

# LINUX 网络编程

## 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}

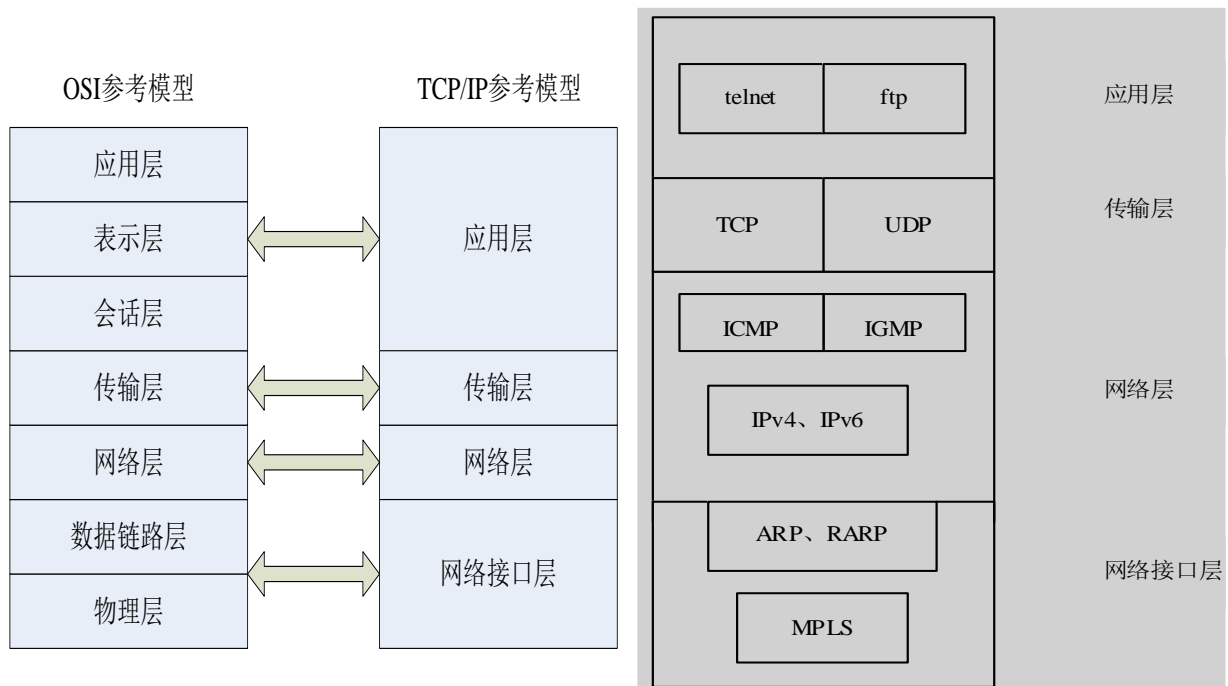
TCP 协议: 传输控制协议 面向连接的协议 能保证传输安全可靠 速度慢 (有 3 次握手)

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



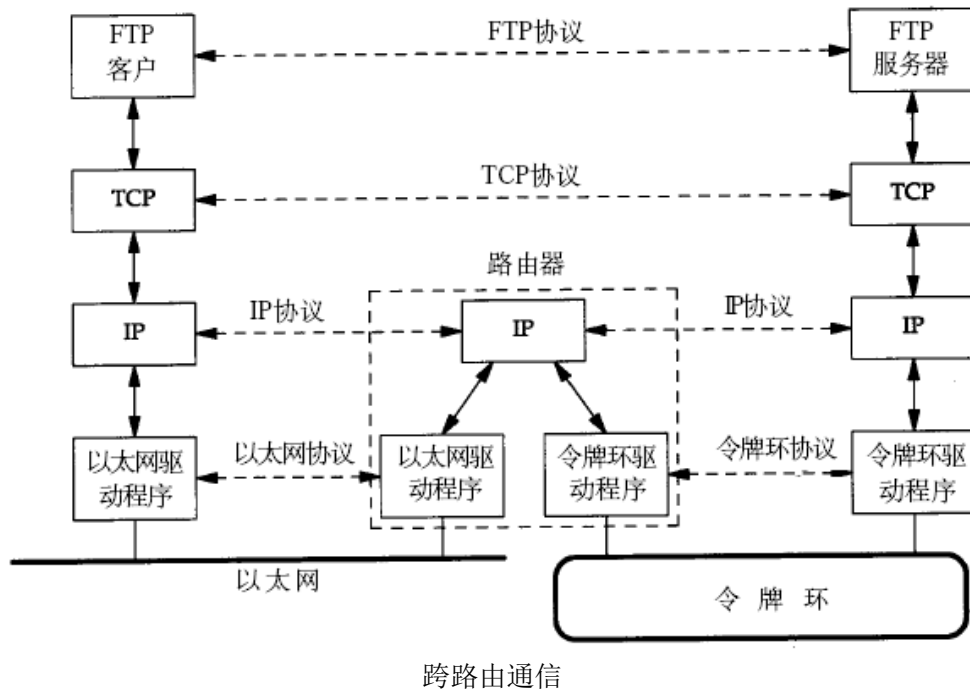
TCP/IP 协议族的每一层的作用:

- 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。



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

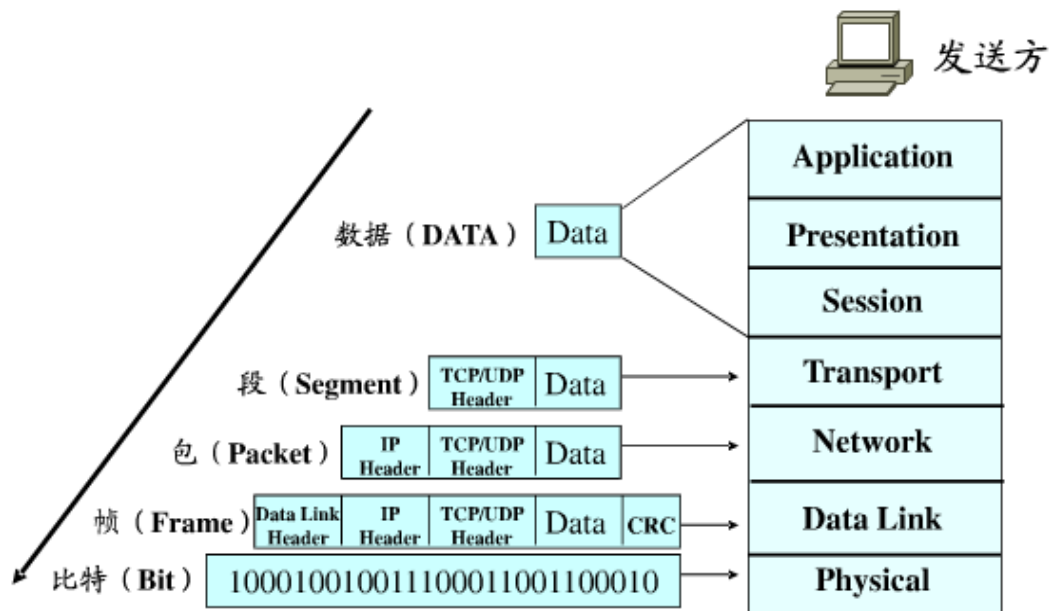
• **IP：（网际互联协议）**负责在主机和网络之间寻址和路由数据包。

• **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。

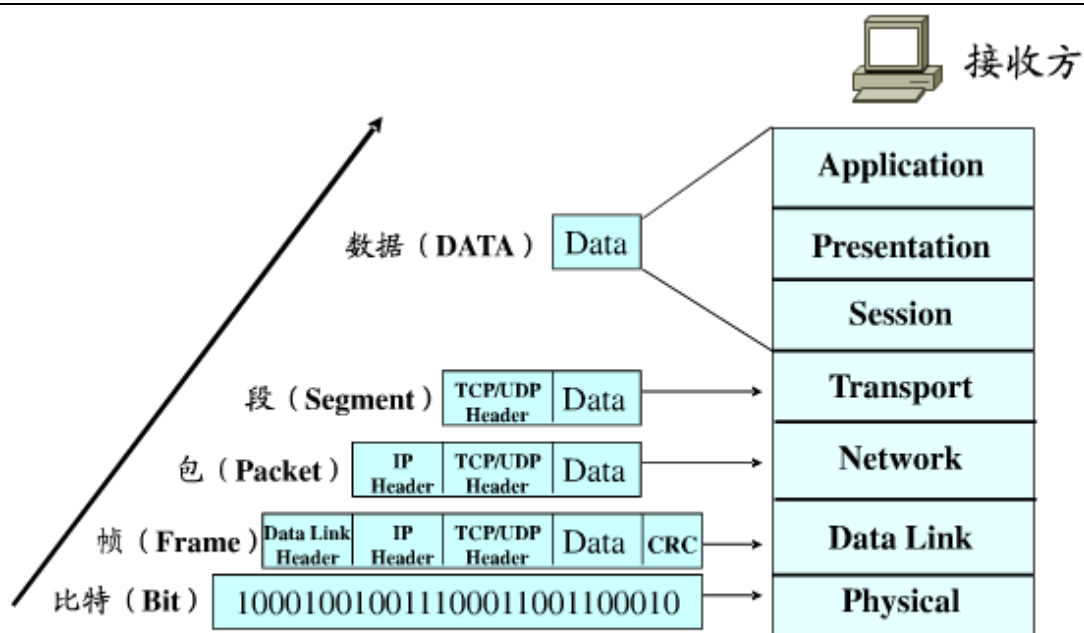
• **IGMP：（网络组管理协议）**被 IP 主机用来向本地多路广播路由器报告主机成员的协议。主机与本地路由器之间使用 Internet 组管理协议（IGMP，Internet Group Management Protocol）来进行组播组成员信息的交互。

• **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。



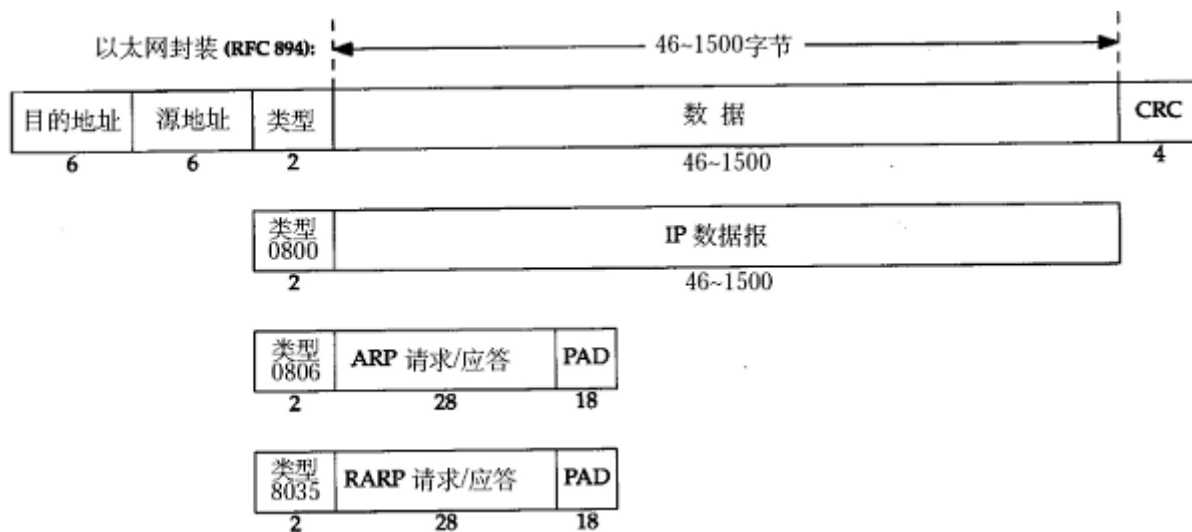
□ 数据封装过程是在不同的层次对数据打上相应的标识



□ 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。

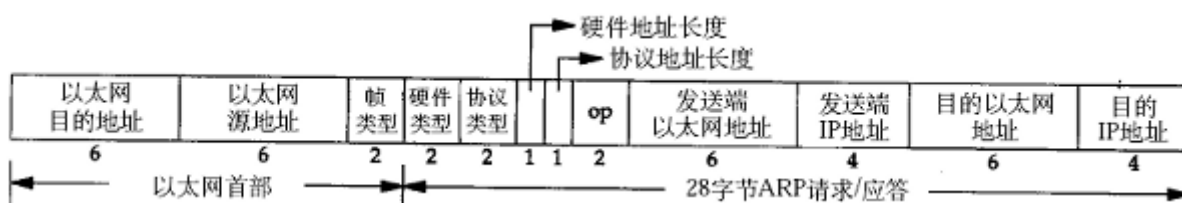
以太网帧中的数据长度规定最小 46 字节，最大 1500 字节，ARP 和 RARP 数据包的长度不够 46 字节，要在后面补填充位。**最大值 1500 称为以太网的最大传输单元（MTU）**，不同的网络类型有不同的 MTU，如果一个数据包从以太网路由到拨号链路上，数据包长度大于拨号链路的 MTU，则需要对数据包进行分片（fragmentation）。ifconfig 命令输出中也有“MTU:1500”。注意，MTU 这个概念指数据帧中有效载荷的最大长度，不包括帧头长度。

## 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:



ARP 数据报格式

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 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

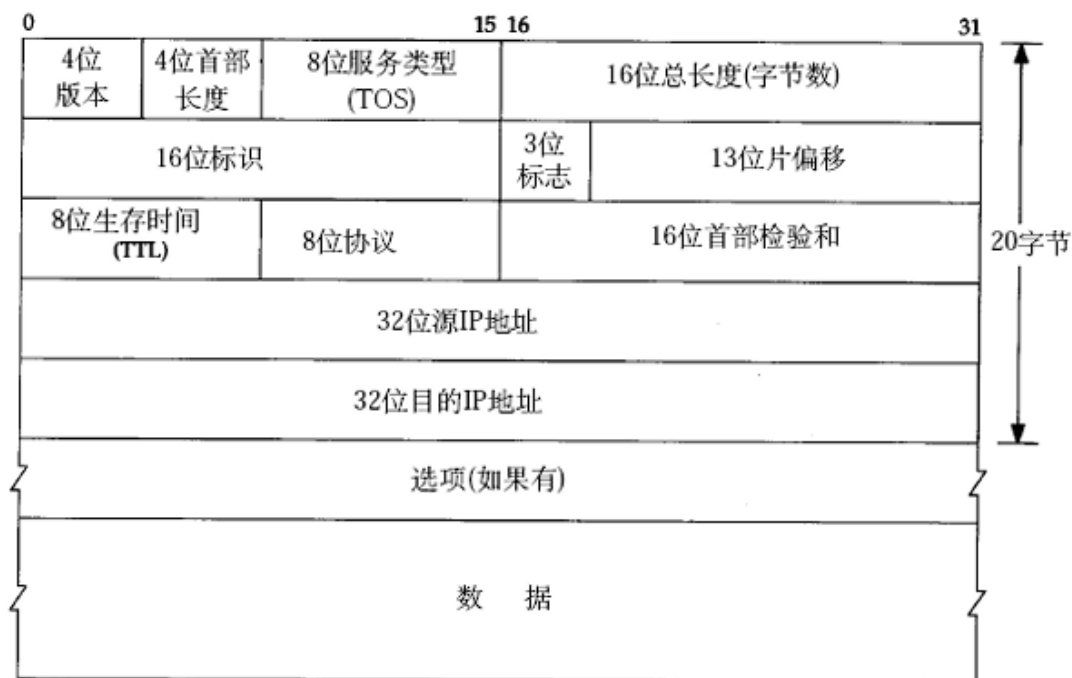
以太网首部：目的主机的 MAC 地址是 00:05:5d:61:58:a8，源主机的 MAC 地址是 00:05:5d:a1:b8:40，上层协议类型 0x0806 表示 ARP。

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 数据包格式



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 DTRC0

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

0 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。

**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 字节的要求用填充字节补上, 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顺序到达目的地后，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，都必须要经过两步操作：

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。

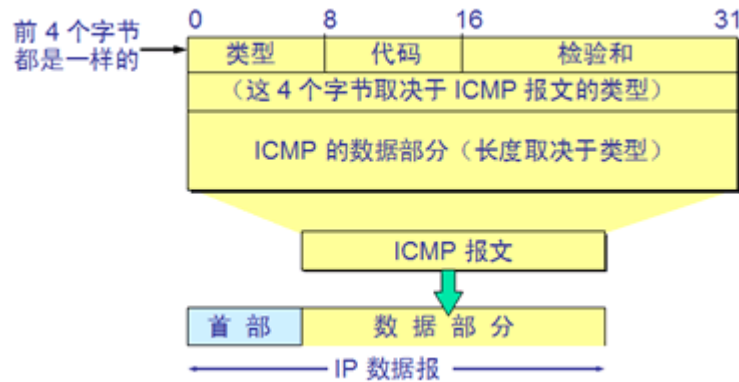
Hub 功能非常弱，目前已经几乎消失殆尽（都购买不着），现在都是 4 口交换机，8 口交换机，或者是无线路由器。

## 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，主要用于在主机、路由器之间传递控制消息，包括报告错误、交换受限控制和状态信息等。**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。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。

