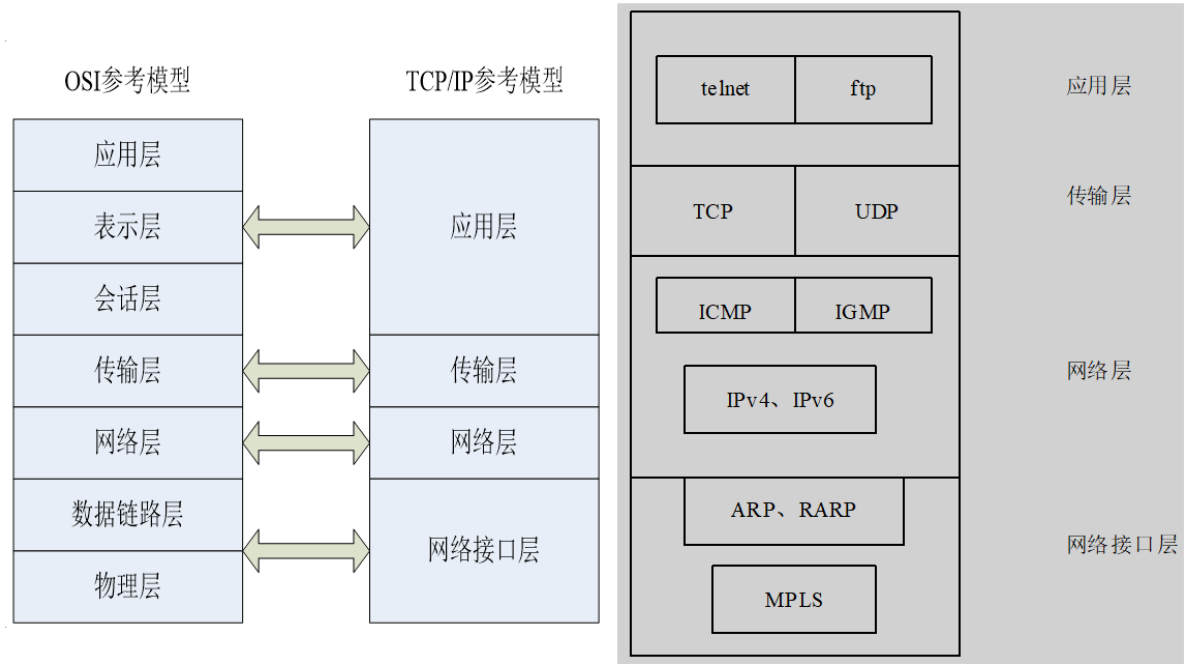
Linux网络编程

1.网络编程协议概述

1.1协议概述

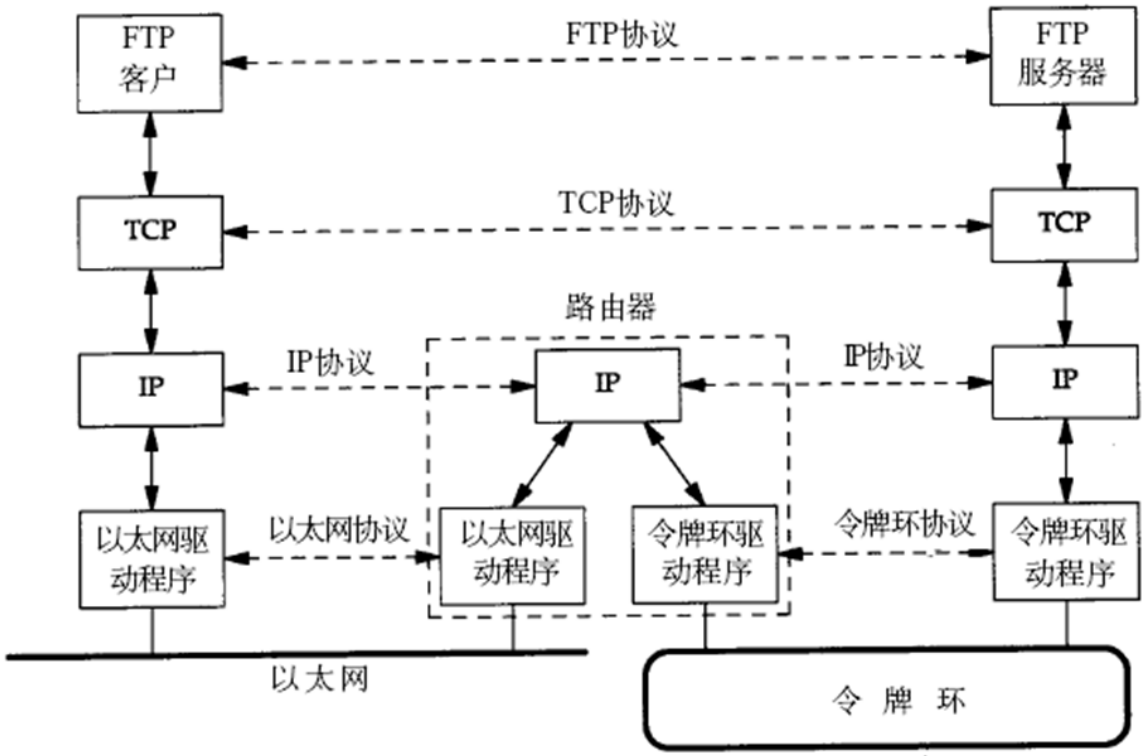
1. 协议protocol：通信双方必须遵守的规则
2. 协议按层次划分可以分为**osi7层模型**和**tcp/ip四层模型**
    1. osi 参考模型： 7 层（应-表-会-传-网-数-物） 是由 ISO（国际标准化组织） 定义
        - 应用层 表示层 会话层 传输层 网络层 数据链路层 物理层
    2. tcp/ip 模型 4 层: ARPA 在研究 APRAnet 时提出的 （ARPA： 美国高级研究计划署）
        - 应用层 传输层 网络层 网络接口层
            - 应用层协议：http 超文本传输协议、ftp 文件传输协议、telnet 远程登录、ssh 安全外壳协议、 smtp简单邮件发送、pop3 收邮件等
            - 传输层协议：tcp 传输控制协议、udp 用户数据包协议等
                - 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
                - UDP 协议： 用户数据包协议、非面向连接、速度快、不可靠
            - 网络层协议：ip 网际互联协议、 icmp 网络控制消息协议、 igmp 网络组管理协议等
            - 网络接口层协议：arp 地址转换协议、rarp 反向地址转换协议、mpls 多协议标签交换等
3. 协议按应用划分可以分为**公有协议**和**私有协议。**
4. 通常是ip地址后面跟端口号(范围0-65535)：ip用来定位主机，port区别应用（进程）； 用户自己定义的通常要大于1024（小于1024通常分配给常用的进程和协议），小于1万

