步，表明在后面传送数据时的第一个数据字节的序号为 X+1（即 101）。

2) 目标主机 B 的 TCP 收到连接请求报文段后，如同意，则发回确认。再确认报中应将 ACK 位和 SYN 位置为 1。确认号为 X+1，同时也为自己选择一个序号 Y。

3) 源主机 A 的 TCP 收到目标主机 B 的确认后要想目标主机 B 给出确认。其 ACK 置为 1, 确认号为 $Y+1$, 而自己的序号为 $X+1$ 。TCP 的标准规定, SYN 置 1 的报文段要消耗掉一个序号。

运行客户进程的源主机 A 的 TCP 通知上层应用进程, 连接已经建立。当源主机 A 向目标主机 B 发送第一个数据报文段时, 其序号仍为 $X+1$, 因为前一个确认报文段并不消耗序号。

当运行服务进程的目标主机 B 的 TCP 收到源主机 A 的确认后, 也通知其上层应用进程, 连接已经建立。至此建立了一个全双工的连接。



三次握手: 为应用程序提供可靠的通信连接。适合于一次传输大批数据的情况。并适用于要求得到响应的应用程序。

(3) TCP 的四次挥手

由于 TCP 连接是全双工的, 因此每个方向都必须单独进行关闭。这原则是当一方完成它的数据发送任务后就能发送一个 FIN 来终止这个方向的连接。收到一个 FIN 只意味着这一方向上没有数据流动, 一个 TCP 连接在收到一个 FIN 后仍能发送数据。首先进行关闭的一方将执行主动关闭, 而另一方执行被动关闭。

- (1) TCP 客户端发送一个 FIN, 用来关闭客户到服务器的数据传送。
- (2) 服务器收到这个 FIN, 它发回一个 ACK, 确认序号为收到的序号加 1。和 SYN 一样, 一个 FIN 将占用一个序号。
- (3) 服务器关闭客户端的连接, 发送一个 FIN 给客户端。
- (4) 客户端发回 ACK 报文确认, 并将确认序号设置为收到序号加 1。

CLOSED:

这个没什么好说的了, 表示初始状态。

LISTEN:

这个也是非常容易理解的一个状态，表示服务器端的某个 **SOCKET** 处于监听状态，可以接受连接了。

SYN_RCVD:

这个状态表示接受到了 **SYN** 报文，在正常情况下，这个状态是服务器端的 **SOCKET** 在建立 **TCP** 连接时的三次握手会话过程中的一个中间状态，很短暂，基本上用 **netstat** 你是很难看到这种状态的，除非你特意写了一个客户端测试程序，故意将三次 **TCP** 握手过程中最后一个 **ACK** 报文不予发送。因此这种状态时，当收到客户端的 **ACK** 报文后，它会进入到 **ESTABLISHED** 状态。

SYN_SENT:

这个状态与 **SYN_RCVD** 遥想呼应，当客户端 **SOCKET** 执行 **CONNECT** 连接时，它首先发送 **SYN** 报文，因此也随即它会进入到了 **SYN_SENT** 状态，并等待服务端的发送三次握手中的第 2 个报文。**SYN_SENT** 状态表示客户端已发送 **SYN** 报文。

ESTABLISHED:

这个容易理解了，表示连接已经建立了。

FIN_WAIT_1:

这个状态要好好解释一下，其实 **FIN_WAIT_1** 和 **FIN_WAIT_2** 状态的真正含义都是表示等待对方的 **FIN** 报文。而这两种状态的区别是：**FIN_WAIT_1** 状态实际上是当 **SOCKET** 在 **ESTABLISHED** 状态时，它想主动关闭连接，向对方发送了 **FIN** 报文，此时该 **SOCKET** 即进入到 **FIN_WAIT_1** 状态。而当对方回应 **ACK** 报文后，则进入到 **FIN_WAIT_2** 状态，当然在实际的正常情况下，无论对方何种情况下，都应该马上回应 **ACK** 报文，所以 **FIN_WAIT_1** 状态一般是比较难见到的，而 **FIN_WAIT_2** 状态还有时常常可以用 **netstat** 看到。

FIN_WAIT_2:

上面已经详细解释了这种状态，实际上 **FIN_WAIT_2** 状态下的 **SOCKET**，表示半连接，也即有一方要求 **close** 连接，但另外还告诉对方，我暂时还有点数据需要传送给你，稍后再关闭连接。

TIME_WAIT:

表示收到了对方的 **FIN** 报文，并发送出了 **ACK** 报文，就等 **2MSL** 后即可回到 **CLOSED** 可用状态了。如果 **FIN_WAIT_1** 状态下，收到了对方同时带 **FIN** 标志和 **ACK** 标志的报文时，可以直接进入到 **TIME_WAIT** 状态，而无须经过 **FIN_WAIT_2** 状态。

CLOSING:

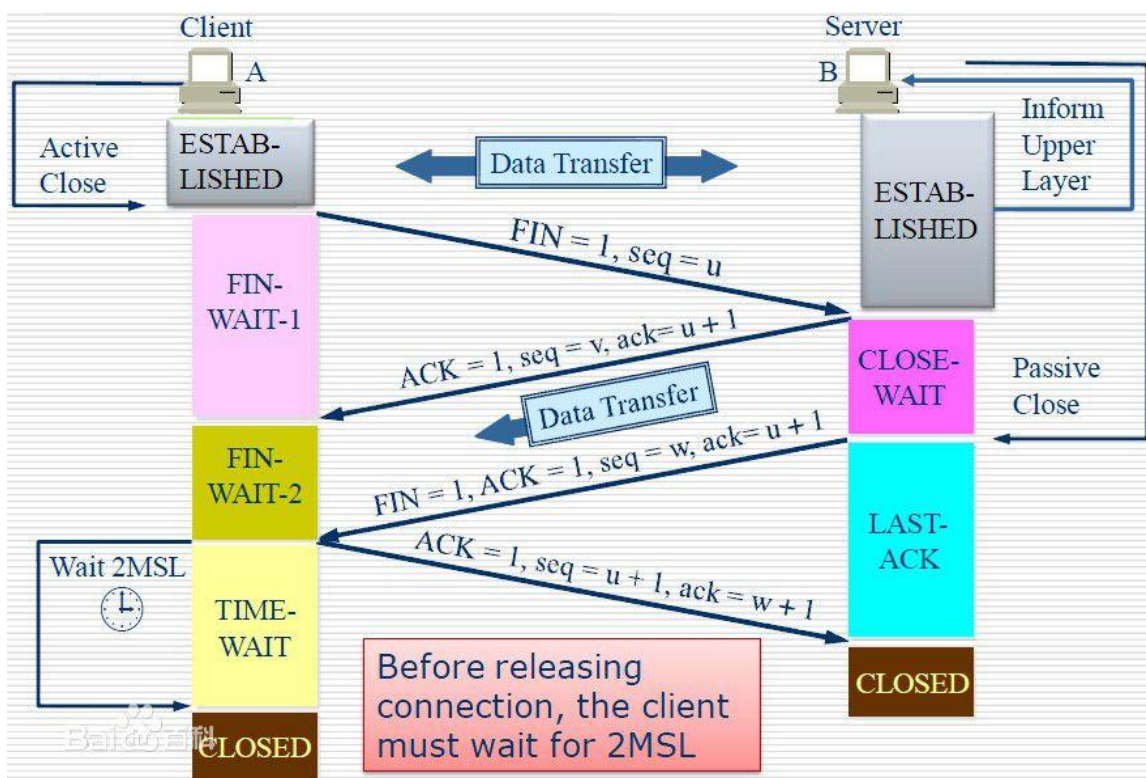
这种状态比较特殊，实际情况中应该是很少见，属于一种比较罕见的例外状态。正常情况下，当你发送 **FIN** 报文后，按理来说是应该先收到（或同时收到）对方的 **ACK** 报文，再收到对方的 **FIN** 报文。但是 **CLOSING** 状态表示你发送 **FIN** 报文后，并没有收到对方的 **ACK** 报文，反而却也收到了对方的 **FIN** 报文。什么情况下会出现此种情况呢？其实细想一下，也不难得出结论：那就是如果双方几乎在同时 **close** 一个 **SOCKET** 的话，那么就出现了双方同时发送 **FIN** 报文的情况，也就会出现 **CLOSING** 状态，表示双方都正在关闭 **SOCKET** 连接。

CLOSE_WAIT:

这种状态的含义其实是表示在等待关闭。怎么理解呢？当对方 **close** 一个 **SOCKET** 后发送 **FIN** 报文给自己，你系统毫无疑问地会回应一个 **ACK** 报文给对方，此时则进入到 **CLOSE_WAIT** 状态。接下来呢，实际上你真正需要考虑的事情是查看你是否还有数据发送给对方，如果没有的话，那么你也就可以 **close** 这个 **SOCKET**，发送 **FIN** 报文给对方，也即关闭连接。所以你在 **CLOSE_WAIT** 状态下，需要完成的事情是等待你去关闭连接。

LAST_ACK:

这个状态还是比较好理解的，它是被动关闭一方在发送 **FIN** 报文后，最后等待对方的 **ACK** 报文。当收到 **ACK** 报文后，也即可以进入到 **CLOSED** 可用状态了。



MSL 值是 `/proc/sys/net/ipv4/tcp_fin_timeout` 默认 60 秒

(4) TCP 数据报头 TCP 头信息



- 源端口、目的端口：16 位长。标识出远端和本地的端口号。
- 序号：32 位长。标识发送的数据报的顺序。

- 确认号：32 位长。希望收到的下一个数据报的序列号。
- TCP 头长：4 位长。表明 TCP 头中包含多少个 32 位字。就是有多少个 4 个字节
- 4 位未用。