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。

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通不通的 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

1.8.1. 概述

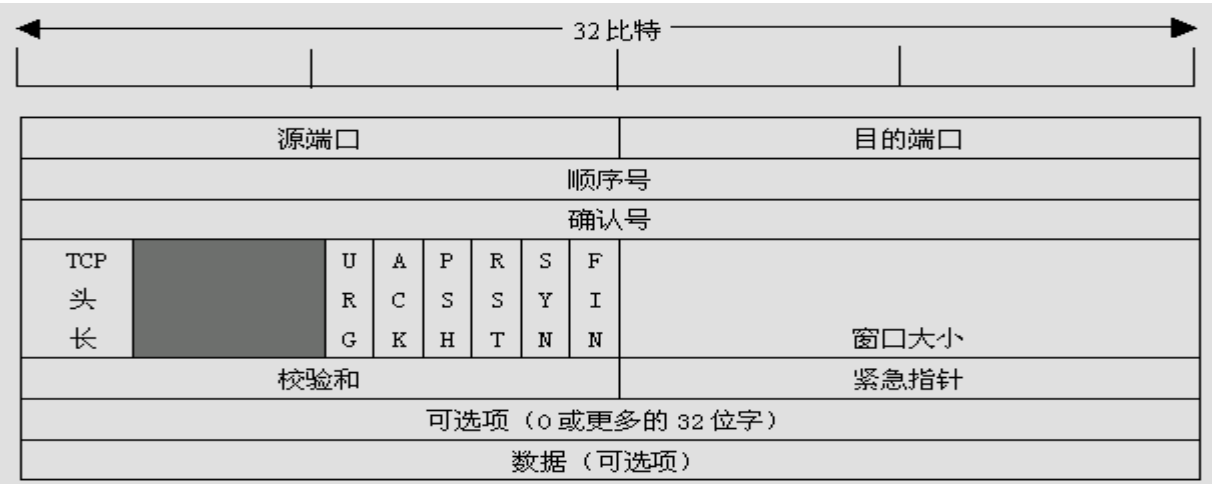
TCP 是 TCP/IP 体系中面向连接的运输层协议，它提供全双工和可靠交付的服务。它采用许多机制来确保端到端结点之间的可靠数据传输，如采用序列号、确认重传、滑动窗口等。

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。

1.8.2. TCP 报文头部格式



- 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。

### 1.8.3. TCP 的三次握手

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:

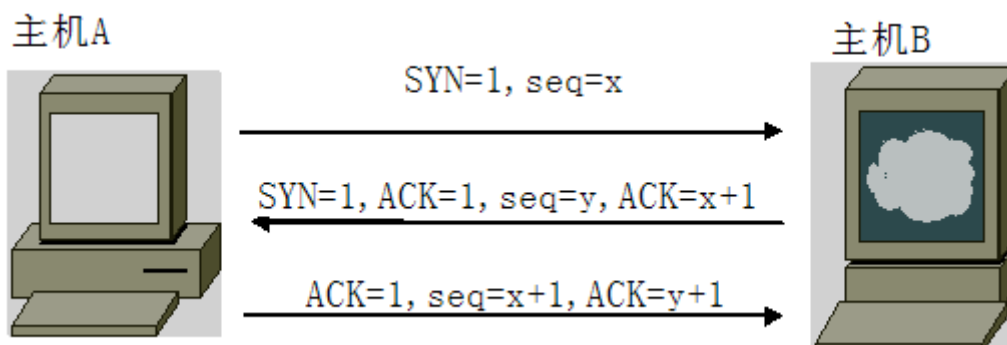
1) 源主机 A 的 TCP 向主机 B 发送连接请求报文段, 其首部中的 SYN (同步) 标志位应置为 1, 表示想跟目标主机 B 建立连接, 进行通信, 并发送一个同步序列号 X (例: SEQ=100) 进行同步, 表明在后面传送数据时的第一个数据字节的序号为 X+1 (即 101)。

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。

### 1.8.4. 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。

TCP 状态:

**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:**

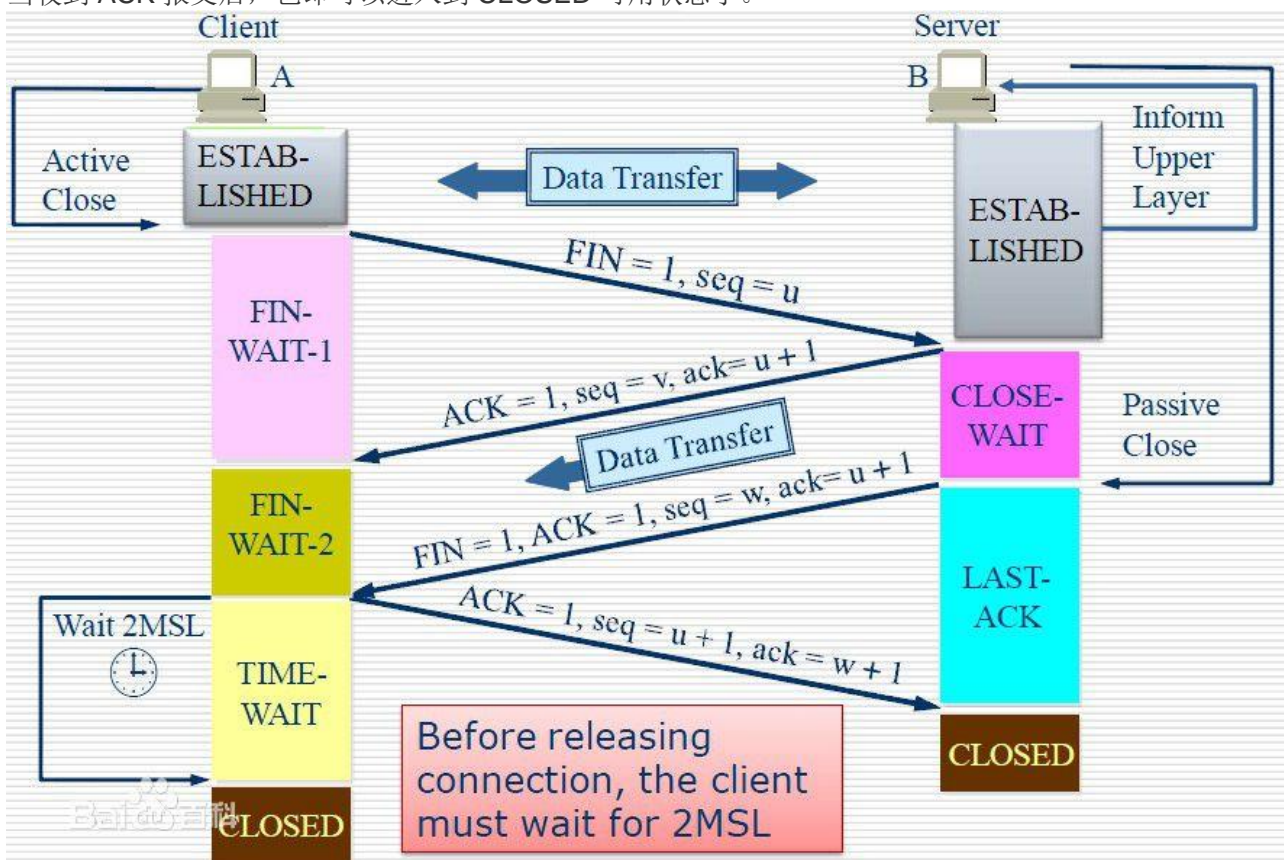
这种状态比较特殊，实际情况中应该是很少见，属于一种比较罕见的例外状态。正常情况下，当你发送 FIN 报文后，按理来说是应该先收到（或同时收到）对方的 ACK 报文，再收到对方的 FIN 报文。但是 CLOSING 状态表示你发送 FIN 报文后，并没有收到对方的 ACK 报文，反而却也收到了对方的 FIN 报文。什么情况下会出现此种情况呢？其实细想一下，也不难得出结论：那就是如果双方几乎在同时 close 一个 SOCKET 的话，那么就出现了双方同时发送 FIN 报文的情况，也就会出现 CLOSING 状态，表示双方都正在关闭 SOCKET 连接。

**CLOSE\_WAIT:**

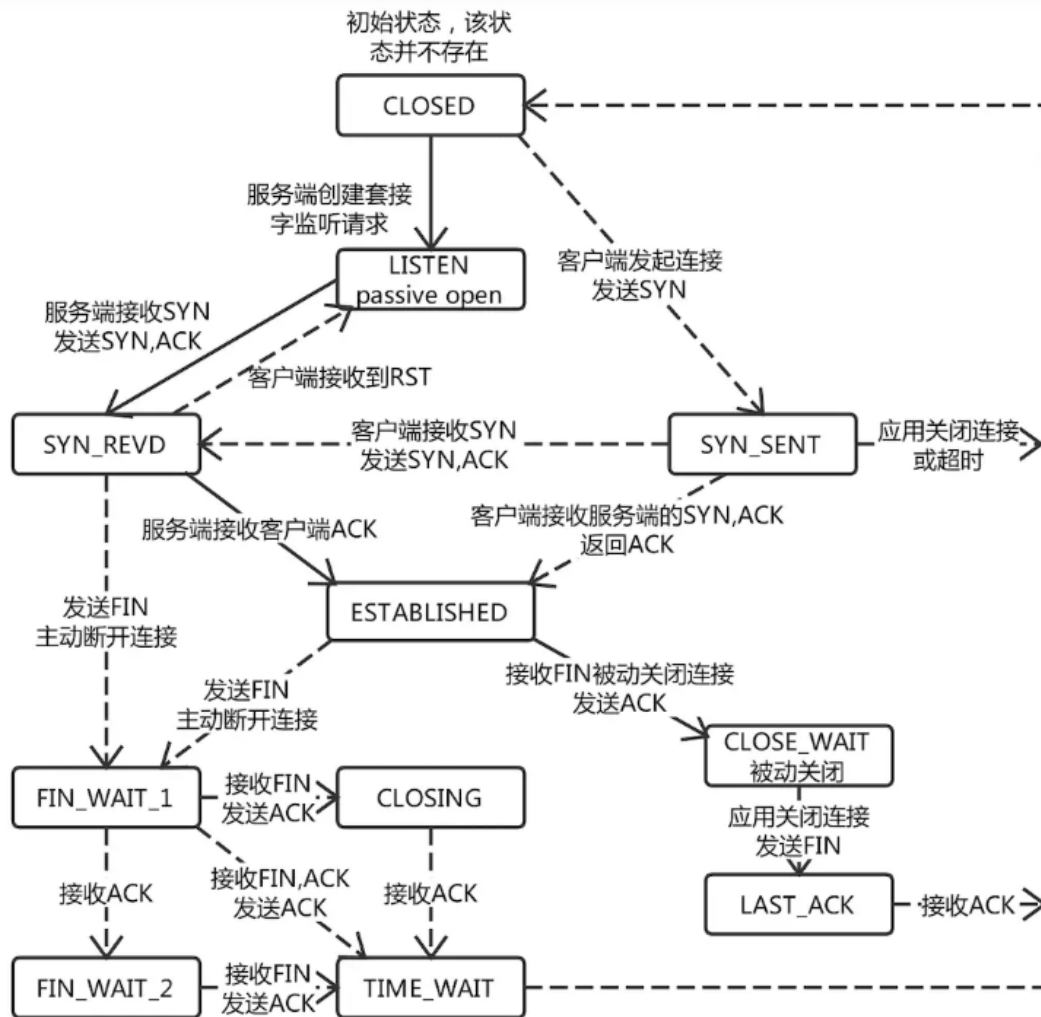
这种状态的含义其实是表示在等待关闭。怎么理解呢？当对方 close 一个 SOCKET 后发送 FIN 报文给自己，你系统毫无疑问地会回应一个 ACK 报文给对方，此时则进入到 CLOSE\_WAIT 状态。接下来呢，实际上你真正需要考虑的事情是查看你是否还有数据发送给对方，如果没有的话，那么你也就可以 close 这个 SOCKET，发送 FIN 报文给对方，也即关闭连接。所以你在 CLOSE\_WAIT 状态下，需要完成的事情是等待你去关闭连接。

**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。







### 1.8.5. MTU 与 MSS

MTU: Maxitum Transmission Unit 最大传输单元。

MSS: Maxitum Segment Size 最大分段大小。

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、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，此字段是可选的，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。

## 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(browser)/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端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。

## 2. 网络相关概念

### 1) 套接口的概念:

套接口，也叫“套接字”。是操作系统内核中的一个数据结构，它是网络中的节点进行相互通信的门户。它是网络进程的 ID。网络通信，归根到底还是进程间的通信（不同计算机上的进程间通信）。在网络中，每一个节点（计算机或路由）都有一个网络地址，也就是 IP 地址。两个进程通信时，首先要确定各自所在的网络节点的网络地址。但是，网络地址只能确定进程所在的计算机，而一台计算机上很可能同时运行着多个进程，所以仅凭网络地址还不能确定到底是和网络中的哪一个进程进行通信，因此套接口中还需要包括其他的信息，也就是端口号（PORT）。在一台计算机中，一个端口号一次只能分配给一个进程，也就是说，在一台计算机中，端口号和进程之间是一一对应关系。所以，使用端口号和网络地址的组合可以唯一的确定整个网络中的一个网络进程。

例如，如网络中某一台计算机的 IP 为 10.92.20.160，操作系统分配给计算机中某一应用程序进程的端口号为 1500，则此时 10.92.20.160 1500 就构成了一个套接口。

### 2) 端口号的概念:

在网络技术中，端口大致有两种意思：一是物理意义上的端口，如集线器、交换机、路由器等用于连接其他网络设备的接口。二是指 TCP/IP 协议中的端口，端口号的范围从 0~65535，一类是由互联网指派名字和号码公司 ICANN 负责分配给一些常用的应用程序固定使用的“周知的端口”，其值一般为 0~1023。

例如 http 的端口号是 80, ftp 为 21, ssh 为 22, telnet 为 23 等。还有一类是用户自己定义的, 通常是大于 1024 的整型值。

3) ip 地址的表示:

通常用户在表达 IP 地址时采用的是点分十进制表示的数值（或者是为冒号分开的十进制 Ipv6 地址），而在通常使用的 socket 编程中使用的则是二进制值，这就需要将这两个数值进行转换。

ipv4 地址: 32bit, 4 字节, 通常采用点分十进制记法。

例如对于: 10000000 00001011 00000011 00011111

点分十进制表示为：128.11.3.31

ip 地址的分类:

A类	7位		24位		
	0	网络号	主机号		0.0.0.0 ~ 127.255.255.255
B类	14位		16位		
	1	0	网络号	主机号	128.0.0.0 ~ 191.255.255.255
C类	21位		8位		
	1	1	0	网络号 主机号	192.0.0.0 ~ 223.255.255.255
D类	28位				
	1	1	1	0	多播组号 224.0.0.0 ~ 239.255.255.255
E类	27位				
	1	1	1	1	0 留待后用 240.0.0.0 ~ 247.255.255.255

特殊的 ip 地址:

网络部分	主机部分	地址类型	用途
Any	全“0”	网络地址	代表一个网段
Any	全“1”	广播地址	特定网段的所有节点
127	any	环回地址	环回测试
全“0”		所有网络	港湾路由器用于指定默认路由
全“1”		广播地址	本网段所有节点

## 2.1. socket 概念

Linux 中的网络编程是通过 **socket 接口**来进行的。**socket** 是一种特殊的 **I/O 接口**，它也是一种**文件描述符**。它是一种常用的**进程之间通信机制**，通过它不仅能实现本地机器上的进程之间的通信，而且通过网络能够在不同机器上的进程之间进行通信。

每一个 **socket** 都用一个半相关描述{协议、本地地址、本地端口}来表示；一个完整的套接字则用一个相关描述{协议、本地地址、本地端口、远程地址、远程端口}来表示。**socket** 也有一个类似于打开文件的函数调用，该函数返回一个整型的 **socket 描述符**，随后的连接建立、数据传输等操作都是通过 **socket** 来实现的；

## 2.2. socket 类型

### (1) 流式 socket (SOCK\_STREAM) →用于 TCP 通信

流式套接字提供可靠的、面向连接的通信流；它使用 **TCP 协议**，从而保证了数据传输的正确性和顺序性。

### (2) 数据报 socket (SOCK\_DGRAM) →用于 UDP 通信

数据报套接字定义了一种无连接的服务，数据通过相互独立的报文进行传输，是无序的，并且不保证是可靠、无差错的。它使用数据报协议 **UDP**。

### (3) 原始 socket (SOCK\_RAW) →用于新的网络协议实现的测试等

原始套接字允许对底层协议如 **IP** 或 **ICMP** 进行直接访问，它功能强大但使用较为不便，主要用于一些协议的开发。

## 2.3. socket 信息数据结构

```
struct sockaddr
{
    unsigned short sa_family; /*地址族*/
    char sa_data[14]; /*14 字节的协议地址，包含该 socket 的 IP 地址和端口号。*/
};
struct sockaddr_in
{
    short int sin_family; /*地址族*/
    unsigned short int sin_port; /*端口号*/
    struct in_addr sin_addr; /*IP 地址*/
    unsigned char sin_zero[8]; /*填充 0 以保持与 struct sockaddr 同样大小*/
};
struct in_addr
{
    in_addr_t s_addr; /* 32 位 IPv4 地址，网络字节序 */
};
```