1.2OSI参考模型及TCP/IP参考模型



TCP/IP协议族的每一层的作用：

1. 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。
2. 网络层：负责点(路由节点)到点的传输，将数据帧封装成IP数据报，并运行必要的路由算法。
3. 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。
4. 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，Inter-net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。

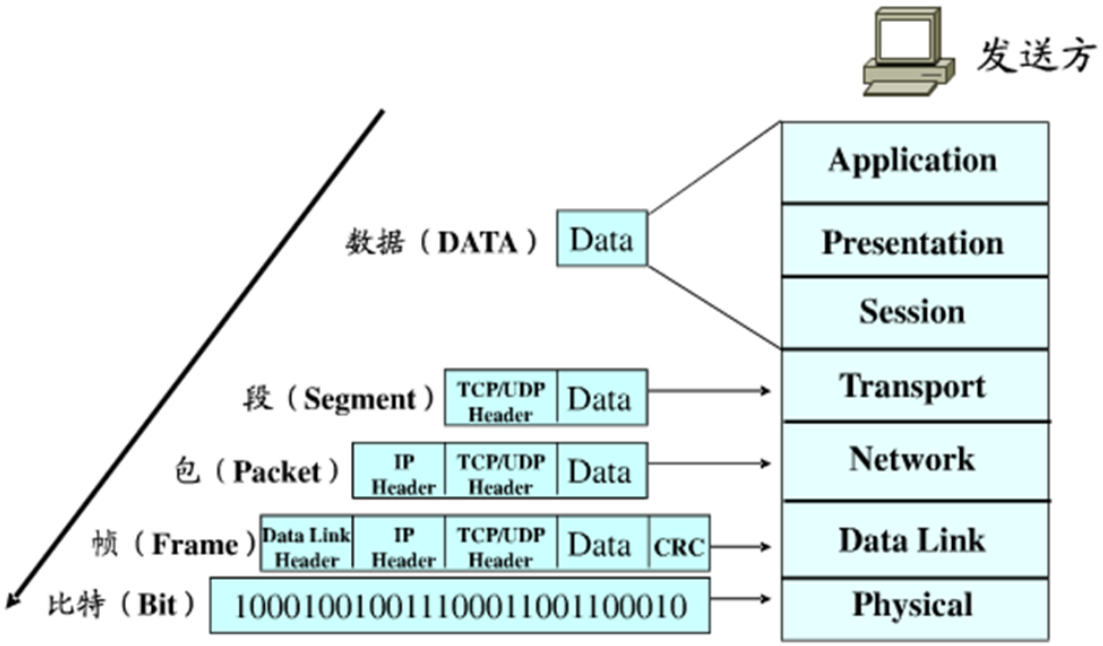
网络层负责点到点（ptop，point-to-point）的传输（这里的“点”指主机或路由器），而传输层负责端到端（etoe，end-to-end）的传输（这里的“端”指源主机和目的主机）。传输层可选择TCP或UDP协议。