CWR: 拥塞窗口减（发送方降低它的发送速率）

ECE: ECN 回显（发送方收到了一个更早的拥塞报告）

URG: 紧急指针（urgent pointer）有效，紧急指针指出在本报文段中的紧急数据的最后一个字节的序号

- ACK: ACK 位置 1 表明确认号是合法的。如果 ACK 为 0，那么数据报不包含确认信息，确认字段被省略。
- PSH: 表示是带有 PUSH 标志的数据。接收方因此请求数据报一到便可送往应用程序而不必等到缓冲区装满时才发送。当 PSH=1 时，则报文段会被尽快地交付给目的方，不会对这样的报文段使用缓存策略
- RST: 用于复位由于主机崩溃或其他原因而出现的错误的连接。还可以用于拒绝非法的数据报或拒绝连接请求。当 RST 为 1 时，表明 TCP 连接中出现了严重的差错，必须释放连接，然后再重新建立连接。
- SYN: 用于建立连接。当 SYN=1 时，表示发起一个连接请求。
- FIN: 用于释放连接。当 FIN=1 时，表明此报文段的发送端的数据已发送完成，并要求释放连接。
- 窗口大小：16 位长。窗口大小字段表示在确认了字节之后还可以发送多少个字节。此字段用来进行流量控制。单位为字节数，这个值是本机期望一次接收的字节数
- 校验和：16 位长。是为了确保高可靠性而设置的。它校验头部、数据和伪 TCP 头部之和。
- 可选项：0 个或多个 32 位字。包括最大 TCP 载荷，窗口比例、选择重复数据报等选项。

大小：从源端口到紧急指针，总计 20 个字节，没有任何选项字段的 TCP 头部长度为 20 字节；最多可以有 60 字节的 TCP 头部

注：IP 地址在 IP 包头部。

1.8.1. MTU 与 MSS

MTU 是网络传输**最大报文包**，MSS 是网络传输**数据最大值**。

具体分析如下：

1、mss 加包头数据就等于 mtu. 简单说拿 TCP 包做例子。报文传输 1400 字节的数据的话，那么 mss 就是 1400，再加上 20 字节 IP 包头，20 字节 tcp 包头，那么 mtu 就是 1400+20+20. 当然传输的时候其他的协议还要加些包头在前面，总之 mtu 就是总的最后发出去的报文大小。mss 就是你需要发出去的数据大小。

2、MSS: Maxitum Segment Size 最大分段大小 2.MSS 最大传输大小的缩写，是 TCP 协议里面的一个概念。3.MSS 就是 TCP 数据包每次能够传输的最大数据分段。

3、为了达到最佳的传输效能 TCP 协议在建立连接的时候通常要协商双方的 MSS 值，这个值 TCP 协议在实现的时候往往用 MTU 值代替（需要减去 IP 数据包包头的大小 20Bytes 和 TCP 数据段的包头 20Bytes）所以往往 MSS 为 1460。通讯双方会根据双方提供的 MSS 值得最小值确定为这次连接的最大 MSS 值。

思考：TCP 三次握手时下列哪种情况未发生

A.确认序列号 B.MSS C.滑动窗口大小 D.拥塞控制大小

由于拥塞较复杂，大家自行阅读

<https://www.cnblogs.com/wuchanming/p/4422779.html>

1.9. UDP 协议

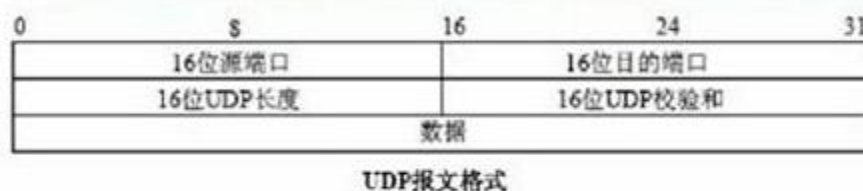
（1）概述

UDP 即**用户数据报协议**，它是一种**无连接协议**，因此不需要像 TCP 那样通过**三次握手来建立一个连接**。同时，一个 UDP 应用可同时作为应用的**客户或服务**方。由于 UDP 协议并不需要建立一个明确的连接，因此建立 UDP 应用要比建立 TCP 应用简单得多。

它比 TCP 协议更为**高效**，也能**更好地解决实时性**的问题。如今，包括网络视频会议系统在内的众多的**客户/服务器模式**的网络应用都使用 UDP 协议。

使用 UDP 协议包括：**TFTP**、**SNMP**、**NFS**、**DNS**、**BOOTP**

（2）Udp 数据包头格式



报头由四个 16 位长（2 字节）字段组成，分别说明该报文的源端口、目的端口、报文长度以及校验值

源、目标端口号字段：占 16 比特。作用与 TCP 数据段中的端口号字段相同，用来标识源端和目标端的应用进程。

●长度字段：占 16 比特。标明 UDP 头部和 UDP 数据的总长度字节。数据报的长度是指包括报头和数据部分在内的总字节数。因为报头的长度是固定的，所以该域主要被用来计算可变长度的数据部分（又称为数据负载）。数据报的最大长度根据操作环境的不同而各异。从理论上说，包含报头在内的数据报的最大长度为 65535 字节。不过，一些实际应用往往会限制数据报的大小，有时会降低到 8192 字节

●校验和字段：占 16 比特。用来对 UDP 头部和 UDP 数据进行校验。和 TCP 不同的是，对 UDP 来说，此字段是可选项，而 TCP 数据段中的校验和字段是必须有的