头文件<netinet/in.h>

sin\_family:AF\_INET →IPv4 协议    AF\_INET6 →IPv6 协议

## 2.4. 数据存储优先顺序的转换

计算机数据存储有两种字节优先顺序：高位字节优先（称为大端模式）和低位字节优先（称为小端模式）。内存的低地址存储数据的低字节，高地址存储数据的高字节的方式叫小端模式。内存的高地址存储数据的低字节，低地址存储数据高字节的方式称为大端模式。

eg: 对于内存中存放的数 0x12345678 来说

如果是采用大端模式存放的，则其真实的数是：0x12345678

如果是采用小端模式存放的，则其真实的数是：0x78563412

如果称某个系统所采用的字节序为主机字节序，则它可能是小端模式的，也可能是大端模式的。而端口号和 IP 地址都是以网络字节序存储的，不是主机字节序，网络字节序都是大端模式。要把主机字节序和网络字节序相互对应起来，需要对这两个字节存储优先顺序进行相互转化。这里用到四个函数：`htons()`、`ntohs()`、`htonl()` 和 `ntohl()`。这四个地址分别实现网络字节序和主机字节序的转化，这里的 h 代表 host, n 代表 network, s 代表 short, l 代表 long。通常 16 位的 IP 端口号用 s 代表，而 IP 地址用 l 来代表。

函数原型如下：

所需头文件	#include <netinet/in.h>
函数原型	<pre>uint16_t htons(uint16_t host16bit); uint32_t htonl(uint32_t host32bit); uint16_t ntohs(uint16_t net16bit); uint32_t ntohl(uint32_t net32bit);</pre>
函数传入值	<p>host16bit: 主机字节序的 16bit 数据</p> <p>host32bit: 主机字节序的 32bit 数据</p> <p>net16bit: 网络字节序的 16bit 数据</p> <p>net32bit: 网络字节序的 32bit 数据</p>
函数返回值	<p>成功: 返回要转换的字节序</p> <p>出错: -1</p>

## 2.5. 地址格式转化

通常用户在表达地址时采用的是点分十进制表示的数值（或者是为冒号分开的十进制 Ipv6 地址），而在通常使用的 socket 编程中使用的则是 32 位的网络字节序的二进制值，这就需要将这两个数值进行转换。这里在 Ipv4 中用到的函数有 `inet_aton()`、`inet_addr()` 和 `inet_ntoa()`，而 IPV4 和 Ipv6 兼容的函数有 `inet_pton()` 和 `inet_ntop()`。

IPv4 的函数原型：

```
#include <sys/socket.h>
```

```
#include <netinet/in.h>
```

```
#include <arpa/inet.h>
```

```
int inet_aton(const char *straddr, struct in_addr *addrptr);
```

```
char *inet_ntoa(struct in_addr inaddr);
```

```
in_addr_t inet_addr(const char *straddr); // in_addr_t 就是 unsigned int, 代表 s_addr
```

函数 `inet_aton()`：将点分十进制数的 IP 地址转换为网络字节序的 32 位二进制数值。返回值：成功，则返回 1，不成功返回 0。

参数 `straddr`：存放输入的点分十进制数 IP 地址字符串。

参数 `addrptr`: 传出参数, 保存网络字节序的 32 位二进制数值。

函数 `inet_ntoa()`: 将网络字节序的 32 位二进制数值转换为点分十进制的 IP 地址。

函数 `inet_addr()`: 功能与 `inet_aton` 相同, 但是结果传递的方式不同。`inet_addr()` 若成功则返回 32 位二进制的网络字节序地址。

IPv4 和 IPv6 的函数原型:

```
#include <arpa/inet.h>
```

```
int inet_pton(int family, const char *src, void *dst);
```

```
const char *inet_ntop(int family, const void *src, char *dst, socklen_t len);
```

函数 `inet_pton` 跟 `inet_aton` 实现的功能类似, 只是多了 `family` 参数, 该参数指定为 `AF_INET`, 表示是 IPv4 协议, 如果是 `AF_INET6`, 表示 IPv6 协议。

函数 `inet_ntop` 跟 `inet_ntoa` 类似, 其中 `len` 表示表示转换之后的长度 (字符串的长度)。

Example:

```
#include <stdio.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int main()
{
    char ip[] = "192.168.0.101";

    struct in_addr myaddr;
    /* inet_aton */
    int iRet = inet_aton(ip, &myaddr);
    printf("%x\n", myaddr.s_addr);

    /* inet_addr */
    printf("%x\n", inet_addr(ip));

    /* inet_pton */
    iRet = inet_pton(AF_INET, ip, &myaddr);
    printf("%x\n", myaddr.s_addr);

    myaddr.s_addr = 0xac100ac4;
    /* inet_ntoa */
    printf("%s\n", inet_ntoa(myaddr));

    /* inet_ntop */
    inet_ntop(AF_INET, &myaddr, ip, 16);
    puts(ip);
    return 0;
}
```

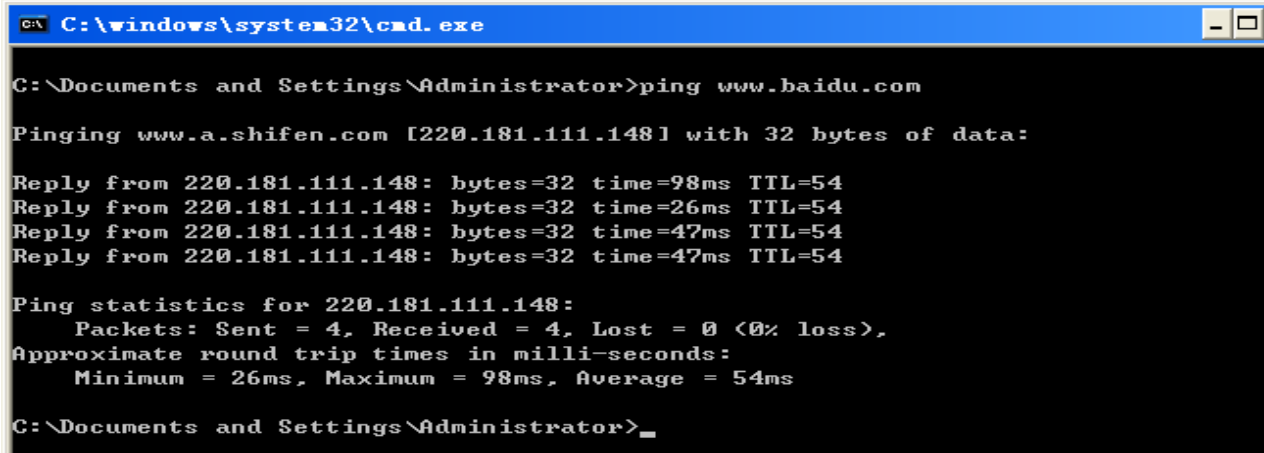
## 2.6. 名字地址转化

通常, 人们在使用过程中都不愿意记忆冗长的 IP 地址, 尤其到 Ipv6 时, 地址长度多达 128 位, 那



时就更加不可能一次性记忆那么长的 IP 地址了。因此，使用主机名或域名将会是很好的选择。主机名与域名的区别：主机名通常在局域网里面使用，通过/etc/hosts 文件，主机名可以解析到对应的 ip；域名通常是在 internet 上使用。

众所周知，百度的域名为：[www.baidu.com](http://www.baidu.com)，而这个域名其实对应了一个百度公司的 IP 地址，那么百度公司的 IP 地址是多少呢？我们可以利用 ping [www.baidu.com](http://www.baidu.com) 来得到百度公司的 ip 地址，如图。那么，系统是如何将 [www.baidu.com](http://www.baidu.com) 这个域名转化为 IP 地址 220.181.111.148 的呢？



```

C:\windows\system32\cmd.exe

C:\Documents and Settings\Administrator>ping www.baidu.com

Pinging www.a.shifen.com [220.181.111.148] with 32 bytes of data:

Reply from 220.181.111.148: bytes=32 time=98ms TTL=54
Reply from 220.181.111.148: bytes=32 time=26ms TTL=54
Reply from 220.181.111.148: bytes=32 time=47ms TTL=54
Reply from 220.181.111.148: bytes=32 time=47ms TTL=54

Ping statistics for 220.181.111.148:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 26ms, Maximum = 98ms, Average = 54ms

C:\Documents and Settings\Administrator>_

```

在 linux 中，有一些函数可以实现主机名和地址的转化，最常见的有 `gethostbyname()`、`gethostbyaddr()` 等，它们都可以实现 IPv4 和 IPv6 的地址和主机名之间的转化。其中 `gethostbyname()` 是将主机名转化为 IP 地址，`gethostbyaddr()` 则是逆操作，是将 IP 地址转化为主机名。

函数原型：

```

#include <netdb.h>

struct hostent* gethostbyname(const char* hostname);
struct hostent* gethostbyaddr(const char* addr, size_t len, int family);

```

结构体：

```

struct hostent
{
    char *h_name;           /*正式主机名*/
    char **h_aliases;       /*主机别名*/
    int h_addrtype;         /*主机 IP 地址类型 IPv4 为 AF_INET*/
    int h_length;           /*主机 IP 地址字节长度，对于 IPv4 是 4 字节，即 32 位*/
    char **h_addr_list;     /*主机的 IP 地址列表*/
}

#define h_addr h_addr_list[0] /*保存的是 ip 地址*/

```

函数 `gethostbyname()`：用于将域名（[www.baidu.com](http://www.baidu.com)）或主机名转换为 IP 地址。参数 `hostname` 指向存放域名或主机名的字符串。

函数 `gethostbyaddr()`：用于将 IP 地址转换为域名或主机名。参数 `addr` 是一个 IP 地址，此时这个 ip 地址不是普通的字符串，而是要通过函数 `inet_aton()` 转换。`len` 为 IP 地址的长度，`AF_INET` 为 4。`family` 可用 `AF_INET`：Ipv4 或 `AF_INET6`：Ipv6。必须在/etc/hosts 中有配置

Example1：将百度的 [www.baidu.com](http://www.baidu.com) 转换为 ip 地址

```

#include <netdb.h>
#include <sys/socket.h>
#include <stdio.h>

int main(int argc, char **argv)

```



```
{
char *ptr, **pptr;
    struct hostent *hptr;
    char str[32] = {'\0'};
    /* 取得命令后第一个参数，即要解析的域名或主机名 */
ptr = argv[1]; //如 www.baidu.com
    /* 调用 gethostbyname()。结果存在 hptr 结构中 */
    if((hptr = gethostbyname(ptr)) == NULL)
    {
        printf(" gethostbyname error for host:%s\n", ptr);
        return 0;
    }
    /* 将主机的规范名打出来 */
    printf("official hostname:%s\n", hptr->h_name);
    /* 主机可能有多个别名，将所有别名分别打出来 */
    for(pptr = hptr->h_aliases; *pptr != NULL; pptr++)
    printf(" alias:%s\n", *pptr);
        /* 根据地址类型，将地址打出来 */
    switch(hptr->h_addrtype)
    {
    case AF_INET:
        case AF_INET6:
pptr=hptr->h_addr_list;
            /* 将刚才得到的所有地址都打出来。其中调用了 inet_ntop()函数 */
            for(; *pptr!=NULL; pptr++)
                printf(" address:%s\n", inet_ntop(hptr->h_addrtype, *pptr, str, sizeof(str)));
                printf(" first address: %s\n", inet_ntop(hptr->h_addrtype, hptr->h_addr, str, sizeof(str)));
            break;
        default:
            printf("unknown address type\n");
            break;
    }
    return 0;
}
```

编译运行

```
#gcc test.c
```

```
#./a.out www.baidu.com
```

```
official hostname:www.a.shifen.com
```

```
alias:www.baidu.com
```

```
address: 220.181.111.148
```

```
.....
```

```
first address: 220.181.111.148
```

(注：这里需要联网才能访问 [www.baidu.com](http://www.baidu.com)。可以尝试用自己的虚拟机的主机名，利用命令 `hostname` 可以查看自己的主机名，用 `hostname -i` 可以查看主机名对应的 ip 地址。那么如何修改主机名呢？直接用 `hostname wangxiao` 只是暂时有效，重启之后就没有了，想要永久有效，需要修改 `/etc/sysconfig/network`

将 HOSTNAME 修改，当然要重启虚拟机。如果 ip 地址不对，你可以修改/etc/hosts 这个文件，添加你自己的主机名对应的 ip 地址）

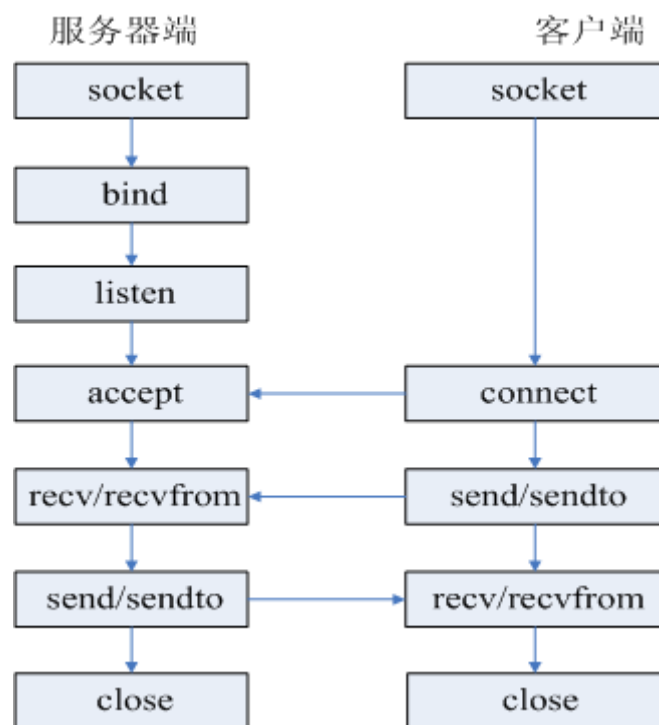
## 3. socket 编程

### 3.1. 使用 TCP 协议的流程图

TCP 通信的基本步骤如下：

服务端：socket---bind---listen---while(1){---accept---recv---send---close---}---close

客户端：socket-----connect---send---recv-----close



服务器端：

1. 头文件包含：

```

#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<unistd.h>
#include<string.h>
#include <stdio.h>
#include <stdlib.h>
  
```

2. socket 函数：生成一个套接口描述符。

原型：int socket(int domain,int type,int protocol);

参数：domain→{ AF\_INET: Ipv4 网络协议 AF\_INET6: Ipv6 网络协议 }

type→ { tcp: SOCK\_STREAM udp: SOCK\_DGRAM }

protocol→指定 socket 所使用的传输协议编号。通常为 0。

返回值：成功则返回套接口描述符，失败返回-1。

常用实例：int sfd = socket(AF\_INET, SOCK\_STREAM, 0);

```
if(sfd == -1){perror("socket");exit(-1);}
```

3. bind 函数：用来绑定一个端口号和 IP 地址，使套接口与指定的端口号和 IP 地址相关联。

原型：int bind(int sockfd, struct sockaddr \* my\_addr, socklen\_t addrlen);

参数：sockfd→为前面 socket 的返回值。

my\_addr→为结构体指针变量

对于不同的 socket domain 定义了一个通用的数据结构

struct sockaddr //此结构体不常用

```
{
unsigned short int sa_family; //调用 socket（）时的 domain 参数，即 AF_INET 值。
char sa_data[14]; //最多使用 14 个字符长度
};
```

此 sockaddr 结构会因使用不同的 socket domain 而有不同结构定义，

例如使用 AF\_INET domain，其 socketaddr 结构定义便为

struct sockaddr\_in //常用的结构体

```
{
unsigned short int sin_family; //即为 sa_family → AF_INET
uint16_t sin_port; //为使用的 port 编号
struct in_addr sin_addr; //为 IP 地址
unsigned char sin_zero[8]; //未使用
};
struct in_addr
{
uint32_t s_addr;
};
```

addrlen→sockaddr 的结构体长度。通常是计算 sizeof(struct sockaddr);

返回值：成功则返回 0，失败返回-1

常用实例：struct sockaddr\_in my\_addr; //定义结构体变量

```
memset(&my_addr, 0, sizeof(struct sockaddr)); //将结构体清空
```

```
//或 bzero(&my_addr, sizeof(struct sockaddr));
```

```
my_addr.sin_family = AF_INET; //表示采用 Ipv4 网络协议
```

```
my_addr.sin_port = htons(8888); //表示端口号为 8888，通常是大于 1024 的一个值。
```

```
//htons()用来将参数指定的 16 位 hostshort 转换成网络字符顺序
```

my\_addr.sin\_addr.s\_addr = inet\_addr("192.168.0.101"); //inet\_addr()用来将 IP 地址字符串转换成网络所使用的二进制数字，如果为 INADDR\_ANY，这表示服务器自动填充本机 IP 地址。

```
if(bind(sfd, (struct sockaddr*)&my_str, sizeof(struct socketaddr)) == -1)
```

```
{perror("bind");close(sfd);exit(-1);}
```