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。

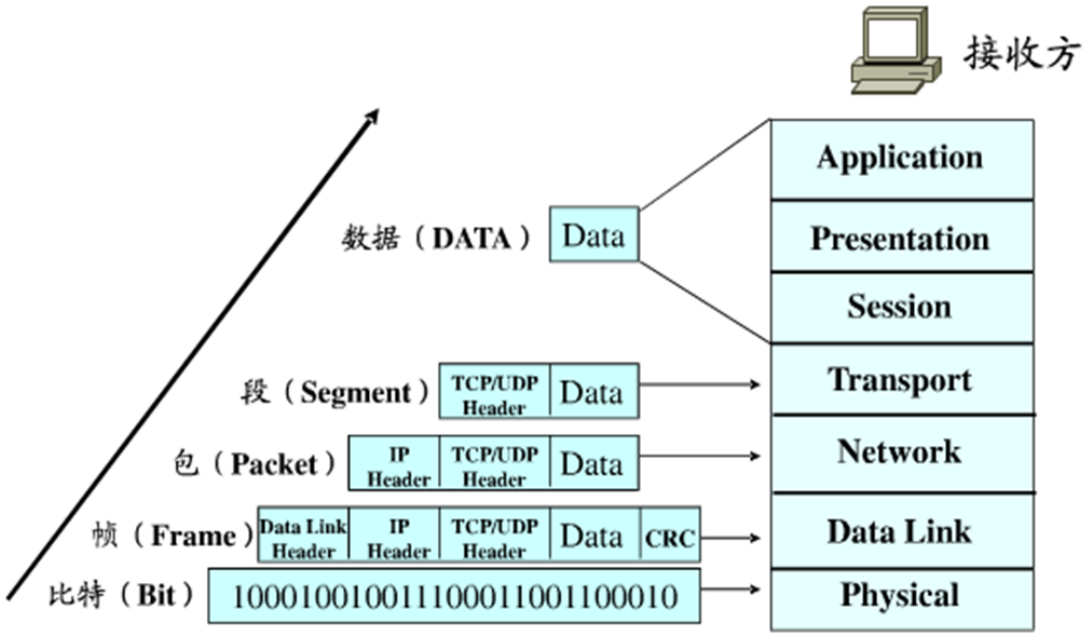
虽然IP、ARP和RARP数据报都需要以太网驱动程序来封装成帧，但是从功能上划分，ARP和RARP属于链路层，IP属于网络层。虽然ICMP、IGMP、TCP、UDP的数据都需要IP协议来封装成数据报，但是从功能上划分，ICMP、IGMP与IP同属于网络层，TCP和UDP属于传输层。

TCP/IP协议族的每一层协议的相关注解：

1. 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。
2. 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、
3. IP：（网际互联协议）负责在主机和网络之间寻址和传递数据包。
4. 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。
5. 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。
6. 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。
7. 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。



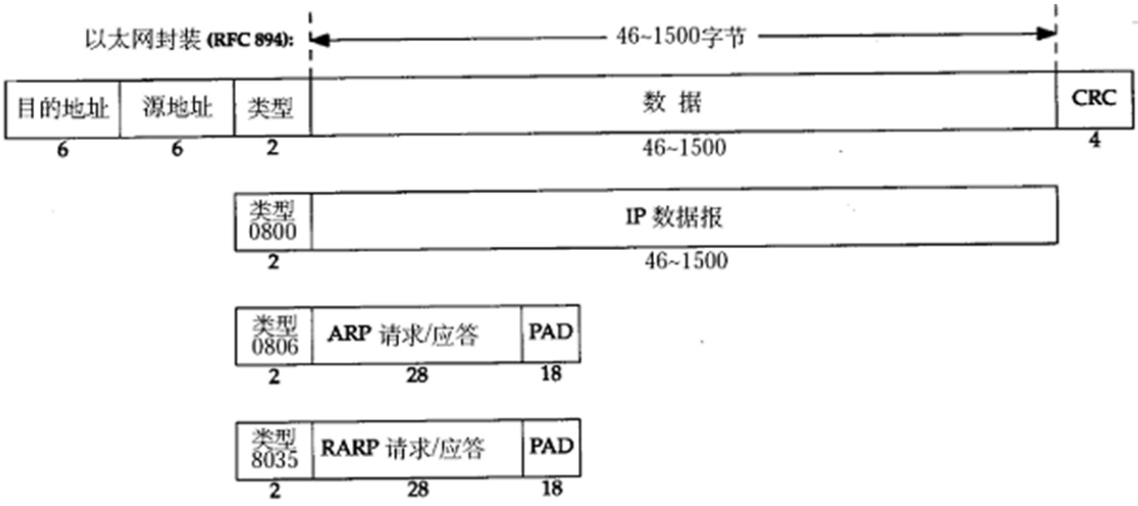
- 数据封装过程是在不同的层次对数据打上相应的标识



- 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以太网帧数据格式

以太网的帧格式如下所示：



以太网帧格式

1. 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，“ether 00:0c:29:b4:c6:ef”部分就是硬件地址。
2. **协议字段**有三类值，分别对应IP、ARP、RARP
3. 帧尾是CRC校验码
4. 以太网帧中的数据长度规定**最小46字节**，**最大1500字节**，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。