1.10. 协议的选择

（1）对数据可靠性的要求

对数据要求高可靠性的应用需选择 TCP 协议，如验证、密码字段的传送都是不允许出错的，而对数据的可靠性要求不那么高的应用可选择 UDP 传送。

（2）应用的实时性

TCP 协议在传送过程中要使用三次握手、重传确认等手段来保证数据传输的可靠性。使用 TCP 协议会有较大的时延，因此不适合对实时性要求较高的应用，如 VOIP、视频监控等。相反，UDP 协议则在这些应用中能发挥很好的作用。

（3）网络的可靠性

由于 TCP 协议的提出主要是解决网络的可靠性问题，它通过各种机制来减少错误发生的概率。因此，在网络状况不是很好的情况下需选用 TCP 协议（如在广域网等情况），但是若在网络状况很好的情况下（如局域网等）就不需要再采用 TCP 协议，而建议选择 UDP 协议来减少网络负荷。

TCP 和 UDP，一台机器向另外一台机器发送了 3 个包，对方可能收到几个包？

TCP >=3 3 个到任意多个

UDP 0 到 3 个

1.11. C/S 与 B/S 模式分析

C/S 模式

传统的网络应用设计模式，客户机(client)/服务器(server)模式。需要在通讯两端各自部署客户机和服务器来完成数据通信。

B/S 模式

浏览器() / 服务器(server)模式。只需在一端部署服务器，而另外一端使用每台 PC 都默认配置的浏览器即可完成数据的传输。

优缺点

对于 C/S 模式来说，其优点明显。客户端位于目标主机上可以保证性能，将数据缓存至客户端本地，从而**提高数据传输效率**。且，一般来说客户端和服务程序由一个开发团队创作，所以他们之间**所采用的协议相对灵活**。可以在标准协议的基础上根据需求裁剪及定制。例如，腾讯公司所采用的通信协议，即为 ftp 协议的修改剪裁版。

因此，传统的网络应用程序及较大型的网络应用程序都首选 C/S 模式进行开发。如，知名的网络游戏魔兽世界。3D 画面，数据量庞大，使用 C/S 模式可以提前在本地进行大量数据的缓存处理，从而提高观感。

C/S 模式的缺点也较突出。由于客户端和服务端都需要有一个开发团队来完成开发。**工作量**将成倍提升，开发周期较长。另外，从用户角度出发，需要将客户端安插至用户主机上，对用户主机的**安全性构成威胁**。这也是很多用户不愿使用 C/S 模式应用程序的重要原因。

B/S 模式相比 C/S 模式而言，由于它没有独立的客户端，使用标准浏览器作为客户端，其工作**开发量较小**。只需开发服务器端即可。另外由于其采用浏览器显示数据，因此移植性非常好，**不受平台限制**。如早期的偷菜游戏，在各个平台上都可以完美运行。

B/S 模式的缺点也较明显。由于使用第三方浏览器，因此**网络应用支持受限**。另外，没有客户端放到对方主机上，**缓存数据不尽如人意**，从而传输数据量受到限制。应用的观感大打折扣。第三，必须与浏览器一样，采用标准 http 协议进行通信，**协议选择不灵活**。

因此在开发过程中，模式的选择由上述各自的特点决定。根据实际需求选择应用程序设计模式。

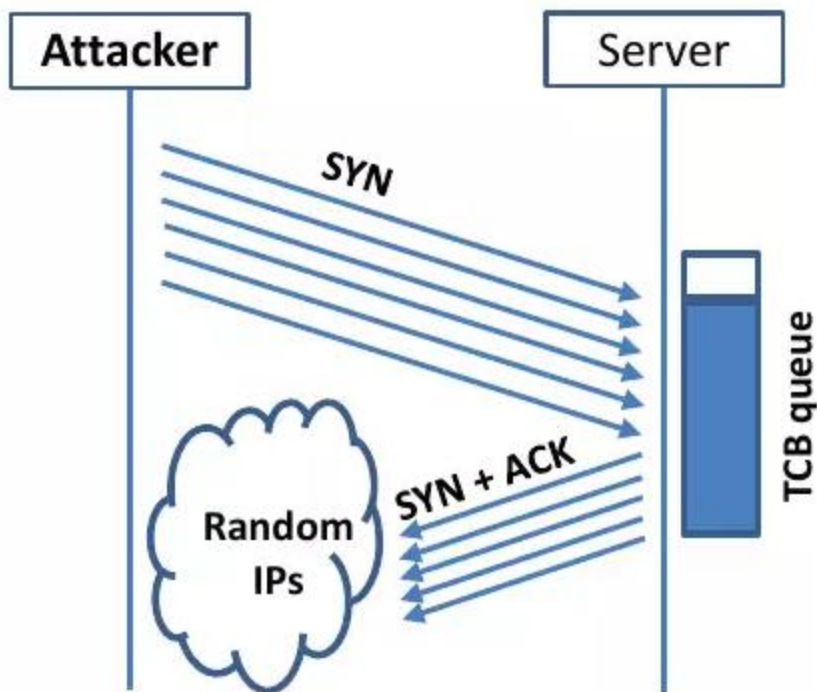
游戏，金融，直播多用 C/S 模式，电商，政企事业主站多用 B/S 模式。

12 什么是 DDos（SYN Flooding）攻击，如何防护

在探讨 SYN 攻击之前，我们先看看 linux 内核对 SYN 是怎么处理的： 1. Server 接收到 SYN 连接请求。内部维护一个队列（我们暂称之为半连接队列，半连接并不准确），发送 ack 及 syn 给 Client 端，等待 Client 端的 ack 应答，接收到则完成三次握手建立连接。如果接收不到 ack 应答，则根据延时重传规则继续发送 ack 及 syn 给客户端。