(注：通过将 my\_addr.sin\_port 置为 0，函数会自动为你选择一个未占用的端口来使用。同样，通过将 my\_addr.sin\_addr.s\_addr 置为 INADDR\_ANY，系统会自动填入本机 IP 地址。)

4. listen 函数：使服务器的这个端口和 IP 处于监听状态，等待网络中某一客户机的连接请求。如果客户端有连接请求，端口就会接受这个连接。

原型: `int listen(int sockfd,int backlog);`

参数: `sockfd`→为前面 `socket` 的返回值.即 `sfd`

`backlog`→指定同时能处理的最大连接要求,通常为 10 或者 5。最大值可设至 128

返回值: 成功则返回 0, 失败返回-1

常用实例: `if(listen(sfd, 10) == -1)`  
`{perror("listen");close(sfd);exit(-1);}`

5. `accept` 函数: 接受远程计算机的连接请求,建立起与客户机之间的通信连接。服务器处于监听状态时,如果某时刻获得客户机的连接请求,此时并不是立即处理这个请求,而是将这个请求放在等待队列中,当系统空闲时再处理客户机的连接请求。当 `accept` 函数接受一个连接时,会返回一个新的 `socket` 标识符,以后的数据传输和读取就要通过这个新的 `socket` 编号来处理,原来参数中的 `socket` 也可以继续使用,继续监听其它客户机的连接请求。(也就是说,类似于移动营业厅,如果有客户打电话给 10086,此时服务器就会请求连接,处理一些事务之后,就通知一个话务员接听客户的电话,也就是说,后面的所有操作,此时已经于服务器没有关系,而是话务员跟客户的交流。对应过来,客户请求连接我们的服务器,我们服务器先做了一些绑定和监听等等操作之后,如果允许连接,则调用 `accept` 函数产生一个新的套接字,然后用这个新的套接字跟我们的客户进行收发数据。也就是说,服务器跟一个客户端连接成功,会有两个套接字。)

原型: `int accept(int s,struct sockaddr * addr,socklen_t* addrlen);`

参数: `s`→为前面 `socket` 的返回值.即 `sfd`

`addr`→为结构体指针变量,和 `bind` 的结构体是同种类型的,系统会把远程主机的信息(远程主机的地址和端口号信息)保存到这个指针所指的结构体中。

`addrlen`→表示结构体的长度,为整型指针

返回值: 成功则返回新的 `socket` 处理代码 `new_fd`, 失败返回-1

常用实例: `struct sockaddr_in clientaddr;`  
`memset(&clientaddr, 0, sizeof(struct sockaddr));`  
`int addrlen = sizeof(struct sockaddr);`  
`int new_fd = accept(sfd, (struct sockaddr*)&clientaddr, &addrlen);`  
`if(new_fd == -1)`  
`{perror("accept");close(sfd);exit(-1);}`  
`printf("%s %d success connect\n",inet_ntoa(clientaddr.sin_addr),ntohs(clientaddr.sin_port));`

6. `recv` 函数: 用新的套接字来接收远端主机传来的数据,并把数据存到由参数 `buf` 指向的内存空间

原型: `int recv(int sockfd,void *buf,int len,unsigned int flags);`

参数: `sockfd`→为前面 `accept` 的返回值.即 `new_fd`, 也就是新的套接字。

`buf`→表示缓冲区

`len`→表示缓冲区的长度

`flags`→通常为 0

返回值: 成功则返回实际接收到的字符数,可能会少于你所指定的接收长度。失败返回-1

常用实例: `char buf[512] = {0};`  
`if(recv(new_fd, buf, sizeof(buf), 0) == -1)`  
`{perror("recv");close(new_fd);close(sfd);exit(-1);}`  
`puts(buf);`

7. `send` 函数: 用新的套接字发送数据给指定的远端主机

原型: `int send(int s,const void * msg,int len,unsigned int flags);`

参数: `s`→为前面 `accept` 的返回值.即 `new_fd`

`msg`→一般为常量字符串

`len`→表示长度

`flags`→通常为 0

返回值: 成功则返回实际传送出去的字符数, 可能会少于你所指定的发送长度。失败返回-1

常用实例: `if(send(new_fd, "hello", 6, 0) == -1)`

`{perror("send");close(new_fd);close(sfd);exit(-1);}`

8. `close` 函数: 当使用完文件后若已不再需要则可使用 `close()` 关闭该文件, 并且 `close()` 会让数据写回磁盘, 并释放该文件所占用的资源

原型: `int close(int fd);`

参数: `fd`→为前面的 `sfd,new_fd`

返回值: 若文件顺利关闭则返回 0, 发生错误时返回-1

常用实例: `close(new_fd);`

`close(sfd);`

可以调用 `shutdown` 实现半关闭

`int shutdown(int sockfd, int how);`

客户端:

1. `connect` 函数: 用来请求连接远程服务器, 将参数 `sockfd` 的 `socket` 连至参数 `serv_addr` 指定的服务器 IP 和端口号上去。

原型: `int connect (int sockfd,struct sockaddr * serv_addr,int addrlen);`

参数: `sockfd`→为前面 `socket` 的返回值, 即 `sfd`

`serv_addr`→为结构体指针变量, 存储着远程服务器的 IP 与端口号信息。

`addrlen`→表示结构体变量的长度

返回值: 成功则返回 0, 失败返回-1

常用实例: `struct sockaddr_in seraddr;//请求连接服务器`

`memset(&seraddr, 0, sizeof(struct sockaddr));`

`seraddr.sin_family = AF_INET;`

`seraddr.sin_port = htons(8888); //服务器的端口号`

`seraddr.sin_addr.s_addr = inet_addr("192.168.0.101"); //服务器的 ip`

`if(connect(sfd, (struct sockaddr*)&seraddr, sizeof(struct sockaddr)) == -1)`

`{perror("connect");close(sfd);exit(-1);}`

将上面的头文件以及各个函数中的代码全部拷贝就可以形成一个完整的例子, 此处省略。

Example: 将一些通用的代码全部封装起来, 以后要用直接调用函数即可。如下:

通用网络封装代码头文件: `tcp_net_socket.h`

`#ifndef __TCP__NET__SOCKET__H`

`#define __TCP__NET__SOCKET__H`

`#include <stdio.h>`

`#include <stdlib.h>`

`#include <string.h>`

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <signal.h>
```

```
extern int tcp_init(const char* ip,int port);
extern int tcp_accept(int sfd);
extern int tcp_connect(const char* ip,int port);
extern void signalhandler(void);
```

```
#endif
```

具体的通用函数封装如下： tcp\_net\_socket.c

```
#include "tcp_net_socket.h"
int tcp_init(const char* ip, int port) //用于初始化操作
{
    int sfd = socket(AF_INET, SOCK_STREAM, 0); //首先创建一个 socket，向系统申请
    if(sfd == -1)
    {
        perror("socket");
        exit(-1);
    }
    struct sockaddr_in serveraddr;
    memset(&serveraddr, 0, sizeof(struct sockaddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(port);
    serveraddr.sin_addr.s_addr = inet_addr(ip); //或 INADDR_ANY
    if(bind(sfd, (struct sockaddr*)&serveraddr, sizeof(struct sockaddr)) == -1)
        //将新的 socket 与制定的 ip、port 绑定
    {
        perror("bind");
        close(sfd);
        exit(-1);
    }
    if(listen(sfd, 10) == -1) //监听它，并设置其允许最大的连接数为 10 个
    {
        perror("listen");
        close(sfd);
        exit(-1);
    }
    return sfd;
}
```

int tcp\_accept(int sfd) //用于服务端的接收

```
{
    struct sockaddr_in clientaddr;
    memset(&clientaddr, 0, sizeof(struct sockaddr));
    int addrlen = sizeof(struct sockaddr);
    int new_fd = accept(sfd, (struct sockaddr*)&clientaddr, &addrlen);
    //sfd 接受客户端连接，并创建新的 socket 为 new_fd，将请求连接的客户端的 ip、port 保存在结构体
    clientaddr 中
    if(new_fd == -1)
    {
        perror("accept");
        close(sfd);
        exit(-1);
    }
    printf("%s %d success connect...\n",inet_ntoa(clientaddr.sin_addr),ntohs(clientaddr.sin_port));
    return new_fd;
}
```

int tcp\_connect(const char\* ip, int port) //用于客户端的连接

```
{
    int sfd = socket(AF_INET, SOCK_STREAM, 0); //向系统注册申请新的 socket
    if(sfd == -1)
    {
        perror("socket");
        exit(-1);
    }
    struct sockaddr_in serveraddr;
    memset(&serveraddr, 0, sizeof(struct sockaddr));
    serveraddr.sin_family = AF_INET;
    serveraddr.sin_port = htons(port);
    serveraddr.sin_addr.s_addr = inet_addr(ip);
    if(connect(sfd, (struct sockaddr*)&serveraddr, sizeof(struct sockaddr)) == -1)
    //将 sfd 连接至制定的服务器网络地址 serveraddr
    {
        perror("connect");
        close(sfd);
        exit(-1);
    }
    return sfd;
}
```

void signalhandler(void) //用于信号处理，让服务端在按下 Ctrl+c 或 Ctrl+\的时候不会退出

```
{
    sigset_t sigSet;
    sigemptyset(&sigSet);
```

```
    sigaddset(&sigSet,SIGINT);
    sigaddset(&sigSet,SIGQUIT);
    sigprocmask(SIG_BLOCK,&sigSet,NULL);
}
```

服务器端: tcp\_net\_server.c

```
#include "tcp_net_socket.h"
int main(int argc, char* argv[])
{
    if(argc < 3)
    {
        printf("usage:./servertcp ip port\n");
        exit(-1);
    }
    signalhandler();
    int sfd = tcp_init(argv[1], atoi(argv[2])); //或 int sfd = tcp_init("192.168.0.164", 8888);
    while(1) //用 while 循环表示可以与多个客户端接收和发送, 但仍是阻塞模式的
    {
        int cfd = tcp_accept(sfd);
        char buf[512] = {0};
        if(recv(cfd, buf, sizeof(buf), 0) == -1)//从 cfd 客户端接收数据存于 buf 中
        {
            perror("recv");
            close(cfd);
            close(sfd);
            exit(-1);
        }
        puts(buf);
        if(send(cfd, "hello world", 12, 0) == -1)//从 buf 中取向 cfd 客户端发送数据
        {
            perror("send");
            close(cfd);
            close(sfd);
            exit(-1);
        }
        close(cfd);
    }
    close(sfd);
}
```

客户端: tcp\_net\_client.c

```
#include "tcp_net_socket.h"
int main(int argc, char* argv[])
{
    if(argc < 3)
```



```
{
    printf("usage:./clienttcp ip port\n");
    exit(-1);
}
int sfd = tcp_connect(argv[1],atoi(argv[2]));
char buf[512] = {0};
send(sfd, "hello", 6, 0);    //向 sfd 服务端发送数据
recv(sfd, buf, sizeof(buf), 0); //从 sfd 服务端接收数据
puts(buf);
close(sfd);
}
#gcc -o tcp_net_server tcp_net_server.c tcp_net_socket.c
#gcc -o tcp_net_client tcp_net_client.c tcp_net_socket.c
#./tcp_net_server 192.168.0.164 8888
#./tcp_net_client 192.168.0.164 8888
/* 备注
可以通过 gcc -fpic -c tcp_net_socket.c -o tcp_net_socket.o
gcc -shared tcp_net_socket.o -o libtcp_net_socket.so
cp lib*.so /lib    //这样以后就可以直接使用该库了
cp tcp_net_socket.h /usr/include/    //这样头文件包含可以用 include <tcp_net_socket.h>了
以后再用到的时候就可以直接用：
gcc -o main main.c -ltcp_net_socket //其中 main.c 要包含头文件：include <tcp_net_socket.h>
./main
*/
```

注：上面的虽然可以实现多个客户端访问，但是仍然是阻塞模式（即一个客户访问的时候会阻塞不让另外的客户访问）。解决办法有：

1. 多进程（因为开销比较大，所以不常用）

```
int main(int argc, char* argv[])
{
    if(argc < 3)
    {
        printf("usage:./servertcp ip port\n");
        exit(-1);
    }
    int sfd = tcp_init(argv[1], atoi(argv[2]));
    char buf[512] = {0};
    while(1)
    {
        int cfd = tcp_accept(sfd);
        if(fork() == 0)
        {
            recv(cfd,buf,sizeof(buf),0);
            puts(buf);
            send(cfd,"hello",6,0);
        }
    }
}
```

```
    close(cfd);
    }
    else
    {
    close(cfd);
    }
    }
    close(sfd);
}
```

## 2. 多线程

将服务器上文件的内容全部发给客户端

/\* TCP 文件服务器演示代码 \*/

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/fcntl.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <pthread.h>

#define DEFAULT_SVR_PORT 2828
#define FILE_MAX_LEN 64
char filename[FILE_MAX_LEN+1];

static void * handle_client(void * arg)
{
    int sock = (int)arg;
    char buff[1024];
    int len ;
    printf("begin send\n");
    FILE* file = fopen(filename,"r");
    if(file == NULL)
    {
        close(sock);
        exit;
    }
    //发文件名
    if(send(sock,filename,FILE_MAX_LEN,0) == -1)
    {
        perror("send file name\n");
        goto EXIT_THREAD;
    }
}
```

```
printf("begin send file %s....\n",filename);
//发文件内容
while(!feof(file))
{
    len = fread(buff,1,sizeof(buff),file);
    printf("server read %s,len %d\n",filename,len);
    if(send(sock,buff,len,0) < 0)
    {
        perror("send file:");
        goto EXIT_THREAD;
    }
}
EXIT_THREAD:
if(file)
    fclose(file);
close(sock);
}

int main(int argc,char * argv[])
{
    int sockfd,new_fd;
    //第 1.定义两个 ipv4 地址
    struct sockaddr_in my_addr;
    struct sockaddr_in their_addr;
    int sin_size,numbytes;
    pthread_t cli_thread;
    unsigned short port;
    if(argc < 2)
    {
        printf("need a filename without path\n");
        exit;
    }
    strncpy(filename,argv[1],FILE_MAX_LEN);
    port = DEFAULT_SVR_PORT;
    if(argc >= 3)
    {
        port = (unsigned short)atoi(argv[2]);
    }
    //第一步:建立 TCP 套接字 Socket
    // AF_INET --> ip 通讯
    //SOCK_STREAM -->TCP
    if((sockfd = socket(AF_INET,SOCK_STREAM,0)) == -1)
    {
        perror("socket");
        exit(-1);
    }
}
```

```
}
//第二步:设置侦听端口
//初始化结构体,并绑定 2828 端口
memset(&my_addr,0,sizeof(struct sockaddr));
//memset(&my_addr,0,sizeof(my_addr));
my_addr.sin_family = AF_INET;    /* ipv4 */
my_addr.sin_port = htons(port);  /* 设置侦听端口是 2828,用 htons 转成网络序*/
my_addr.sin_addr.s_addr = INADDR_ANY; /* INADDR_ANY 来表示任意 IP 地址可能其通讯 */
//bzero(&(my_addr.sin_zero),8);
//第三步:绑定套接口,把 socket 队列与端口关联起来.
if(bind(sockfd,(struct sockaddr*)&my_addr,sizeof(struct sockaddr)) == -1)
{
    perror("bind");
    goto EXIT_MAIN;
}
//第四步:开始在 2828 端口侦听,是否有客户端发来联接
if(listen(sockfd,10) == -1)
{
    perror("listen");
    goto EXIT_MAIN;
}
printf("#@ listen port %d\n",port);
//第五步:循环与客户端通讯
while(1)
{
    sin_size = sizeof(struct sockaddr_in);
    printf("server waiting...\n");
    //如果有客户端建立连接,将产生一个全新的套接字 new_fd,专门用于跟这个客户端通信
    if((new_fd = accept(sockfd,(struct sockaddr *)&their_addr,&sin_size)) == -1)
    {
        perror("accept:");
        goto EXIT_MAIN;
    }
    printf("---client (ip=%s:port=%d) request \n",inet_ntoa(their_addr.sin_addr),ntohs(their_addr.sin_port));
    //生成一个线程来完成和客户端的会话,父进程继续监听
    pthread_create(&cli_thread,NULL,handle_client,(void *)new_fd);
}
//第六步:关闭 socket
EXIT_MAIN:
close(sockfd);
return 0;
}
```

```
/* TCP 文件接收客户端 */
#include <stdio.h>
```

```
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <sys/wait.h>

#define FILE_MAX_LEN 64
#define DEFAULT_SVR_PORT 2828

main(int argc, char * argv[])
{
    int sockfd, numbytes;
    char buf[1024], filename[FILE_MAX_LEN+1];
    char ip_addr[64];
    struct hostent *he;
    struct sockaddr_in their_addr;
    int i = 0, len, total;
    unsigned short port;
    FILE * file = NULL;
    if(argc < 2)
    {
        printf("need a server ip \n");
        exit;
    }
    strncpy(ip_addr, argv[1], sizeof(ip_addr));
    port = DEFAULT_SVR_PORT;
    if(argc >= 3)
    {
        port = (unsigned short)atoi(argv[2]);
    }
    //做域名解析(DNS)
    //he = gethostbyname(argv[1]);
    //第一步:建立一个 TCP 套接字
    if((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
        perror("socket");
        exit(1);
    }
    //第二步:设置服务器地址和端口 2828
    memset(&their_addr, 0, sizeof(their_addr));
    their_addr.sin_family = AF_INET;
    their_addr.sin_port = htons(port);
    their_addr.sin_addr.s_addr = inet_addr(ip_addr);
    //their_addr.sin_addr = *((struct in_addr *)he->h_addr);
```

```
//bzero(&(their_addr.sin_zero),8);
printf("connect server %s:%d\n",ip_addr,port);
/*第三步:用 connect 和服务端建立连接 ,注意,这里没有使用本地端口,将由协议栈自动分配一个端口*/
if(connect(sockfd,(struct sockaddr *)&their_addr,sizeof(struct sockaddr))== -1){
    perror("connect");
    exit(1);
}
if(send(sockfd,"hello",6,0)< 0)
{
    perror("send ");
    exit(1);
}
/* 接收文件名,为编程简单,假设前 64 字节固定是文件名,不足用 0 来增充 */
total = 0;
while(total< FILE_MAX_LEN){
    /* 注意这里的接收 buffer 长度,始终是未接收文件名称剩下的长度,*/
    len = recv(sockfd,filename+total,(FILE_MAX_LEN - total),0);
    if(len <= 0)
        break;
    total += len ;
}
/* 接收文件名出错 */
if(total != FILE_MAX_LEN){
    perror("failure file name");
    exit(-3);
}
printf("recv file %s.....\n",filename);
file = fopen(filename,"wb");
//file = fopen("/home/hxy/abc.txt","wb");
if(file == NULL)
{
    printf("create file %s failure",filename);
    perror("create:");
    exit(-3);
}
//接收文件数据
printf("recv begin\n");
total = 0;
while(1)
{
    len = recv(sockfd,buf,sizeof(buf),0);
    if(len == -1)
        break;
    total += len;
```

```
        //写入本地文件
        fwrite(buf,1,len,file);
    }
    fclose(file);
    printf("recv file %s success total lenght %d\n",filename,total);
    //第六步:关闭 socket
    close(sockfd);
}
/* 备注读写大容量的文件时, 通过下面的方法效率很高
ssize_t readn(int fd,char *buf,int size)//读大量内容
```

```
{
    char *pbuf=buf;
    int total ,nread;
    for(total = 0; total < size; )
    {
        nread=read(fd,pbuf,size-total);
        if(nread==0)
            return total;
        if(nread == -1)
        {
            if(errno == EINTR)
                continue;
            else
                return -1;
        }
        total+= nread;
        pbuf+=nread;
    }
    return total;
}
```

```
ssize_t writen(int fd, char *buf, int size)//写大量内容
```

```
{
    char *pbuf=buf;
    int total ,nwrite;
    for(total = 0; total < size; )
    {
        nwrite=write(fd,pbuf,size-total);
        if( nwrite <= 0 )
        {
            if( nwrite == -1 && errno == EINTR )
                continue;
            else
                return -1;
        }
        total += nwrite;
    }
}
```

```
        pbuf += nwrite;
    }
    return total;
}
*/
```