1.4ARP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

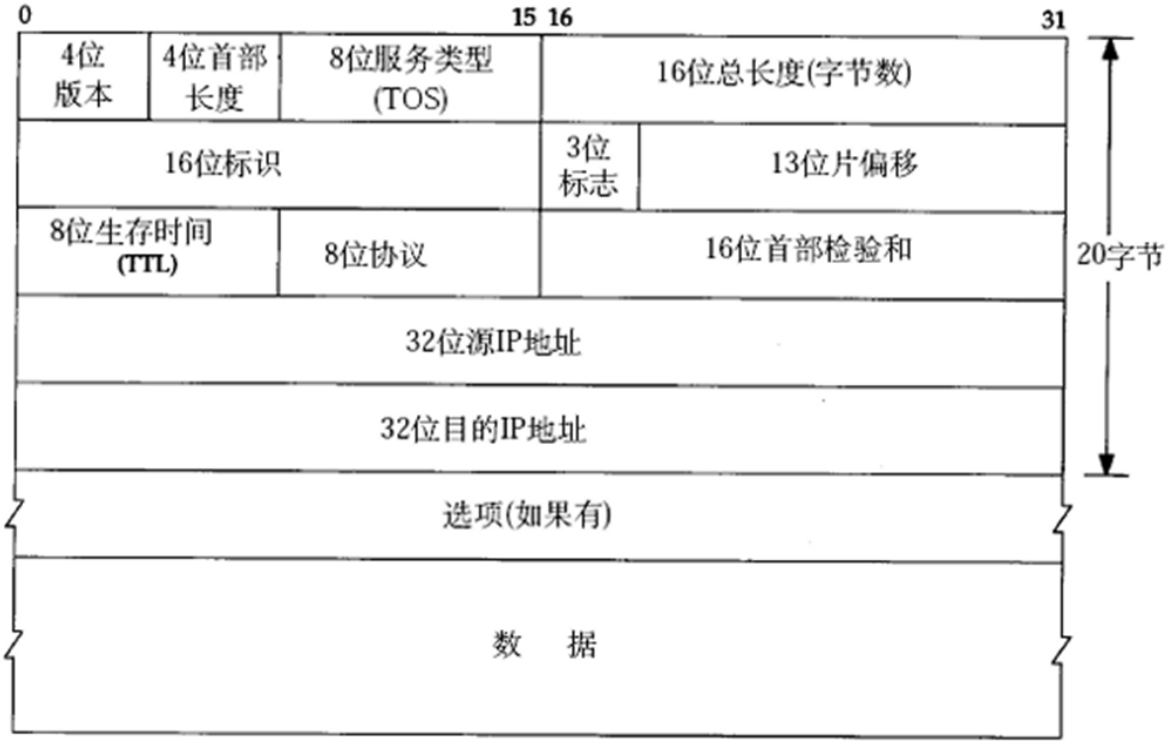
网帧首部的硬件地址填FF:FF:FF:FF:FF:FF表示广播），目的主机接收到广播的ARP请求，发现其中的IP地址与本机相符，则发送一个ARP应答数据包给源主机，将自己的硬件地址填写在应答包中。

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：

1. **硬件类型**指链路层网络类型，1为以太网
2. **协议类型**指要转换的地址类型，0x0800为ipv4地址
3. **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）
4. **op字段**为1表示ARP请求，op字段为2表示ARP应答
5. 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。

1.5IP数据报格式



IP 数据报格式

1. **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）
2. **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
3. **服务类型（Type of Service）**：用于规定本数据报的处理方式

4. **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
5. **标识符（Identifier）**:长度16位。该字段和**Flags**和**Fragment Offest**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。**
6. **标记（Flags）**：占 3 位。**第一位未使用**，其值为 0。第二位称为 DF（不分片），表示是否允许分片。取值为 0 时，表示允许分片；取值为 1 时，表示不允许分片。第三位称为 MF（更多分片），表示是否还有分片正在传输；设置为 0 时，表示没有更多分片需要发送，为1时，表示后面还有分片要发送
7. **片偏移（Fragment Offset）**：长度13位。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
8. **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。TTL的最大值是255，TTL的一个推荐值是64。
9. **协议（Protocol）**：长度8比特。标识了上层所使用的协议(数据发送的协议类型)；

   比较常用的协议号：

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

10. **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。
11. **源IP地址和目的IP地址**：这两个地段都是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。
12. **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。可选项目包含以下内容：
    1. 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。
    2. 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。
    3. 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。
    4. 时间戳（Timestamps）：当IP包离开每个路由器的时候记录时间。