利用上述特点。我们构造网络包，源地址随机构建，意味着当 Server 接到 SYN 包时，应答 ack 和 syn 时不会得到回应。在这种情况下，Server 端，内核就会维持一个很大的队列来管理这些半连接。当半连接足够多的时候，就会导致新来的正常连接请求得不到响应，也就是所谓的 DOS 攻击。

详细见下图所示：



SYN Flood 攻击防护手段

- tcp_max_syn_backlog: 半连接队列长度
- tcp_synack_retries: syn+ack 的重传次数
- tcp_syncookies : syn dookie

一般的防御措施就是减小 SYN+ACK 重传次数，增加半连接队列长度，启用 syn cookie。不过在高强度攻击面前，调优 tcp_syn_retries 和 tcp_max_syn_backlog 并不能解决根本问题，更有效的防御手段是激活 tcp_syncookies，在连接真正创建起来之前，它并不会立刻给请求分配数据区存储连接状态，而是通过构建一个带签名的序号来屏蔽伪造请求。

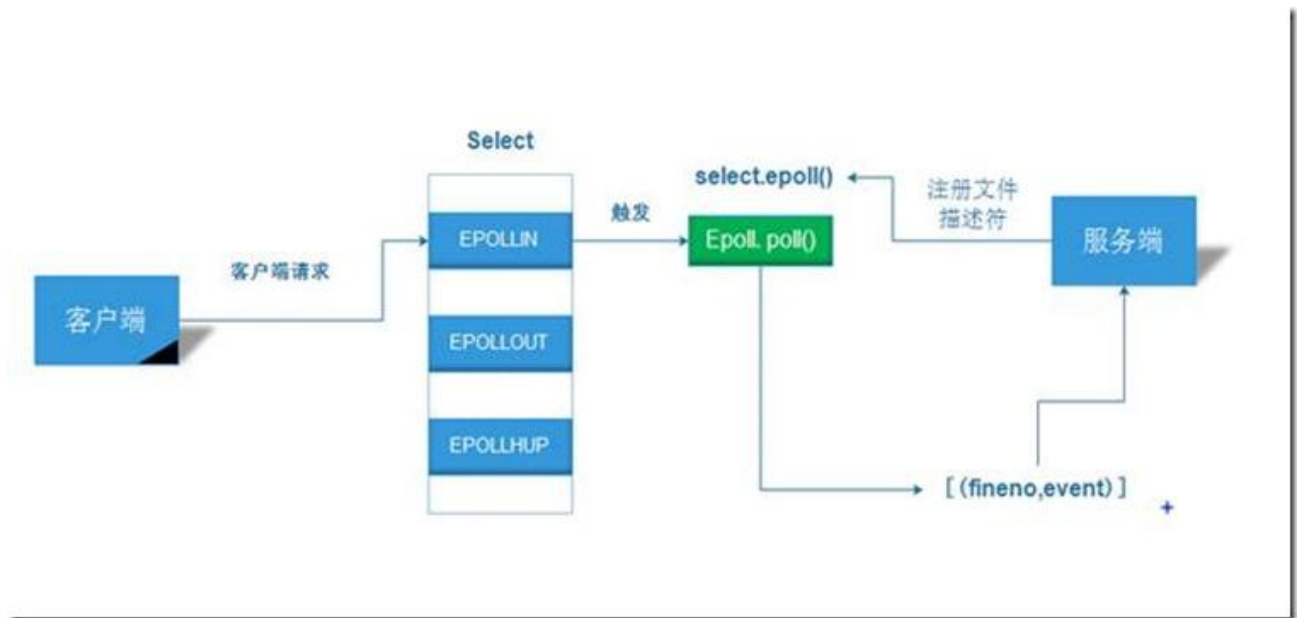
13 网络编程之 epoll 使用

从以上可知，epoll 是对 select、poll 模型的改进，提高了网络编程的性能，广泛应用于大规模并发请求的 C/S 架构中。

1、触发方式：

边缘触发/水平触发，只适用于 Unix/Linux 操作系统

2、原理图



3、一般步骤

1. Create an epoll object——创建 1 个 epoll 对象

2. Tell the epoll object to monitor specific events on specific sockets——告诉 epoll 对象，在指定的 socket 上监听指定的事件
 3. Ask the epoll object which sockets may have had the specified event since the last query——询问 epoll 对象，从上次查询以来，哪些 socket 发生了哪些指定的事件
 4. Perform some action on those sockets——在这些 socket 上执行一些操作
 5. Tell the epoll object to modify the list of sockets and/or events to monitor——告诉 epoll 对象，修改 socket 列表和（或）事件，并监控
 6. Repeat steps 3 through 5 until finished——重复步骤 3-5，直到完成
 7. Destroy the epoll object——销毁 epoll 对象
- #### 4、相关用法

import select 导入 select 模块

epoll = select.epoll() 创建一个 epoll 对象

epoll.register(文件句柄,事件类型) 注册要监控的文件句柄和事件
事件类型:

select.EPOLLIN 可读事件

select.EPOLLOUT 可写事件

select.EPOLLERR 错误事件

select.EPOLLHUP 客户端断开事件

epoll.unregister(文件句柄) 销毁文件句柄

epoll.poll(timeout) 当文件句柄发生变化，则会以列表的形式主动报告给用户进程,timeout
为超时时间，默认为-1，即一直等待直到文件句柄发生变化，如果指定为 1
那么 epoll 每 1 秒汇报一次当前文件句柄的变化情况，若无变化则返回空

epoll.fileno() 返回 epoll 的控制文件描述符(Return the epoll control file descriptor)

epoll.modify(fineno,event) fineno 为文件描述符 event 为事件类型 作用是修改文件描述符所对应的事件

epoll.fromfd(fileno) 从 1 个指定的文件描述符创建 1 个 epoll 对象

epoll.close() 关闭 epoll 对象的控制文件描述符


```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

import socket
import select
import Queue

#创建 socket 对象
serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
#设置 IP 地址复用
serversocket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
#ip 地址和端口号
server_address = ("127.0.0.1", 8888)
#绑定 IP 地址
serversocket.bind(server_address)
#监听，并设置最大连接数
serversocket.listen(10)
print "服务器启动成功，监听 IP: ", server_address
#服务端设置非阻塞
serversocket.setblocking(False)
#超时时间
timeout = 10
#创建 epoll 事件对象，后续要监控的事件添加到其中
epoll = select.epoll()
#注册服务器监听 fd 到等待读事件集合
epoll.register(serversocket.fileno(), select.EPOLLIN)
#保存连接客户端消息的字典，格式为{}
message_queues = {}
```

#文件句柄到所对应对象的字典，格式为{句柄：对象}

fd_to_socket = {serversocket.fileno():serversocket,}

while True:

print "等待活动连接....."

#轮询注册的事件集合，返回值为[(文件句柄，对应的事件)，(...),....]

events = epoll.poll(timeout)

if not events:

print "epoll 超时无活动连接，重新轮询....."

continue

print "有", len(events), "个新事件，开始处理....."

for fd, event in events:

socket = fd_to_socket[fd]

#如果活动 socket 为当前服务器 socket，表示有新连接

if socket == serversocket:

connection, address = serversocket.accept()

print "新连接: ", address

#新连接 socket 设置为非阻塞

connection.setblocking(False)

#注册新连接 fd 到待读事件集合

epoll.register(connection.fileno(), select.EPOLLIN)

#把新连接的文件句柄以及对象保存到字典

fd_to_socket[connection.fileno()] = connection

#以新连接的对象为键值，值存储在队列中，保存每个连接的信息

message_queues[connection] = Queue.Queue()

#关闭事件

elif event & select.EPOLLHUP:

print 'client close'

#在 epoll 中注销客户端的文件句柄

```
    epoll.unregister(fd)
    #关闭客户端的文件句柄

    fd_to_socket[fd].close()
    #在字典中删除与已关闭客户端相关的信息

    del fd_to_socket[fd]
    #可读事件

elif event & select.EPOLLIN:
    #接收数据

    data = socket.recv(1024)

    if data:
        print "收到数据: ", data, "客户端: ", socket.getpeername()
        #将数据放入对应客户端的字典

        message_queues[socket].put(data)
        #修改读取到消息的连接到等待写事件集合(即对应客户端收到消息后, 再将其 fd 修改并
        加入写事件集合)

        epoll.modify(fd, select.EPOLLOUT)
    #可写事件

elif event & select.EPOLLOUT:
    try:
        #从字典中获取对应客户端的信息

        msg = message_queues[socket].get_nowait()

    except Queue.Empty:
        print socket.getpeername(), " queue empty"
        #修改文件句柄为读事件

        epoll.modify(fd, select.EPOLLIN)

    else :
        print "发送数据: ", data, "客户端: ", socket.getpeername()
        #发送数据

        socket.send(msg)
```

```
#在 epoll 中注销服务端文件句柄
epoll.unregister(serversocket.fileno())
#关闭 epoll
epoll.close()
#关闭服务器 socket
serversocket.close()
```

服务端代码

1 即时聊天实现--对方断开怎么办?

TCP 对方断开时，内核把 socket 对象对应的描述符标记为可读状态，epoll 就会检测到，这时候 recv 读时，读到到的内容为空，通过这个来判断，进行断开

```
#!/usr/bin/python3
from socket import *
import select
import sys

tcp_server_socket = socket(AF_INET, SOCK_STREAM)

#重用对应地址和端口
tcp_server_socket.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)

# 本地 IP 地址和端口
address = ('192.168.1.111', 2000)

tcp_server_socket.bind(address)
# 端口激活
tcp_server_socket.listen(100)

client_socket, clientAddr = tcp_server_socket.accept()

print(clientAddr)

# print(tcp_server_socket.fileno())
# print(client_socket.fileno())
## 创建一个 epoll 对象
epoll = select.epoll()
```

```
epoll.register(client_socket.fileno(), select.EPOLLIN)
epoll.register(sys.stdin.fileno(), select.EPOLLIN)
while True:
    epoll_list = epoll.poll()
    for fd, event in epoll_list:
        if fd == sys.stdin.fileno():
            input_data = input()
            if not input_data:
                print('I want go')
                exit(1)
            client_socket.send(input_data.encode('utf-8'))
        if fd == client_socket.fileno() and event == select.EPOLLIN:
            recv_data = client_socket.recv(1024)
            if not recv_data: #通过判断 recv_data 来判断对方是否断开
                print('byebye')
                exit(0) #紧张
            print(recv_data.decode('utf-8'))

client_socket.close()

tcp_server_socket.close()
```