3. 调用 `fcntl` 将 `sockfd` 设置为非阻塞模式。

```
#include <unistd.h>
#include <fcntl.h>
.....
sockfd = socket(AF_INET, SOCK_STREAM, 0);
iflags = fcntl(sockfd, F_GETFL, 0);
fcntl(sockfd, F_SETFL, O_NONBLOCK | iflags);
.....
```

4. 多路选择 `select`

```
#include <sys/select.h>
#include "tcp_net_socket.h"
#define MAXCLIENT 10
main()
{
    int sfd = tcp_init("192.168.0.164", 8888);
    int fd = 0;
    char buf[512] = {0};
    fd_set rdset;
    while(1)
    {
        FD_ZERO(&rdset);
        FD_SET(sfd, &rdset);
        if(select(MAXCLIENT + 1, &rdset, NULL, NULL, NULL) < 0)
            continue;
        for(fd = 0; fd < MAXCLIENT; fd++)
        {
            if(FD_ISSET(fd, &rdset))
            {
                if(fd == sfd)
                {
                    int cfd = tcp_accept(sfd);
                    FD_SET(cfd, &rdset);
                    //.....
                }
            }
            else
            {
                bzero(buf, sizeof(buf));
                recv(fd, buf, sizeof(buf), 0);
                puts(buf);
                send(fd, "java", 5, 0);
            }
        }
    }
}
```



```
// FD_CLR(fd, &rdset);
close(fd);
}
}
}
close(sfd);
}
```

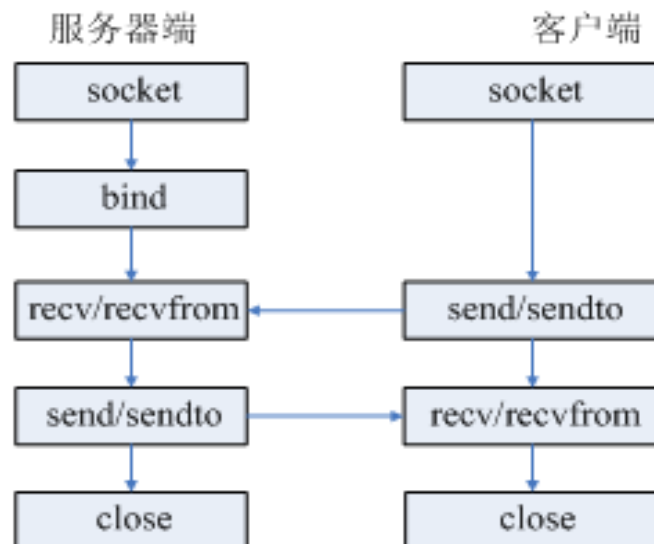
具体例子请参考《网络编程之 select.doc》或《tcp\_select》

## 3.2. 使用 UDP 协议的流程图

UDP 通信流程图如下：

服务端：socket---bind---recvfrom---sendto---close

客户端：socket-----sendto---recvfrom---close



- sendto()函数原型：

```
int sendto(int sockfd, const void *msg, int len, unsigned int flags, const struct sockaddr *to, socklen_t tolen);
```

该函数比 send()函数多了两个参数，to 表示目地机的 IP 地址和端口号信息，而 tolen 常常被赋值为 sizeof(struct sockaddr)。sendto 函数也返回实际发送的数据字节长度或在出现发送错误时返回-1。

- recvfrom()函数原型：

```
int recvfrom(int sockfd, void *buf, int len, unsigned int flags, struct sockaddr *from, int *fromlen);
```

from 是一个 struct sockaddr 类型的变量，该变量保存连接机的 IP 地址及端口号。fromlen 常置为 sizeof(struct sockaddr)。当 recvfrom()返回时，fromlen 包含实际存入 from 中的数据字节数。Recvfrom()函数返回接收到的字节数或当出现错误时返回-1，并置相应的 errno。

Example:UDP 的基本操作

服务器端：

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
```

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
main()
{
    int sfd = socket(AF_INET, SOCK_DGRAM, 0);
    if(sfd == -1)
    {
        perror("socket");
        exit(-1);
    }

    struct sockaddr_in saddr;
    bzero(&saddr, sizeof(saddr));
    saddr.sin_family = AF_INET;
    saddr.sin_port = htons(8888);
    saddr.sin_addr.s_addr = INADDR_ANY;
    if(bind(sfd, (struct sockaddr*)&saddr, sizeof(struct sockaddr)) == -1)
    {
        perror("bind");
        close(sfd);
        exit(-1);
    }

    char buf[512] = {0};
    while(1)
    {
        struct sockaddr_in fromaddr;
        bzero(&fromaddr, sizeof(fromaddr));
        int fromaddrlen = sizeof(struct sockaddr);
        if(recvfrom(sfd, buf, sizeof(buf), 0, (struct sockaddr*)&fromaddr, &fromaddrlen) == -1)
        {
            perror("recvfrom");
            close(sfd);
            exit(-1);
        }
        printf("receive from %s %d,the message is:%s\n", inet_ntoa(fromaddr.sin_addr),
            ntohs(fromaddr.sin_port), buf);

        sendto(sfd, "world", 6, 0, (struct sockaddr*)&fromaddr, sizeof(struct sockaddr));
    }

    close(sfd);
}
```

客户端:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char* argv[])
{
    int sfd = socket(AF_INET, SOCK_DGRAM, 0);
    if(sfd == -1)
    {
        perror("socket");
        exit(-1);
    }

    struct sockaddr_in toaddr;
    bzero(&toaddr, sizeof(toaddr));
    toaddr.sin_family = AF_INET;
    toaddr.sin_port = htons(atoi(argv[2])); //此处的端口号要跟服务器一样
    toaddr.sin_addr.s_addr = inet_addr(argv[1]); //此处为服务器的 ip
    sendto(sfd, "hello", 6, 0, (struct sockaddr*)&toaddr, sizeof(struct sockaddr));

    char buf[512] = {0};
    struct sockaddr_in fromaddr;
    bzero(&fromaddr, sizeof(fromaddr));
    int fromaddrlen = sizeof(struct sockaddr);
    if(recvfrom(sfd, buf, sizeof(buf), 0, (struct sockaddr*)&fromaddr, &fromaddrlen) == -1)
    {
        perror("recvfrom");
        close(sfd);
        exit(-1);
    }
    printf("receive from %s %d,the message is:%s\n", inet_ntoa(fromaddr.sin_addr),
    ntohs(fromaddr.sin_port), buf);

    close(sfd);
}
```

Example: UDP 发送文件先发文件大小再发文件内容

//服务器端

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
```

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
main()
{
    int sfd = socket(AF_INET, SOCK_DGRAM, 0);
    if(sfd == -1)
    {
        perror("socket");
        exit(-1);
    }

    struct sockaddr_in saddr;
    bzero(&saddr, sizeof(saddr));
    saddr.sin_family = AF_INET;
    saddr.sin_port = htons(8888);
    saddr.sin_addr.s_addr = INADDR_ANY;
    if(bind(sfd, (struct sockaddr*)&saddr, sizeof(struct sockaddr)) == -1)
    {
        perror("bind");
        close(sfd);
        exit(-1);
    }

    char buf[512] = {0};
    struct sockaddr_in fromaddr;
    bzero(&fromaddr, sizeof(fromaddr));
    int fromaddrlen = sizeof(struct sockaddr);
    if(recvfrom(sfd, buf, sizeof(buf), 0, (struct sockaddr*)&fromaddr, &fromaddrlen) == -1)
    {
        perror("recvfrom");
        close(sfd);
        exit(-1);
    }
    printf("receive from %s %d,the message is:%s\n", inet_ntoa(fromaddr.sin_addr),
    ntohs(fromaddr.sin_port), buf);

    FILE* fp = fopen("1.txt","rb");
    struct stat st; //用于获取文件内容的大小
    stat("1.txt", &st);
    int filelen = st.st_size;
    sendto(sfd, (void*)&filelen, sizeof(int), 0, (struct sockaddr*)&fromaddr, sizeof(struct sockaddr));
```

```
while(!feof(fp))    //表示没有到文件尾
{
    int len = fread(buf,1,sizeof(buf),fp);
    sendto(sfd, buf, len, 0, (struct sockaddr*)&fromaddr, sizeof(struct sockaddr));
}

close(sfd);
}
```

//客户端

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#define BUFSIZE 512
int main(int argc, char* argv[])
{
    int sfd = socket(AF_INET, SOCK_DGRAM, 0);
    if(sfd == -1)
    {
        perror("socket");
        exit(-1);
    }

    struct sockaddr_in toaddr;
    bzero(&toaddr, sizeof(toaddr));
    toaddr.sin_family = AF_INET;
    toaddr.sin_port = htons(atoi(argv[2]));
    toaddr.sin_addr.s_addr = inet_addr(argv[1]);
    sendto(sfd, "hello", 6, 0, (struct sockaddr*)&toaddr, sizeof(struct sockaddr));

    char buf[BUFSIZE] = {0};
    struct sockaddr_in fromaddr;
    bzero(&fromaddr, sizeof(fromaddr));
    int fromaddrlen = sizeof(struct sockaddr);
    int filelen = 0;    //用于保存文件长度
    FILE* fp = fopen("2.txt","w+b");
    //接收文件的长度
    recvfrom(sfd, (void*)&filelen, sizeof(int), 0, (struct sockaddr*)&fromaddr, &fromaddrlen);
    printf("the length of file is %d\n",filelen);
    printf("Create a new file!\n");
}
```

```

    printf("begin to receive file content!\n");
//接收文件的内容
    while(1)
    {
        int len = recvfrom(sfd, buf, sizeof(buf), 0, (struct sockaddr*)&fromaddr, &fromaddrlen);
        if(len < BUFSIZE)
            //如果接收的长度小于 BUFSIZE, 则表示最后一次接收, 此时要用 break 退出循环
        {
            fwrite(buf, sizeof(char), len, fp);
            break;
        }
        fwrite(buf, sizeof(char), len, fp);
    }
    printf("receive file finished!\n");
    close(sfd);
}

```

注意：操作系统的 UDP 接收流程如下：收到一个 UDP 包后，验证没有错误后，放入一个包队列中，队列中的每一个元素就是一个完整的 UDP 包。当应用程序通过 `recvfrom()` 读取时，OS 把相应的一个完整 UDP 包取出，然后拷贝到用户提供的内存中，物理用户提供的内存大小是多少，OS 都会完整取出一个 UDP 包。如果用户提供的内存小于这个 UDP 包的大小，那么在填充慢内存后，UDP 包剩余的部分就会被丢弃，以后再也无法取回。

这与 TCP 接收完全不同，TCP 没有完整包的概念，也没有边界，OS 只会取出用户要求的大小，剩余的仍然保留在 OS 中，下次还可以继续取出。

思考题：机器 A 向机器 B 发送数据，以 TCP 方式发送 3 个包，对方可能会收到几个包，以 UDP 方式发送 3 个包，对方可能会收到几个包

### 3.3. 设置套接口的选项 `setsockopt` 的用法

函数原型：

```

#include <sys/types.h>
#include <sys/socket.h>
int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);

```

`sockfd`: 标识一个套接口的描述符

`level`: 选项定义的层次；支持 `SOL_SOCKET`、`IPPROTO_TCP`、`IPPROTO_IP` 和 `IPPROTO_IPV6`

`optname`: 需设置的选项

`optval`: 指针，指向存放选项值的缓冲区

`optlen`: `optval` 缓冲区长度

选项名称	说明	数据类型
------	----	------

=====		
<code>SOL_SOCKET</code>		

SO_BROADCAST	允许发送广播数据	int
SO_DEBUG	允许调试	int
SO_DONTROUTE	不查找路由	int
SO_ERROR	获得套接字错误	int
SO_KEEPALIVE	保持连接	int
SO_LINGER	延迟关闭连接	struct linger
SO_OOBINLINE	带外数据放入正常数据流	int
SO_RCVBUF	接收缓冲区大小	int
SO_SNDBUF	发送缓冲区大小	int
SO_RCVLOWAT	接收缓冲区下限	int
SO_SNDLOWAT	发送缓冲区下限	int
SO_RCVTIMEO	接收超时	struct timeval
SO_SNDTIMEO	发送超时	struct timeval
SO_REUSEADDR	允许重用本地地址和端口	int
SO_TYPE	获得套接字类型	int
SO_BSDCOMPAT	与 BSD 系统兼容	int

## IPPROTO\_IP

IP_HDRINCL	在数据包中包含 IP 首部	int
IP_OPTIONS	IP 首部选项	int
IP_TOS	服务类型	
IP_TTL	生存时间	int

## IPPROTO\_TCP

TCP_MAXSEG	TCP 最大数据段的大小	int
TCP_NODELAY	不使用 Nagle 算法	int

全部都必须要放在 bind 之前，另外通常是用于 UDP 的。

1. 如果在已经处于 ESTABLISHED 状态下的 socket(一般由端口号和标志符区分)调用 closesocket(一般不会立即关闭而经历 TIME\_WAIT 的过程)后想继续重用该 socket:

```
int reuse=1;
```

```
setsockopt(s,SOL_SOCKET,SO_REUSEADDR,(const char*)&reuse,sizeof(int));
```

2.在 send(),recv()过程中有时由于网络状况等原因，收发不能预期进行,而设置收发时限:

```
int nNetTimeout=1000; // 1 秒
```

```
// 发送时限
```

```
setsockopt(socket, SOL_SOCKET,SO_SNDTIMEO, (char *)&nNetTimeout,sizeof(int));
```

```
// 接收时限
```

```
setsockopt(socket, SOL_SOCKET,SO_RCVTIMEO, (char *)&nNetTimeout,sizeof(int));
```