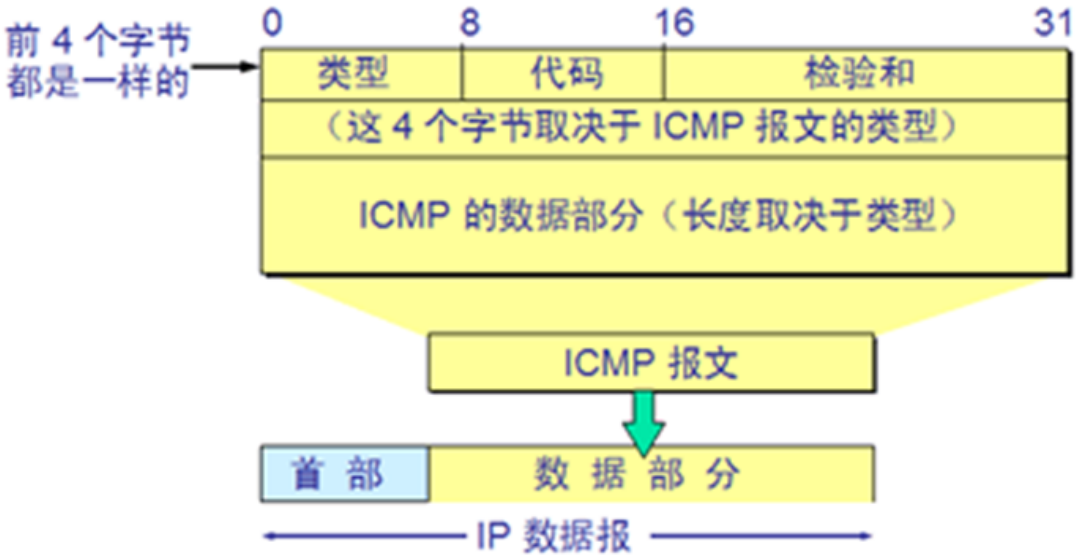
1.6ICMP协议格式

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。

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。



1. 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；
2. 类型表示 ICMP 的消息类型，代码表示对类型的进一步说明，校验和表示对整个报文的报文信息的校验。后4个字节根据ICMP包的类型而取不同的值
3. 在 ICMP 报文中，如果类型和代码不同，ICMP 数据包报告的消息含义也会不同。常见的类型和代码的 ICMP 含义如表所示

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		

TYPE	CODE	Description	Query	Error
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		

1.7路由(了解)

路由是指路由器从一个接口上收到数据包，根据数据包的目的地址进行定向并转发到另一个接口的过程。路由工作在OSI参考模型第三层——网络层

1.7.1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。

1.7.2路由表

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。

1.7.3hub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：

1. 第一步是将信息上传到上联设备；
2. 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。

1.7.4以太网交换机工作原理

交换机分为两种：广域网交换机和局域网交换机。广域网交换机主要应用于电信领域，提供通信基础平台。而局域网交换机则应用于局域网络，用于连接终端设备，如PC机及网络打印机等。

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。

1.8TCP协议

1.8.1概述

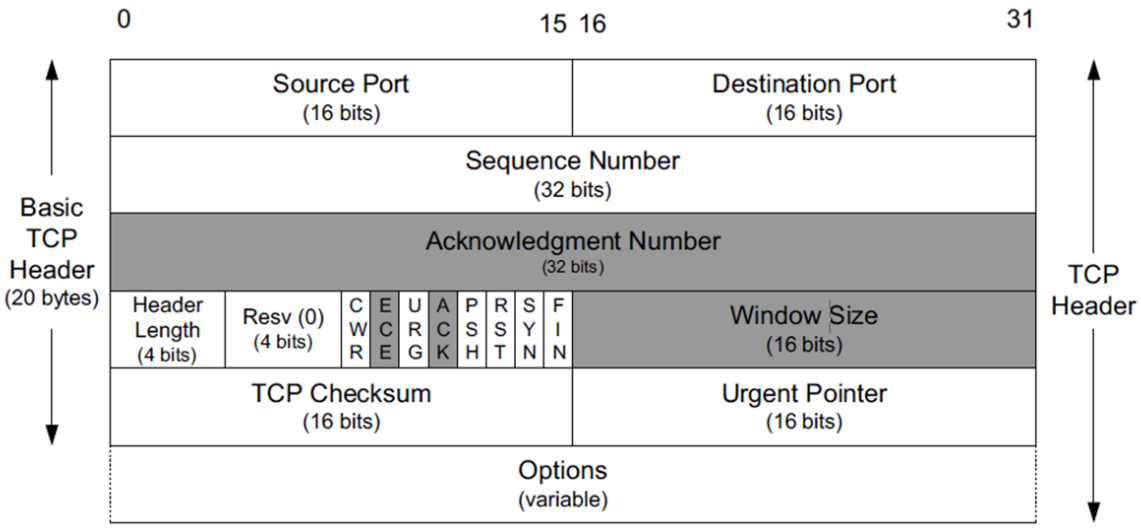
TCP是TCP/IP体系中面向连接的运输层协议，它提供全双工和可靠交付的服务。它采用许多机制来确保可靠数据传输，如采用序列号、确认重传、滑动窗口等。

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"是指发送发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。