UDP 是什么样子

UDP 即时聊天断开以后，对方不知道的

2 非阻塞编程

1、同步阻塞

正常的 `recv`, `recvfrom` 读取数据会阻塞

2、同步非阻塞编程

轮训

```
#!/usr/bin/python3
```

```
from socket import *
```

```
import select
```

```
import sys
```

```
tcp_server_socket = socket(AF_INET, SOCK_STREAM)
```

```
# 重用对应地址和端口
tcp_server_socket.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)

# 本地 IP 地址和端口
address = ('192.168.1.111', 2000)

tcp_server_socket.bind(address)
# 端口激活
tcp_server_socket.listen(100)
tcp_server_socket.setblocking(False)
client_socket = None
temp_client = None
while True:
    try:
        temp_client, clientAddr = tcp_server_socket.accept()
    except Exception as e:
        # print(e)
        client_socket = temp_client
        if client_socket:
            client_socket.setblocking(False)
            try:
                text = client_socket.recv(1024)
                #如果对方断开
                if not text:
                    print('byebye')
                    exit(0)
                print(text.decode('utf-8'))
            except Exception as e:
                pass
```

3、多路复用编程

客户端:

```
#!/usr/bin/python3
from socket import *
import select
import sys
tcp_client_socket = socket(AF_INET, SOCK_STREAM)

if len(sys.argv) != 2:
    print('./chat_client.py IP')
```

```
        exit(2)
# 本地 IP 地址和端口
address = (sys.argv[1], 2000)
name = input("请输入你的名字: ")
# 连接服务器
tcp_client_socket.connect(address)

epoll = select.epoll()
epoll.register(tcp_client_socket.fileno(), select.EPOLLIN)
epoll.register(sys.stdin.fileno(), select.EPOLLIN)
while True:
    epoll_list = epoll.poll()
    for fd, event in epoll_list:
        if fd == sys.stdin.fileno():
            try: #这里主要是 ctrl+D 后我们能够自己控制退出码
                input_data = input()
            except:
                print('I want go')
                exit(2)
            finally:
                input_data = name + ":" + input_data
                tcp_client_socket.send(input_data.encode('utf-8'))
        if fd == tcp_client_socket.fileno():
            recv_data = tcp_client_socket.recv(1024)
            if not recv_data:
                exit(0)
            print(recv_data.decode('utf-8'))

tcp_client_socket.close()
```

聊天室的代码实现:

服务器端

```
#!/usr/bin/python3
from socket import *
import select
import sys

tcp_server_socket = socket(AF_INET, SOCK_STREAM)

#重用对应地址和端口
```



```
tcp_server_socket.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)

# 本地 IP 地址和端口
address = ('192.168.1.111', 2000)

tcp_server_socket.bind(address)
# 端口激活
tcp_server_socket.listen(100)


# # 创建一个 epoll 对象
epoll = select.epoll()

epoll.register(tcp_server_socket.fileno(), select.EPOLLIN)
#存放聊天室每个用户的 client_socket 信息
client_list = []
while True:
    epoll_list = epoll.poll()
    for fd, event in epoll_list:
        if fd == tcp_server_socket.fileno():
            client_socket, clientAddr = tcp_server_socket.accept()
            #每个连接上的用户添加到列表中, 同时把它注册了
            print("{} is coming".format(clientAddr))
            client_list.append(client_socket)
            epoll.register(client_socket.fileno(), select.EPOLLIN)
        for client in client_list:
            if fd == client.fileno():
                #读取 client 的数据, 群发给其他人, 没数据代表断开, 有数据
发给其他人
                recv_data=client.recv(1024)
                if not recv_data:
                    epoll.unregister(client.fileno())
                    client_list.remove(client)
                    client.close()
                else:
                    for other_client in client_list:
                        if other_client is not client:
                            other_client.send(recv_data)
```

```
client_socket.close()
```

```
tcp_server_socket.close()
```