3.在 send()的时候，返回的是实际发送出去的字节(同步)或发送到 socket 缓冲区的字节(异步),系统默认的状态发送和接收一次为 8688 字节(约为 8.5K);在实际的过程中发送数据和接收数据量比较大，可以设置 socket 缓冲区，而避免了 send(),recv()不断的循环收发:

```
// 接收缓冲区
```

```
int nRecvBuf=32*1024;    // 设置为 32K
setsockopt(s,SOL_SOCKET,SO_RCVBUF,(const char*)&nRecvBuf,sizeof(int));
// 发送缓冲区
int nSendBuf=32*1024;    // 设置为 32K
setsockopt(s,SOL_SOCKET,SO_SNDBUF,(const char*)&nSendBuf,sizeof(int));
4. 如果在发送数据时, 希望不经历由系统缓冲区到 socket 缓冲区的拷贝而影响程序的性能:
int nZero=0;
setsockopt(socket, SOL_SOCKET,SO_SNDBUF, (char *)&nZero,sizeof(int));
5.同在上在 recv()完成上述功能(默认情况是将 socket 缓冲区的内容拷贝到系统缓冲区):
int nZero=0;
setsockopt(socket, SOL_SOCKET,SO_RCVBUF, (char *)&nZero,sizeof(int));
6.一般在发送 UDP 数据报的时候, 希望该 socket 发送的数据具有广播特性:
int bBroadcast= 1;
setsockopt(s,SOL_SOCKET,SO_BROADCAST,(const char*)&bBroadcast,sizeof(int));
```

### 3.4. 单播、广播、组播（多播）

多播广播是用于建立分步式系统：例如网络游戏、ICQ 聊天构建、远程视频会议系统的重要工具。使用多播广播的程序和 UDP 的单播程序相似。区别在于多播广播程序使用特殊的 IP 地址。

对于单播而言，单播用于两个主机之间的端对端通信。

对于广播而言，广播用于一个主机对整个局域网上所有主机上的数据通信。广播只能用于客户机向服务器广播，因为客户机要指明广播的 IP 地址“192.168.0.255”和广播的端口号。服务器端 bind 的时候，绑定的端口号要跟广播的端口号是同一个。这样才能收到广播消息。实例请参考《udp\_广播》。

对于多播而言，也称为“组播”，将网络中同一业务类型主机进行了逻辑上的分组，进行数据收发的时候其数据仅仅在同一分组中进行，其他的主机没有加入此分组不能收发对应的数据。单播和广播是两个极端，要么对一个主机进行通信，要么对整个局域网上的主机进行通信。实际情况下，经常需要对一组特定的主机进行通信，而不是整个局域网上的所有主机，这就是多播的用途。例如，我们通常所说的讨论组。IPv4 多播地址采用 D 类 IP 地址确定多播的组。在 Internet 中，多播地址范围是从 224.0.0.0 到 234.255.255.255。

多播的程序设计也要使用 setsockopt()函数和 getsockopt()函数来实现。其中对于 setsockopt 的第二个参数 level 不再是 SOL\_SOCKET，而是 IPPROTO\_IP；而且第三个参数 optname 常见的选项有：

Optname	含义
IP_ADD_MEMBERSHIP	在指定接口上加入组播组
IP_DROP_MEMBERSHIP	退出组播组

选项 IP\_ADD\_MEMBERSHIP 和 IP\_DROP\_MEMBERSHIP 加入或者退出一个组播组，通过选项 IP\_ADD\_MEMBERSHIP 和 IP\_DROP\_MEMBERSHIP，对一个结构 struct ip\_mreq 类型的变量进行控制。

struct ip\_mreq 原型如下：

```
struct ip_mreq
{
    struct in_addr  imr_multiaddr; /*加入或者退出的多播组 IP 地址*/
    struct in_addr  imr_interface; /*加入或者退出的网络接口 IP 地址，本机 IP*/
};
```

选项 IP\_ADD\_MEMBERSHIP 用于加入某个多播组，之后就可以向这个多播组发送数据或者从多播组接收数据。此选项的值为 mreq 结构，成员 imr\_multiaddr 是需要加入的多播组 IP 地址，成员 imr\_interface 是本机需要加入多播组的网络接口 IP 地址。例如：



```
struct ip_mreq mreq;
memset(&mreq, 0, sizeof(struct ip_mreq));
mreq.imr_interface.s_addr = INADDR_ANY;
mreq.imr_multiaddr.s_addr = inet_addr("224.1.1.1");
if(-1 == setsockopt(sfd, IPPROTO_IP, IP_ADD_MEMBERSHIP, &mreq, sizeof(struct ip_mreq)))
{
    perror("setsockopt");
    exit(-1);
}
```

接下来再绑定组播的 port 号（如 65000），就可以接收组播消息了。实例请参考《udp\_多播》

选项 IP\_ADD\_MEMBERSHIP 每次只能加入一个网络接口的 IP 地址到多播组，但并不是一个多播组仅允许一个主机 IP 地址加入，可以多次调用 IP\_ADD\_MEMBERSHIP 选项来实现多个 IP 地址加入同一个广播组，或者同一个 IP 地址加入多个广播组。

选项 IP\_DROP\_MEMBERSHIP 用于从一个多播组中退出。例如：

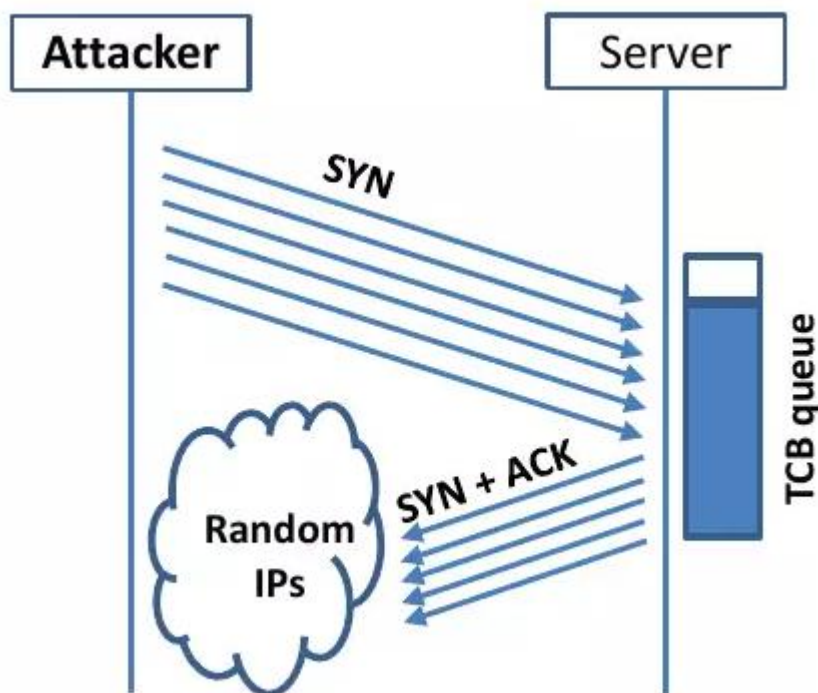
```
if(-1 == setsockopt(sfd, IPPROTO_IP, IP_DROP_MEMBERSHIP, &mreq, sizeof(struct ip_mreq)))
{
    perror("setsockopt");
    exit(-1);
}
```

### 3.5. 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 cookie

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。

## 3.6. 描述符属性修改及文件描述符的传递

### 1.fcntl 函数简介

fcntl 函数可以改变已打开的文件描述符性质

```
#include <sys/types.h>
```

```
#include <unistd.h>
```

```
#include <fcntl.h>
```

```

定义函数 int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, int arg);
int fcntl(int fd, int cmd, struct flock *lock);

```

fcntl()针对(文件)描述符提供控制,参数 fd 是被参数 cmd 操作(如下面的描述)的描述符。  
针对 cmd 的值,fcntl 能够接受第三个参数 int arg

参数 fd

参数 fd 代表欲设置的文件描述符。

参数 cmd

参数 cmd 代表打算操作的指令。

有以下几种情况:

F\_DUPFD 用来查找大于或等于参数 arg 的最小且仍未使用的文件描述词,并且复制参数 fd 的文件描述词。执行成功则返回新复制的文件描述词。新描述符与 fd 共享同一文件表项,但是新描述符有它自己的一套文件描述符标志,其中 FD\_CLOEXEC 文件描述符标志被清除。请参考 dup2()。

F\_GETFD 取得 close-on-exec 旗标。若此旗标的 FD\_CLOEXEC 位为 0, 代表在调用 exec()相关函数时文件将不会关闭。

F\_SETFD 设置 close-on-exec 旗标。该旗标以参数 arg 的 FD\_CLOEXEC 位决定。

F\_GETFL 取得文件描述词状态旗标, 此旗标为 open ( ) 的参数 flags。

F\_SETFL 设置文件描述词状态旗标, 参数 arg 为新旗标, 但只允许 O\_APPEND、O\_NONBLOCK 和 O\_ASYNC 位的改变, 其他位的改变将不受影响。

## 2.Socketpair 函数简介

int socketpair(int domain, int type, int protocol, int sv[2]); 前面 3 个参数参照 socket, domain 变为 AF\_LOCAL, sv[2],放我们 fd[2]

## 3.Sendmsg 函数简介

```

ssize_t sendmsg (int s, const struct msghdr *msg, int flags);

```

sendmsg 系统调用用于发送消息到另一个套接字

函数参数描述如下:

要在其上发送消息的套接口 s

信息头结构指针 msg, 这会控制函数调用的功能

可选的标记位参数 flags。这与 send 或是 sendto 函数调用的标记参数相同。

函数的返回值为实际发送的字节数。否则, 返回-1 表明发生了错误, 而 errno 表明错误原因。

## 理解 struct msghdr

当我第一次看到他时，他看上去似乎是一个需要创建的巨大的结构。但是不要怕。其结构定义如下：

```
struct msghdr {
    void      *msg_name;
    socklen_t  msg_namelen;
    struct iovec *msg_iov;
    size_t     msg_iovlen;
    void      *msg_control;
    size_t     msg_controllen;
    int        msg_flags;
};
```

结构成员可以分为四组。他们是：

套接口地址成员 `msg_name` 与 `msg_namelen`。

I/O 向量引用 `msg_iov` 与 `msg_iovlen`。

附属数据缓冲区成员 `msg_control` 与 `msg_controllen`。

接收信息标记位 `msg_flags`。

在我们将这个结构分为上面的几类以后，结构看起来就不那样巨大了。

成员 `msg_name` 与 `msg_namelen`

这些成员只有当我们的套接口是一个数据报套接口时才需要。`msg_name` 成员指向我们要发送或是接收信息的套接口地址。成员 `msg_namelen` 指明了这个套接口地址的长度。

当调用 `recvmsg` 时，`msg_name` 会指向一个将要接收的地址的接收区域。当调用 `sendmsg` 时，这会指向一个数据报将要发送到的目的地址。

注意，`msg_name` 定义为一个 `(void *)` 数据类型。我们并不需要将我们的套接口地址转换为 `(struct sockaddr *)`。

成员 `msg_iov` 与 `msg_iovlen`

这些成员指定了我们的 I/O 向量数组的位置以及他包含多少项。`msg_iov` 成员指向一个 `struct iovec` 数组。我们将会回忆起 I/O 向量指向我们的缓冲区。成员 `msg_iov` 指明了在我们的 I/O 向量数组中有多少元素。

成员 `msg_control` 与 `msg_controllen`

这些成员指向了我们附属数据缓冲区并且表明了缓冲区大小。`msg_control` 指向附属数据缓冲区，而 `msg_controllen` 指明了缓冲区大小。

这个通过 `man cmsg` 了解，或者通过下面的 `cmsg` 讲解熟悉

## 4.Recvmsg

```
int recvmsg(int s, struct msghdr *msg, unsigned int flags);
```

函数参数如下：

要在其上接收信息的套接口 `s`

信息头结构指针 `msg`，这会控制函数调用的操作。

可选标记位参数 `flags`。这与 `recv` 或是 `recvfrom` 函数调用的标记参数相同。

这个函数的返回值为实际接收的字节数。否则，返回-1 表明发生了错误，而 `errno` 表明错误原因。

## 5. Writev

```
ssize_t writev(int fd, const struct iovec *iov, int iovcnt)
```

一次写入多个 buf 内容

```
struct iovec {
    void *iov_base;    /* Starting address */
    size_t iov_len;    /* Number of bytes to transfer */
};
```

详细例子参考课堂代码

## 6. Readv

```
ssize_t readv(int fd, const struct iovec *iov, int iovcnt);
```

一次读取多个 buf 内容

## 7. Cmsg 用来设定我们的\*msg\_control 指针

```
size_t CMSG_LEN(size_t length);
```

返回结构体 `cmsg_hdr` 的大小，`length` 填入的是 `cmsg_data[]` 的大小，我们填入的是描述符 `fd`，所以是 `sizeof(int)`

```
unsigned char *CMSG_DATA(struct cmsg_hdr *cmsg);
```

```
struct cmsg_hdr {
    socklen_t cmsg_len;    /* data byte count, including header */
    int cmsg_level;    /* originating protocol */
    int cmsg_type;    /* protocol-specific type */
    /* followed by unsigned char cmsg_data[]; */
};
```

`CMSG_DATA` 返回 `cmsg_data` 的起始地址，也就是通过它放入我们的文件描述符

```
*(int*)CMSG_DATA(cmsg)=fd;
```

```
cmsg->cmsg_level = SOL_SOCKET;
```

```
cmsg->cmsg_type = SCM_RIGHTS;
```

# 4. EPOLL 多路复用

## 4.1. 背景介绍

`epoll` 是在 2.6 内核中提出的，是之前的 `select` 和 `poll` 的增强版本。相对于 `select` 和 `poll` 来说，`epoll` 更加灵活，没有描述符限制。`epoll` 使用一个文件描述符管理多个描述符，将用户关系的文件描述符的事件存放到内核的一个事件表中，这样在用户空间和内核空间的 `copy` 只需一次。同时 `epoll` 的性能更好，因此在工作中首选 `epoll`。

## 4.2. 接口使用

epoll 操作过程需要三个接口，分别如下：

```
#include <sys/epoll.h>
```

```
int epoll_create(int size);
```

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

```
int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout);
```

### (1) int epoll\_create(int size);

创建一个 epoll 的句柄，size 用来告诉内核这个监听的数目一共有多大。这个参数不同于 select() 中的第一个参数，给出最大监听的 fd+1 的值。需要注意的是，当创建好 epoll 句柄后，它就是会占用一个 fd 值，在 linux 下如果查看 /proc/进程 id/fd/，是能够看到这个 fd 的，所以在用完 epoll 后，必须调用 close() 关闭，否则可能导致 fd 被耗尽。

### (2) int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);

epoll 的事件注册函数，它不同与 select() 是在监听事件时告诉内核要监听什么类型的事件 epoll 的事件注册函数，它不同与 select() 是在监听事件时告诉内核要监听什么类型的事件，而是在这里先注册要监听的事件类型。第一个参数是 epoll\_create() 的返回值，第二个参数表示动作，用三个宏来表示：

EPOLL\_CTL\_ADD: 注册新的 fd 到 epfd 中；

EPOLL\_CTL\_MOD: 修改已经注册的 fd 的监听事件；

EPOLL\_CTL\_DEL: 从 epfd 中删除一个 fd；

第三个参数是需要监听的 fd，第四个参数是告诉内核需要监听什么事，struct epoll\_event 结构如下：

```
struct epoll_event {
    __uint32_t events; /* Epoll events */
    epoll_data_t data; /* User data variable */
};
```

events 可以是以下几个宏的集合：

EPOLLIN: 表示对应的文件描述符可以读（包括对端 SOCKET 正常关闭）；

EPOLLOUT: 表示对应的文件描述符可以写；

EPOLLPRI: 表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）；

EPOLLERR: 表示对应的文件描述符发生错误；

EPOLLHUP: 表示对应的文件描述符被挂断；

EPOLLET: 将 EPOLL 设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)来说的。

EPOLLONESHOT: 只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个 socket 的话，需要再次把这个 socket 加入到 EPOLL 队列里

epoll 对文件描述符的操作有两种模式：LT (level trigger) 和 ET (edge trigger)。LT 模式是默认模式，LT 模式与 ET 模式的区别如下：

LT 模式：当 epoll\_wait 检测到描述符事件发生并将此事件通知应用程序，应用程序可以不立即处理该事件。下次调用 epoll\_wait 时，会再次响应应用程序并通知此事件。

ET 模式：当 epoll\_wait 检测到描述符事件发生并将此事件通知应用程序，应用程序必须立即处理该事件。如果不处理，下次调用 epoll\_wait 时，不会再次响应应用程序并通知此事件。

ET 模式在很大程度上减少了 epoll 事件被重复触发的次数，因此效率要比 LT 模式高。epoll 工作在 ET 模式的时候，必须使用非阻塞套接口，以避免由于一个文件句柄的阻塞读/阻塞写操作把处理多个文件描述符的任务饿死。

### (3) int epoll\_wait(int epfd, struct epoll\_event \* events, int maxevents, int timeout);

等待事件的产生，类似于 `select()` 调用。参数 `events` 用来从内核得到事件的集合，`maxevents` 告之内核这个 `events` 有多大，这个 `maxevents` 的值不能大于创建 `epoll_create()` 时的 `size`，参数 `timeout` 是超时时间（毫秒，0 会立即返回，-1 将不确定，也有说法说是永久阻塞）。该函数返回需要处理的事件数目，如返回 0 表示已超时。