1.8.2TCP报文头部格式



1. **源端口、目的端口**：16位长。标识出远端和本地的端口号
2. **序列号**：32位长。标识发送的数据报的顺序
3. **确认号**：32位长。希望收到的下一个数据报的序列号
4. **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
5. **Resv**：占4位。为 TCP 将来的发展预留空间，目前必须全部为 0
6. **CWR**：拥塞窗口减（发送方降低它的发送速率）
7. **ECE**：ECN回显（发送方收到了一个更早的拥塞报告）
8. **URG**：表示本报文段中发送的数据是否包含紧急数据。URG=1 时表示有紧急数据。当 URG=1 时，后面的紧急指针字段才有效
9. **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。TCP 规定，连接建立后，ACK 必须为 1
10. **PSH**：告诉对方收到该报文段后是否立即把数据推送给上层。如果值为 1，表示应当立即把数据提交给上层，而不是缓存起来
11. **RST**：表示是否重置连接。如果 RST=1，说明 TCP 连接出现了严重错误（如主机崩溃），必须释放连接，然后再重新建立连接

12. **SYN**：用于建立连接。当SYN=1时，表示发起一个连接请求
13. **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
14. **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。
15. **校验和字段**：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
16. **紧急指针字段**：紧急指针（Urgent Pointer）：仅当前面的 URG 控制位为 1 时才有意义。它指出本数据段中为紧急数据的字节数，占 16 位。当所有紧急数据处理完后，TCP 就会告诉应用程序恢复到正常操作。即使当前窗口大小为 0，也是可以发送紧急数据的，因为紧急数据无须缓存
17. **可选项**：长度不定，但长度必须是 32位的整数倍。包括最大TCP载荷，窗口比例、选择重复数据报等选项
18. **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**

1.8.3TCP的三次握手

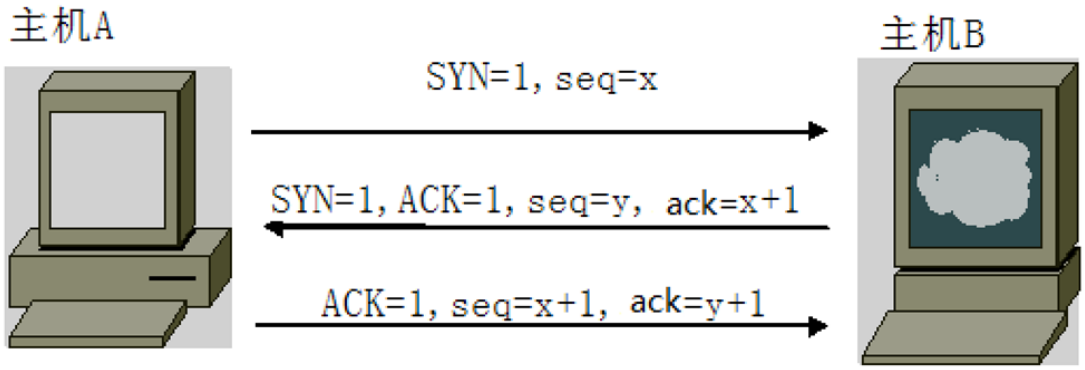
在利用TCP实现源主机和目的主机通信时，目的主机必须同意，否则TCP连接无法建立。为了确保TCP连接的成功建立，TCP采用了一种称为三次握手的方式，三次握手方式使得"序号/确认号"系统能够正常工作，从而使它们的序号达成同步。如果三次握手成功，则连接建立成功，可以开始传送数据信息。

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包的序号为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，同时自己也发送一个序号Y。
3. 源主机A的TCP收到目标主机B的确认后，要给目标主机B发出确认。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。

1.8.4TCP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:

表示初始状态。

LISTEN:

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：

表示连接已经建立了。

FIN_WAIT_1:

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。

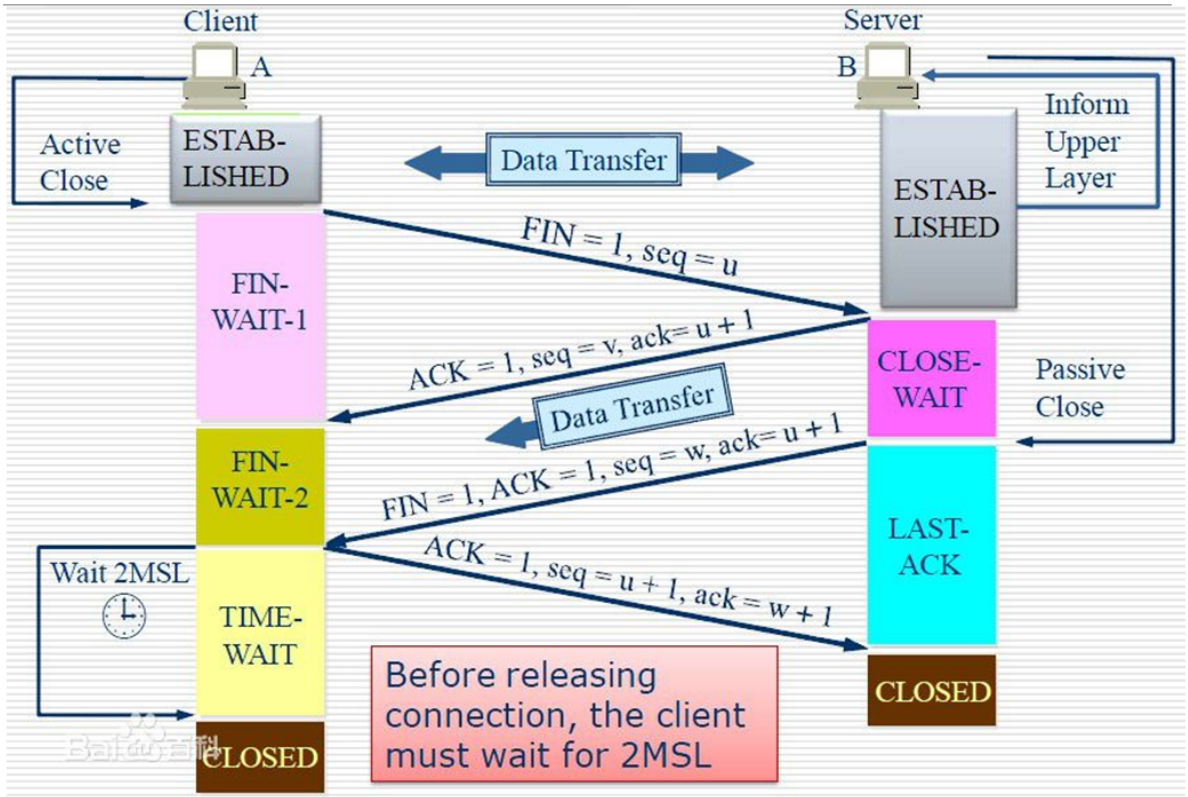
CLOSE_WAIT:

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，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。

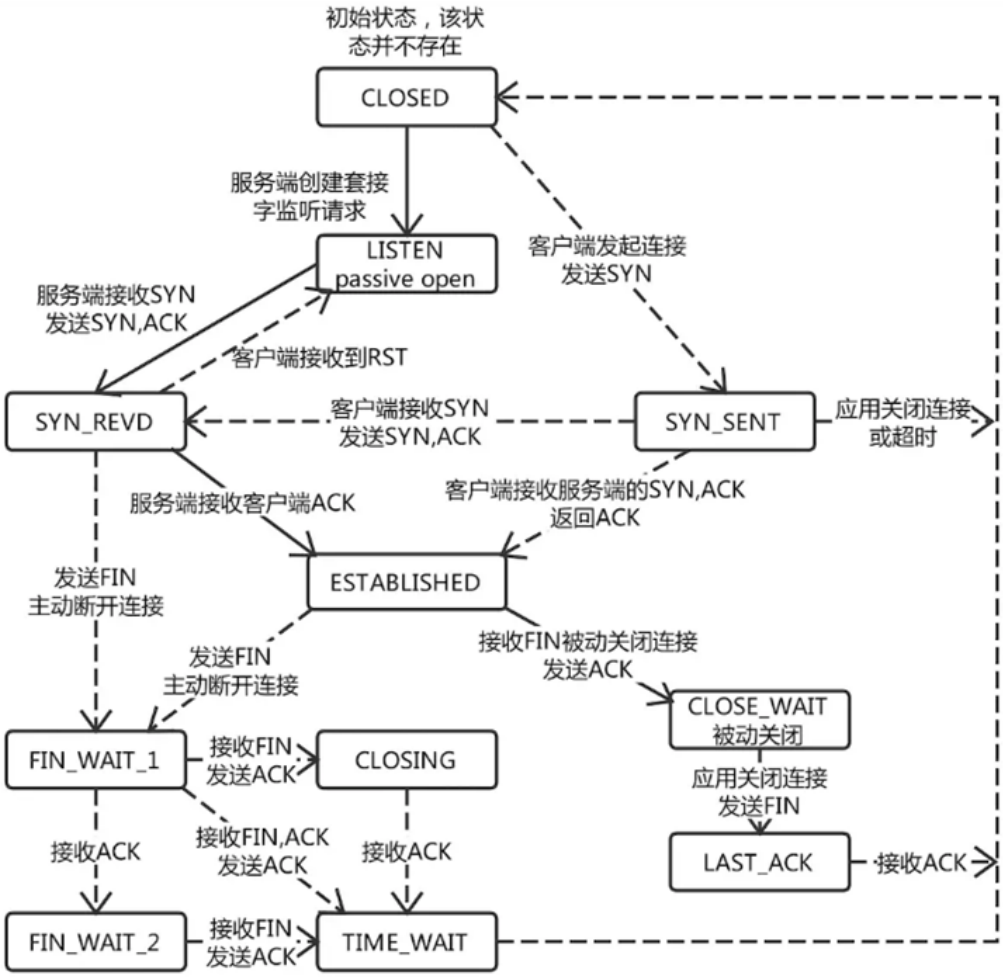
LAST_ACK:

这个状态还是比较容易好理解的，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。

tcp四次挥手示意图：



tcp状态迁移图



1.8.5MTU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、**MTU**：maximum transmission unit，**最大传输单元，由硬件规定**，如以太网的MTU为1500字节。

MSS：maximum segment size，最大分段大小，为TCP数据包每次传输的最大数据分段大小，一般由发送端向对端TCP通知对端在每个分节中能发送的最大TCP数据。MSS值为MTU值减去IPv4 Header（20 Byte）和TCP header（20 Byte）得到。