4、信号驱动编程 ----- Python 中我们不讲信号（虽然也有，大家直接搜 python signal 即可学习）

 游戏里的 event 类似

5、异步编程

 协程

3 写(send)会阻塞么

1、写是会阻塞的，当写满就不能再写，缓冲区不够就阻塞

2、对方断开后，怎么办，判断返回值，或者加入 try 异常判断，避免程序崩溃

MSG_DONTWAIT

通过下面实例去测试：

服务器端，accept 请求后，while True，不做任何事情

```
#!/usr/bin/python3
```

```
from socket import *
```

```
import select
```

```
import sys
```

```
tcp_server_socket = socket(AF_INET, SOCK_STREAM)
```

```
#重用对应地址和端口
```

```
tcp_server_socket.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
```

```
# 本地 IP 地址和端口
```

```
address = ('192.168.1.111', 2000)
```

```
tcp_server_socket.bind(address)
```

```
# 端口激活
```

```
tcp_server_socket.listen(100)
```

```
client_socket, clientAddr = tcp_server_socket.accept()
```

```
print(clientAddr)
```

```
while True:
    pass
客户端:
#!/usr/bin/python3
from socket import *
import select
import sys

tcp_client_socket = socket(AF_INET, SOCK_STREAM)

# 本地 IP 地址和端口
address = ('192.168.1.111', 2000)

# 连接服务器
tcp_client_socket.connect(address)
temp_str = 'a' * 1000
total = 0
try:
    while True:
        send_size = tcp_client_socket.send(temp_str.encode('utf-8'), MSG_DONTWAIT)
        total += send_size
        print(total)
        if send_size != 1000:
            print("peer close")
            exit(1)
except Exception as e:
    print(e)
```

14 持续发送多个文件，协议设计

粘包：两次发送的报文挨在一起，分不开

服务器端如何写：

```
#!/usr/bin/python3
from socket import *
import select
```

```
import sys
import time
import struct
tcp_server_socket = socket(AF_INET, SOCK_STREAM)

# 重用对应地址和端口
tcp_server_socket.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)

# 本地 IP 地址和端口
address = ('192.168.1.112', 2000)

tcp_server_socket.bind(address)
# 端口激活
tcp_server_socket.listen(100)

client_socket, clientAddr = tcp_server_socket.accept()
#连上了
print(clientAddr)

#发文件名
file_name = "Readme"
#先发报文头--文件名的长度
b_file_name = file_name.encode('utf-8')
client_socket.send(struct.pack("I", len(b_file_name)))
client_socket.send(b_file_name)

#发文件内容
file = open(file_name, "rb")
text_bytes = file.read()
client_socket.send(struct.pack("I", len(text_bytes)))
client_socket.send(text_bytes)
file.close()
time.sleep(5)
client_socket.close()
tcp_server_socket.close()
客户端如何写:
```

```
#!/usr/bin/python3
from socket import *
import select
import sys
```

```
import time
import struct
tcp_client_socket = socket(AF_INET, SOCK_STREAM)

# 本地 IP 地址和端口
address = ('192.168.1.112', 2000)

# 连接服务器
tcp_client_socket.connect(address)
time.sleep(1)
# 接文件名
train_len=tcp_client_socket.recv(4)
file_name=tcp_client_socket.recv(struct.unpack('I', train_len)[0])
print(file_name)
file=open(file_name.decode('utf-8'), "wb")

train_len=tcp_client_socket.recv(4)
text_bytes=tcp_client_socket.recv(struct.unpack('I', train_len)[0])
file.write(text_bytes)
file.close()
tcp_client_socket.close()
```

接收方不知道该接收多大的数据才算接收完毕

如何获取文件大小

stat 接口

```
print(os.stat('Readme').st_size)
print(os.stat('Readme').st_ino)
print(hex(os.stat('Readme').st_mode))
print(os.stat('Readme').st_uid)
print(os.stat('Readme').st_gid)
print(time.ctime(os.stat('Readme').st_mtime))
```

如何查看文件下载进度

15 Python 发送整型数，浮点数

<https://blog.csdn.net/yzy1103203312/article/details/78238004>

https://blog.csdn.net/qz_30638831/article/details/80421019

```
import struct
import os
import time
```

```
a = '我很帅你很牛'.encode('utf-8')
```

```
temp_str = struct.pack("I", len(a)) # 转换后的 str 虽然是字符串类型，但
相当于其他语言中的字节流（字节数组），可以在网络上传输
```

```
print('length:', len(temp_str))
print(temp_str)
print(type(temp_str))
```

```
b=struct.unpack('I', temp_str)
print(b[0])
print(type(b[0]))
```

```
temp_big_str = struct.pack(">I", len(a))
```

```
print('length:', len(temp_big_str))
print(temp_big_str)
print(type(temp_big_str))
```

struct模块定义的数据类型表:

Format	C Type	Python type	Standard size	Notes
x	pad byte	no value		
c	char	string of length 1	1	
b	signed char	integer	1	(3)
B	unsigned char	integer	1	(3)
?	_Bool	bool	1	(1)
h	short	integer	2	(3)
H	unsigned short	integer	2	(3)
i	int	integer	4	(3)
I	unsigned int	integer	4	(3)
l	long	integer	4	(3)
L	unsigned long	integer	4	(3)
q	long long	integer	8	(2), (3)
Q	unsigned long long	integer	8	(2), (3)
f	float	float	4	(4)
d	double	float	8	(4)
s	char[]	string		
p	char[]	string		
P	void *	integer		(5), (3)

14、Python 理解大小端

英特尔 x86 小端

ARM 小端

powerpc 大端

Mips 龙芯

整型数，浮点数，要大小端转换，因为 x86，ARM 架构都是小端，遇到龙芯

网络标准 大端

问题

发一个 1G 的电影，TCP 下层是怎么做处理的？

序列号 IP

发一个 1G 的电影，UDP 下层是怎么做处理的？

为什么需要三次握手

二次