下面通过一个回射服务器来学习一下 `Epoll` 的使用，首先是服务器端代码：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <sys/epoll.h>
#include <unistd.h>
#include <sys/types.h>

#define IPADDRESS    "127.0.0.1"
#define PORT         8787
#define MAXSIZE      1024
#define LISTENQ      5
#define FDSIZE       1000
#define EPOLLEVENTS  100

//函数声明
//创建套接字并进行绑定
static int socket_bind(const char* ip,int port);
//IO 多路复用 epoll
static void do_epoll(int listenfd);
//事件处理函数
static void
handle_events(int epollfd,struct epoll_event *events,int num,int listenfd,char *buf);
//处理接收到的连接
static void handle_accpet(int epollfd,int listenfd);
//读处理
static void do_read(int epollfd,int fd,char *buf);
//写处理
static void do_write(int epollfd,int fd,char *buf);
//添加事件
static void add_event(int epollfd,int fd,int state);
//修改事件
static void modify_event(int epollfd,int fd,int state);
//删除事件
static void delete_event(int epollfd,int fd,int state);
```

```
int main(int argc, char *argv[])
{
    int listenfd;
    listenfd = socket_bind(IPADDRESS, PORT);
    listen(listenfd, LISTENQ);
    do_epoll(listenfd);
    return 0;
}

static int socket_bind(const char* ip, int port)
{
    int listenfd;
    struct sockaddr_in servaddr;
    listenfd = socket(AF_INET, SOCK_STREAM, 0);
    if (listenfd == -1)
    {
        perror("socket error:");
        exit(1);
    }
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    inet_pton(AF_INET, ip, &servaddr.sin_addr);
    servaddr.sin_port = htons(port);
    if (bind(listenfd, (struct sockaddr*)&servaddr, sizeof(servaddr)) == -1)
    {
        perror("bind error: ");
        exit(1);
    }
    return listenfd;
}

static void do_epoll(int listenfd)
{
    int epollfd;
    struct epoll_event events[EPOLLEVENTS];
    int ret;
    char buf[MAXSIZE];
    memset(buf, 0, MAXSIZE);
    //创建一个描述符
    epollfd = epoll_create(FDSIZE);
    //添加监听描述符事件
    add_event(epollfd, listenfd, EPOLLIN);
    for ( ; ; )
    {
```



```
//获取已经准备好的描述符事件
ret = epoll_wait(epollfd, events, EPOLLEVENTS, -1);
handle_events(epollfd, events, ret, listenfd, buf);
}
close(epollfd);
}

static void
handle_events(int epollfd, struct epoll_event *events, int num, int listenfd, char *buf)
{
    int i;
    int fd;
    //进行选好遍历
    for (i = 0; i < num; i++)
    {
        fd = events[i].data.fd;
        //根据描述符的类型和事件类型进行处理
        if ((fd == listenfd) && (events[i].events & EPOLLIN))
            handle_accpet(epollfd, listenfd);
        else if (events[i].events & EPOLLIN)
            do_read(epollfd, fd, buf);
        else if (events[i].events & EPOLLOUT)
            do_write(epollfd, fd, buf);
    }
}

static void handle_accpet(int epollfd, int listenfd)
{
    int clifd;
    struct sockaddr_in cliaddr;
    socklen_t cliaddrlen = sizeof(struct sockaddr);
    clifd = accept(listenfd, (struct sockaddr*)&cliaddr, &cliaddrlen);
    if (clifd == -1)
        perror("accpet error:");
    else
    {
        printf("accept a new
client: %s:%d\n", inet_ntoa(cliaddr.sin_addr), cliaddr.sin_port);
        //添加一个客户描述符和事件
        add_event(epollfd, clifd, EPOLLIN);
    }
}

static void do_read(int epollfd, int fd, char *buf)
{
    int nread;
```

```
nread = read(fd, buf, MAXSIZE);
if (nread == -1)
{
    perror("read error:");
    close(fd);
    delete_event(epollfd, fd, EPOLLIN);
}
else if (nread == 0)
{
    fprintf(stderr, "client close.\n");
    close(fd);
    delete_event(epollfd, fd, EPOLLIN);
}
else
{
    printf("read message is : %s", buf);
    //修改描述符对应的事件，由读改为写
    modify_event(epollfd, fd, EPOLLOUT);
}
}

static void do_write(int epollfd, int fd, char *buf)
{
    int nwrite;
    nwrite = write(fd, buf, strlen(buf));
    if (nwrite == -1)
    {
        perror("write error:");
        close(fd);
        delete_event(epollfd, fd, EPOLLOUT);
    }
    else
        modify_event(epollfd, fd, EPOLLIN);
    memset(buf, 0, MAXSIZE);
}

static void add_event(int epollfd, int fd, int state)
{
    struct epoll_event ev;
    ev.events = state;
    ev.data.fd = fd;
    epoll_ctl(epollfd, EPOLL_CTL_ADD, fd, &ev);
}

static void delete_event(int epollfd, int fd, int state)
```

```
{
    struct epoll_event ev;
    ev.events = state;
    ev.data.fd = fd;
    epoll_ctl(epollfd, EPOLL_CTL_DEL, fd, &ev);
}

static void modify_event(int epollfd, int fd, int state)
{
    struct epoll_event ev;
    ev.events = state;
    ev.data.fd = fd;
    epoll_ctl(epollfd, EPOLL_CTL_MOD, fd, &ev);
}
```

客户端代码如下:

```
#include <netinet/in.h>
#include <sys/socket.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/epoll.h>
#include <time.h>
#include <unistd.h>
#include <sys/types.h>
#include <arpa/inet.h>

#define MAXSIZE    1024
#define IPADDRESS  "127.0.0.1"
#define SERV_PORT  8787
#define FDSIZE      1024
#define EPOLLEVENTS 20

static void handle_connection(int sockfd);
static void
handle_events(int epollfd, struct epoll_event *events, int num, int sockfd, char *buf);
static void do_read(int epollfd, int fd, int sockfd, char *buf);
static void do_read(int epollfd, int fd, int sockfd, char *buf);
static void do_write(int epollfd, int fd, int sockfd, char *buf);
static void add_event(int epollfd, int fd, int state);
static void delete_event(int epollfd, int fd, int state);
static void modify_event(int epollfd, int fd, int state);

int main(int argc, char *argv[])
{
    int                sockfd;
```

```
    struct sockaddr_in servaddr;
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(SERV_PORT);
    inet_pton(AF_INET, IPADDRESS, &servaddr.sin_addr);
    connect(sockfd, (struct sockaddr*)&servaddr, sizeof(servaddr));
    //处理连接
    handle_connection(sockfd);
    close(sockfd);
    return 0;
}
```

```
static void handle_connection(int sockfd)
{
    int epollfd;
    struct epoll_event events[EPOLLEVENTS];
    char buf[MAXSIZE];
    int ret;
    epollfd = epoll_create(FDSIZE);
    add_event(epollfd, STDIN_FILENO, EPOLLIN);
    for ( ; ; )
    {
        ret = epoll_wait(epollfd, events, EPOLLEVENTS, -1);
        handle_events(epollfd, events, ret, sockfd, buf);
    }
    close(epollfd);
}
```

```
static void
handle_events(int epollfd, struct epoll_event *events, int num, int sockfd, char *buf)
{
    int fd;
    int i;
    for (i = 0; i < num; i++)
    {
        fd = events[i].data.fd;
        if (events[i].events & EPOLLIN)
            do_read(epollfd, fd, sockfd, buf);
        else if (events[i].events & EPOLLOUT)
            do_write(epollfd, fd, sockfd, buf);
    }
}
```

```
static void do_read(int epollfd, int fd, int sockfd, char *buf)
{
    int nread;
    nread = read(fd, buf, MAXSIZE);
    if (nread == -1)
    {
        perror("read error:");
        close(fd);
    }
    else if (nread == 0)
    {
        fprintf(stderr, "server close.\n");
        close(fd);
    }
    else
    {
        if (fd == STDIN_FILENO)
            add_event(epollfd, sockfd, EPOLLOUT);
        else
        {
            delete_event(epollfd, sockfd, EPOLLIN);
            add_event(epollfd, STDOUT_FILENO, EPOLLOUT);
        }
    }
}

static void do_write(int epollfd, int fd, int sockfd, char *buf)
{
    int nwrite;
    nwrite = write(fd, buf, strlen(buf));
    if (nwrite == -1)
    {
        perror("write error:");
        close(fd);
    }
    else
    {
        if (fd == STDOUT_FILENO)
            delete_event(epollfd, fd, EPOLLOUT);
        else
            modify_event(epollfd, fd, EPOLLIN);
    }
    memset(buf, 0, MAXSIZE);
}
```

```
static void add_event(int epollfd, int fd, int state)
{
    struct epoll_event ev;
    ev.events = state;
    ev.data.fd = fd;
    epoll_ctl(epollfd, EPOLL_CTL_ADD, fd, &ev);
}

static void delete_event(int epollfd, int fd, int state)
{
    struct epoll_event ev;
    ev.events = state;
    ev.data.fd = fd;
    epoll_ctl(epollfd, EPOLL_CTL_DEL, fd, &ev);
}

static void modify_event(int epollfd, int fd, int state)
{
    struct epoll_event ev;
    ev.events = state;
    ev.data.fd = fd;
    epoll_ctl(epollfd, EPOLL_CTL_MOD, fd, &ev);
}
```

## 4.3. Select 与 Epoll 的对比分析

### 4.3.1. select 原理概述

调用 `select` 时，会发生以下事情：

1. 从用户空间拷贝 `fd_set` 到内核空间；
2. 注册回调函数 `__pollwait`；
3. 遍历所有 `fd`，对全部指定设备做一次 `poll`（这里的 `poll` 是一个文件操作，它有两个参数，一个是文件 `fd` 本身，一个是当设备尚未就绪时调用的回调函数 `__pollwait`，这个函数把设备自己特有的等待队列传给内核，让内核把当前的进程挂载到其中）；
4. 当设备就绪时，设备就会唤醒在自己特有等待队列中的【所有】节点，于是当前进程就获取到了完成的信号。`poll` 文件操作返回的是一组标准的掩码，其中的各个位指示当前的不同的就绪状态（全 0 为没有任何事件触发），根据 `mask` 可对 `fd_set` 赋值；
5. 如果所有设备返回的掩码都没有显示任何的事件触发，就去掉回调函数的函数指针，进入有限时的睡眠状态，再恢复和不断做 `poll`，再作有限时的睡眠，直到其中一个设备有事件触发为止。
6. 只要有事件触发，系统调用返回，将 `fd_set` 从内核空间拷贝到用户空间，回到用户态，用户就可以对相关的 `fd` 作进一步的读或者写操作了。

### 4.3.2. epoll 原理概述

调用 `epoll_create` 时，做了以下事情：

1. 内核帮我们在 `epoll` 文件系统里建了个 `file` 结点；
2. 在内核 `cache` 里建了个红黑树用于存储以后 `epoll_ctl` 传来的 `socket`；
3. 建立一个 `list` 链表，用于存储准备就绪的事件。

调用 `epoll_ctl` 时，做了以下事情：

1. 把 `socket` 放到 `epoll` 文件系统里 `file` 对象对应的红黑树上；
2. 给内核中断处理程序注册一个回调函数，告诉内核，如果这个句柄的中断到了，就把它放到准备就绪 `list` 链表里。

调用 `epoll_wait` 时，做了以下事情：

观察 `list` 链表里有没有数据。有数据就返回，没有数据就 `sleep`，等到 `timeout` 时间到后即使链表没数据也返回。而且，通常情况下即使我们要监控百万计的句柄，大多一次也只返回很少量的准备就绪句柄而已，所以，`epoll_wait` 仅需要从内核态 `copy` 少量的句柄到用户态而已。

总结如下：

一颗红黑树，一张准备就绪句柄链表，少量的内核 `cache`，解决了大并发下的 `socket` 处理问题。

执行 `epoll_create` 时，创建了红黑树和就绪链表；

执行 `epoll_ctl` 时，如果增加 `socket` 句柄，则检查在红黑树中是否存在，存在立即返回，不存在则添加到树干上，然后向内核注册回调函数，用于当中断事件来临时向准备就绪链表中插入数据；

执行 `epoll_wait` 时立刻返回准备就绪链表里的数据即可。

### 4.3.3. 优缺点分析

`select` 缺点：

1. **最大并发数限制**：使用 32 个整数的 32 位，即  $32 \times 32 = 1024$  来标识 `fd`，虽然可修改，但是有以下第二点的瓶颈；
2. **效率低**：每次都会线性扫描整个 `fd_set`，集合越大速度越慢；
3. **内核/用户空间内存拷贝问题**。

`epoll` 的提升：

1. 本身没有最大并发连接的限制，仅受系统中进程能打开的最大文件数目限制；

2. **效率提升**: 只有活跃的 **socket** 才会主动的去调用 **callback** 函数;
3. **省去不必要的内存拷贝**: **epoll** 通过内核与用户空间 **mmap** 同一块内存实现。

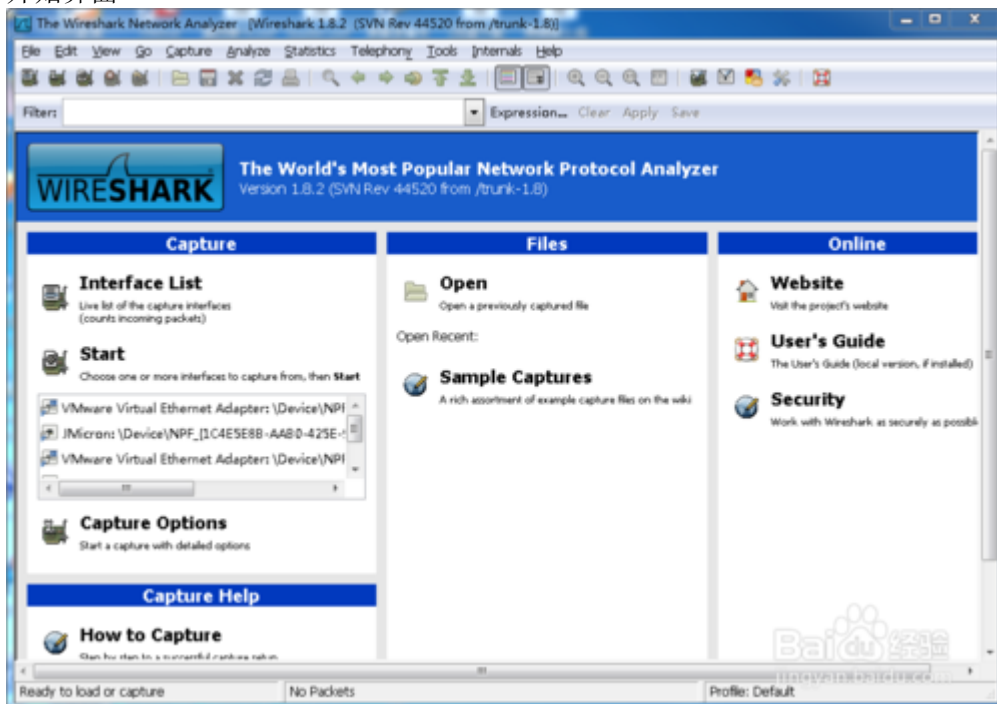
当然, 以上的优缺点仅仅是**特定场景下的情况**: 高并发, 且任一时间只有少数 **socket** 是活跃的。如果没有大量的 **idle -connection** 或者 **dead-connection**, **epoll** 的效率并不会比 **select/poll** 高很多, 但是当遇到大量的 **idle- connection**, 就会发现 **epoll** 的效率大大高于 **select/poll**

## 5. 五种编程模型

### 5.1. 同步异步, 阻塞非阻塞区别联系

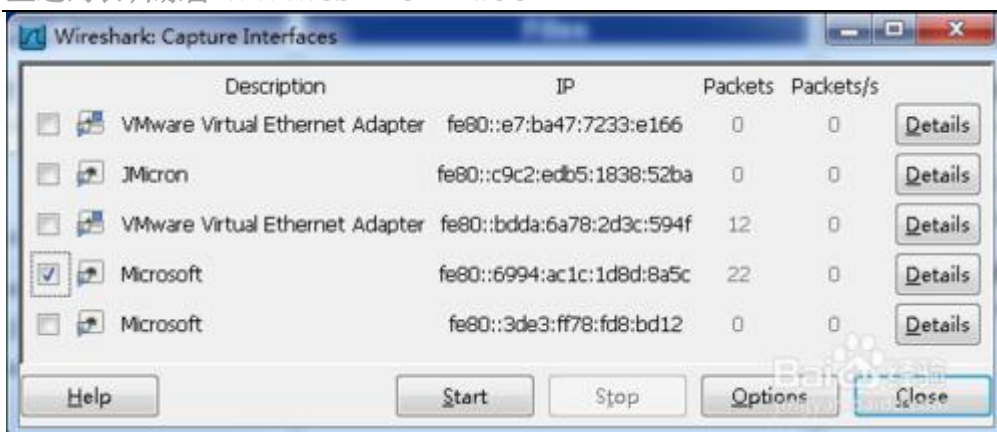
## 6. 抓包工具 wireshark 的使用

开始界面

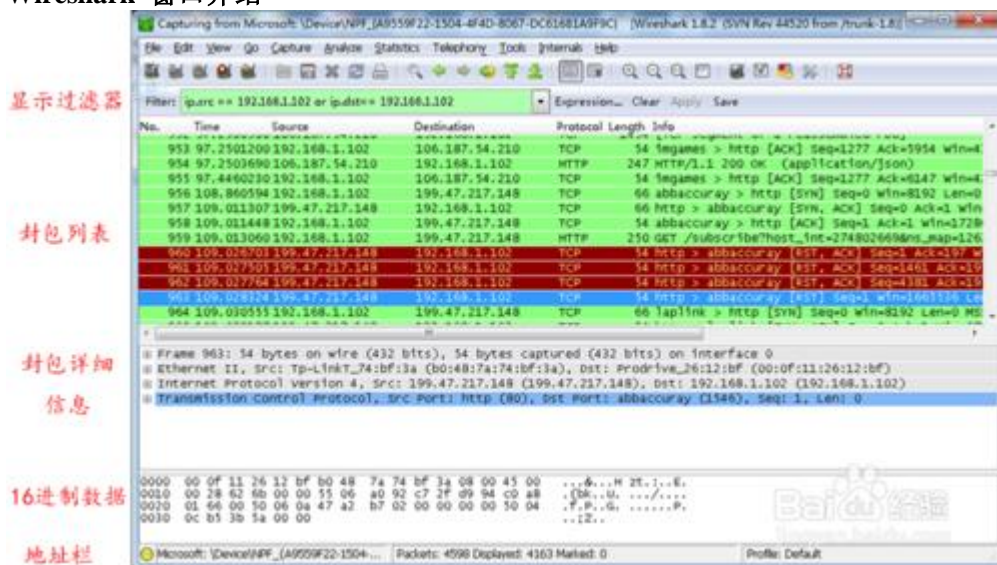


wireshark 是捕获机器上的某一块网卡的网络包, 当你的机器上有多块网卡的时候, 你需要选择一个网卡。点击 **Caputre->Interfaces..** 出现下面对话框, 选择正确的网卡。然后点击**"Start"**按钮, 开始抓包



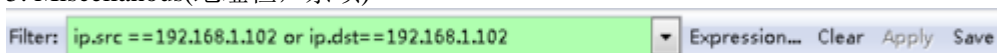


Wireshark 窗口介绍



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. Miscellaneous(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。

Filter: `ip.src == 192.168.1.102 or ip.dst == 192.168.1.102` Expression... Clear Apply Save Filter 102

## 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	74.125.128.136	TCP	60	[TCP Dup ACK 237#8] 8377 > http [ACK] Seq=372
266	15.8921180	74.125.128.136	192.168.1.102	TCP	1484	[TCP Retransmission] [TCP segment of a reasse
267	15.8921790	192.168.1.102	74.125.128.136	TCP	60	[TCP Dup ACK 237#5] 8377 > http [ACK] Seq=372
268	15.8926100	74.125.128.136	192.168.1.102	TCP	610	[TCP Retransmission] [TCP segment of a reasse
269	15.8926540	192.168.1.102	74.125.128.136	TCP	60	[TCP Dup ACK 237#9] 8377 > http [ACK] Seq=372
270	16.5576320	114.80.142.90	192.168.1.102	HTTP	264	HTTP/1.0 304 Not Modified
271	16.5690360	192.168.1.102	180.168.255.118	DNS	76	Standard query 0x30ee A www.blogjava.net
272	16.5695810	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	54	8361 > http [ACK] Seq=2094 Ack=421 Win=16860
275	16.8842490	114.80.142.90	192.168.1.102	HTTP	264	[TCP Retransmission] HTTP/1.0 304 Not Modified
276	16.8843460	192.168.1.102	114.80.142.90	TCP	60	[TCP Dup ACK 274#1] 8361 > http [ACK] Seq=2094
277	17.0635280	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.hjenglish.com
279	17.0661740	180.168.255.118	192.168.1.102	DNS	169	Standard query response 0xb4c CNAME www.huji
280	17.0681610	192.168.1.102	180.168.255.118	DNS	74	Standard query 0xa99 A www.hjenglish.com
281	17.0690520	180.168.255.118	192.168.1.102	DNS	92	Standard query response 0xb4c A 61.155.169.1
282	17.0753540	192.168.1.102	180.168.255.118	DNS	71	Standard query 0x0a15 A hling34.net
283	17.1430580	180.168.255.118	192.168.1.102	DNS	175	Standard query response 0x7272 CNAME www.hjer
284	17.1455140	192.168.1.102	180.168.255.118	DNS	75	Standard query 0x5491 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

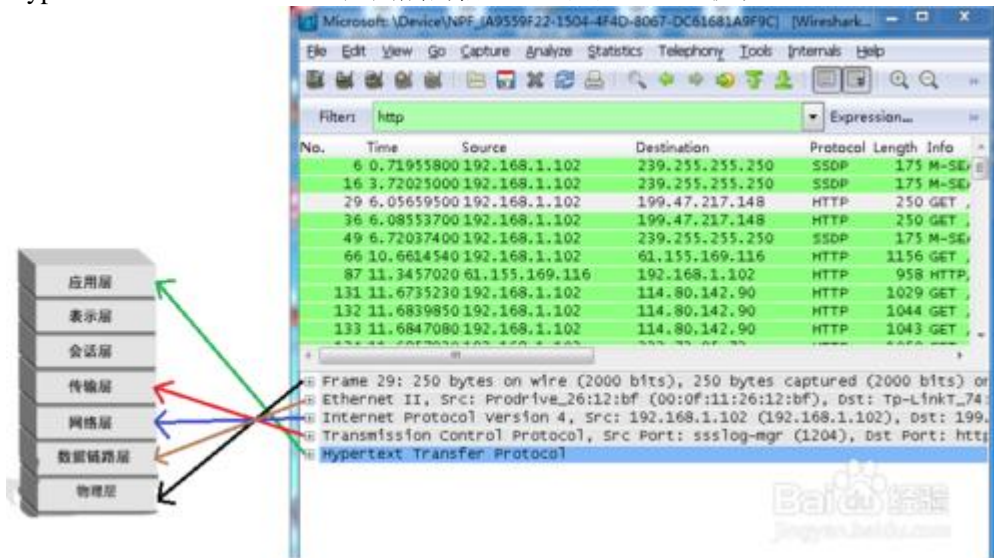
Frame: 物理层的数据帧概况

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

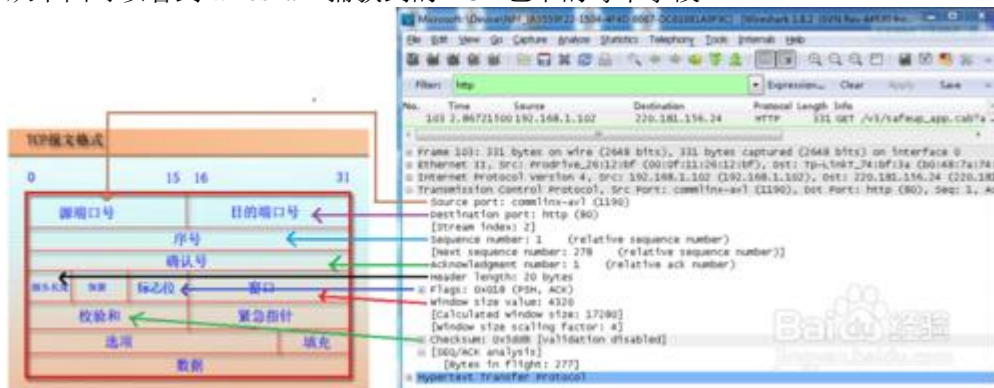
Transmission Control Protocol: 传输层 T 的数据段头部信息，此处是 TCP

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



### TCP 包的具体内容

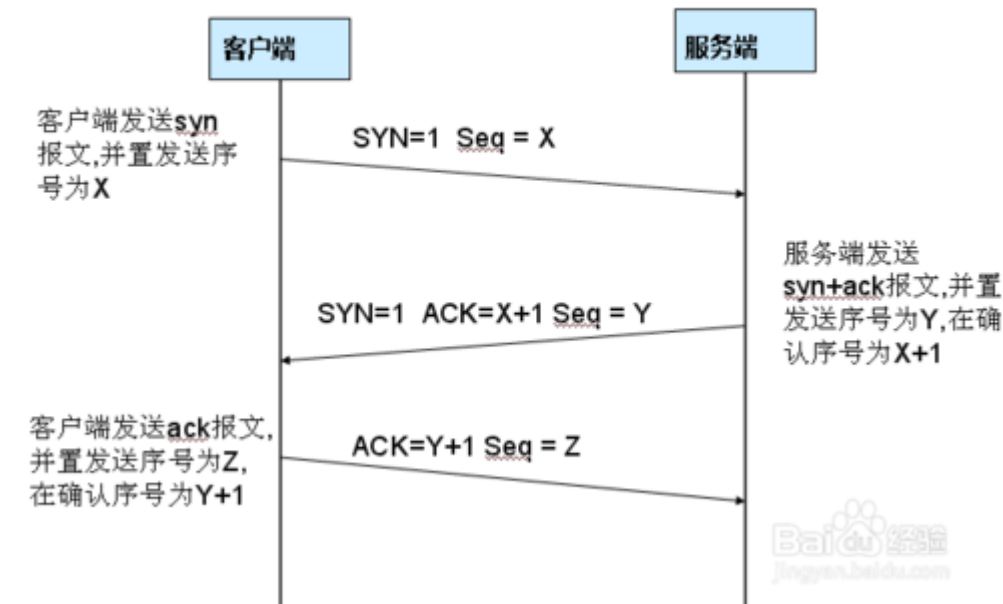
从下图可以看到 wireshark 捕获到的 TCP 包中的每个字段。



看到这，基本上对 wireshark 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

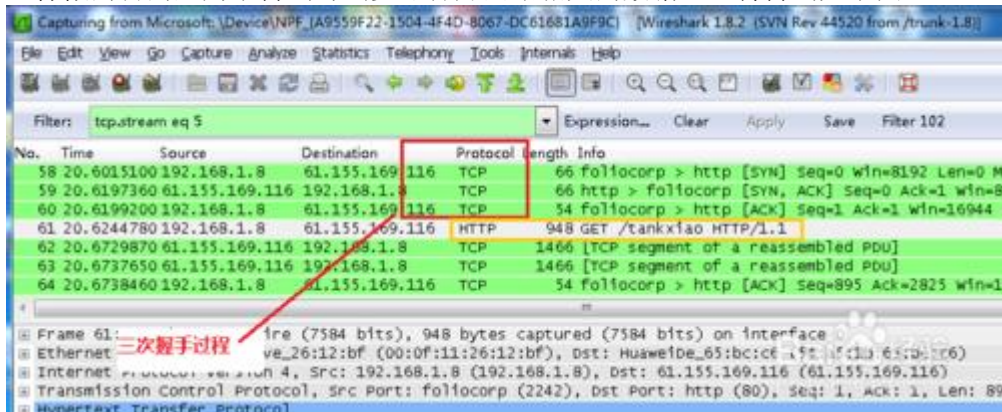


这图我都看过很多遍了，这次我们用 wireshark 实际分析下三次握手的过程。

打开 wireshark, 打开浏览器输入 `http://www.crl73.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",

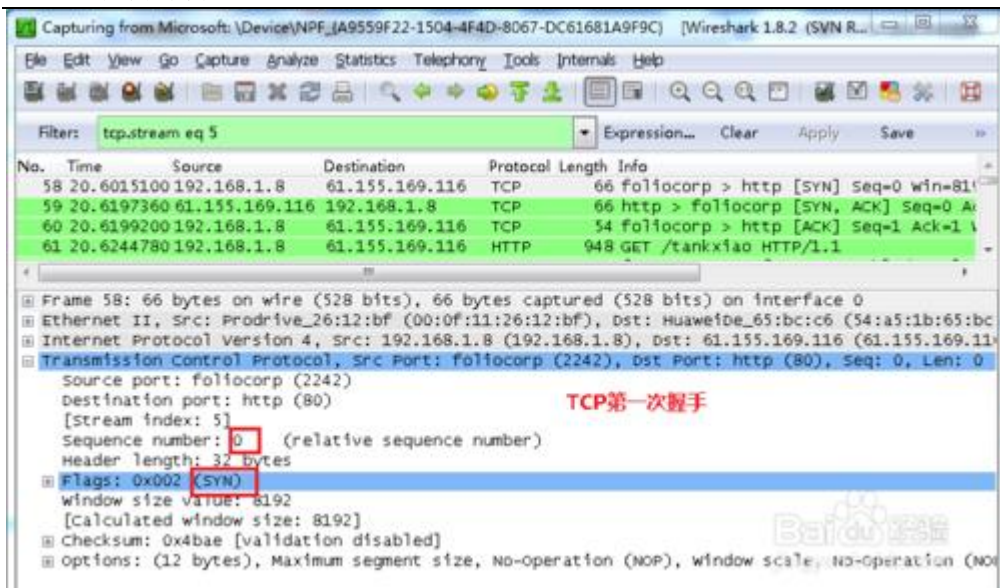
这样做的目的是为了得到与浏览器打开网站相关的数据包，将得到如下图



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。

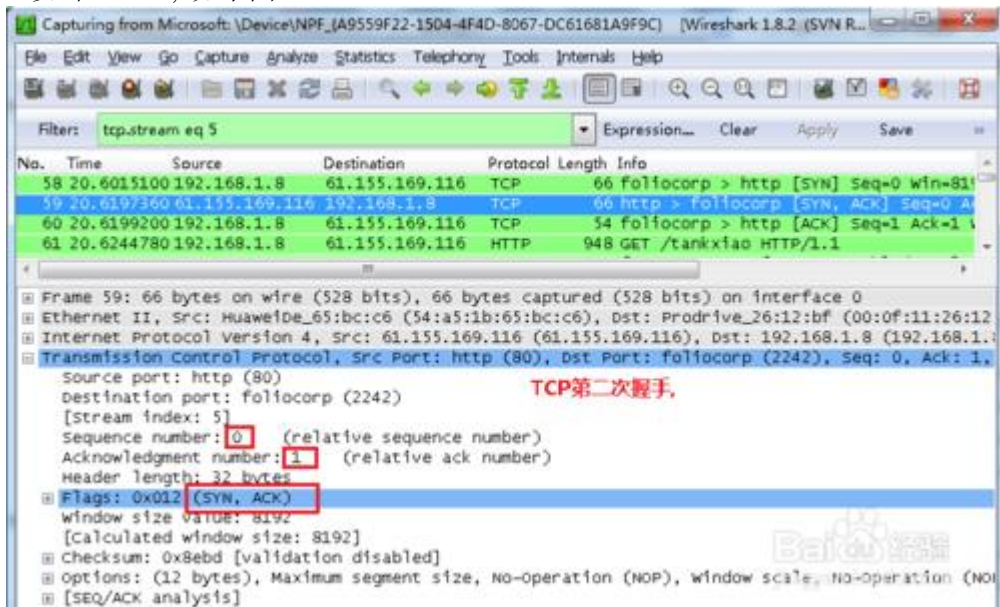
## 第一次握手数据包

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



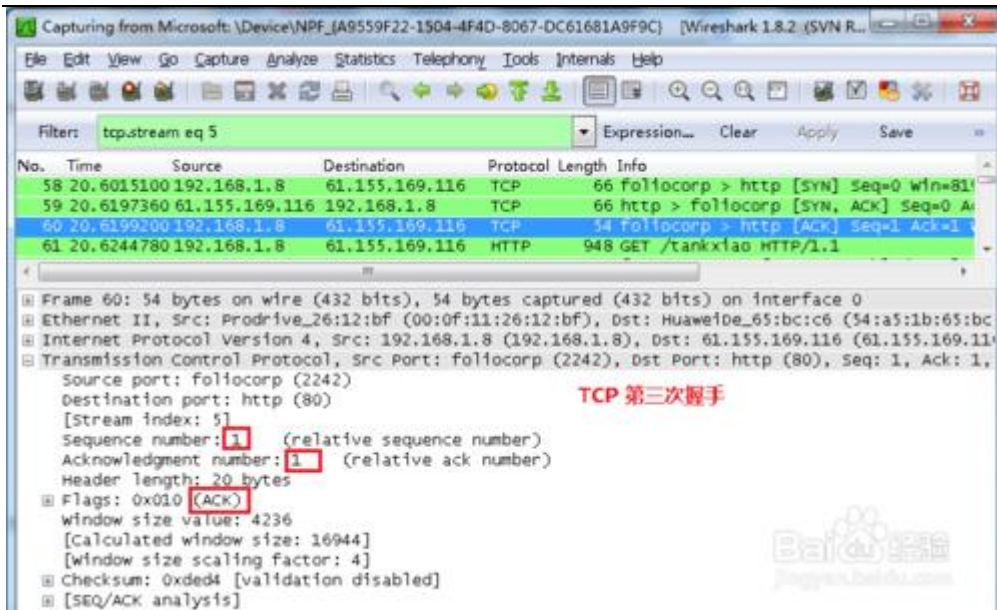
### 第二次握手的数据包

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 I S N 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