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，自行阅读 https://www.cnblogs.com/wuchanming/p/4422779.html

1.9UDP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器方。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

UDP报文格式

1. 源、目标端口号字段：占16位。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
2. 长度：占16位。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
3. 校验和：占16位。可以检验数据在传输过程中是否被损坏。

1.10协议选择

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。

1.11C/S与B/S模式

1. **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。
2. **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。
3. **两种模式的优缺点**
    1. **对于C/S模式：**
        1. **优点**：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器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、LOL等，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。

    2. **缺点**：由于客户端和服务器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。
2. **对于B/S模式：**
    1. **优点**：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。
    2. **缺点**：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。使用的观感大打折扣。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**。
4. **总结**：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，影视播放多用C/S模式；电商，政企事业网站多用B/S模式。

2.网络相关概念

2.1基础概念

1. 套接口概念：

    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，其中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1023的整型值

3. IP地址表示：

    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。

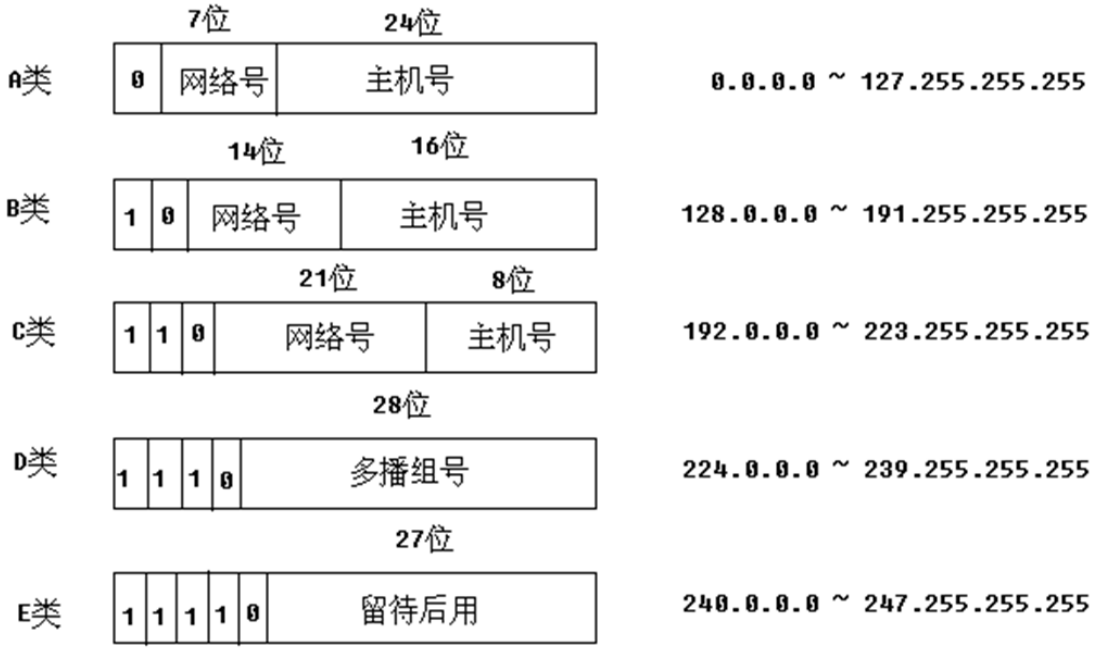
    例如对于：10000000 00001011 00000011 00011111

    点分十进制表示为：128.11.3.31

4. IP地址的分类
    1. A类: 0.0.0.0-127.255.255，其中段0和127不可用，可用地址范围1.0.0.0.-127.255.255.255 **一般用于大型网络**
    2. B类: 128.0.0.0-191.255.255.255，其中可用地址范围128.0.0.0-191.255.255.255 **一般用于中等规模网络**
    3. C类: 192.0.0.0-223.255.255.255其中可用地址范围192.0.0.0-223.255.255.255 **一般用于小型网络**

4. D类: 224.0.0.0-239.255.255.255其中可用地址范围224.0.0.0-239.255.255.255，**用作广播地址**
5. E类: 240.0.0.0-255.255.255.255，其中段255不可用，**用作保留使用**

5. 其中除了段0和段127之外，还有一些IP地址因为有其他的用途，是不可以用作普通IP的。还有一部分被用作私有IP地址。
    1. **公有地址**（Public address）：由Inter NIC（Internet Network Information Center 因特网信息中心）负责。这些IP地址分配给注册并向Inter NIC提出申请的组织机构。通过它直接访问因特网。
    2. **私有IP**的出现是为了解决公有IP地址不够用的情况。从A、B、C三类IP地址中拿出一部分作为私有IP地址，这些IP地址不能被路由到Internet骨干网上，Internet路由器也将丢弃该私有地址。如果私有IP地址想要连至Internet，需要将私有地址转换为公有地址。这个转换过程称为网络地址转换（Network Address Translation，NAT），通常使用路由器来执行NAT转换。范围如下
        1. A类：10.0.0.0~10.255.255.255
        2. B类：172.16.0.0~172.31.255.255
        3. C类：192.168.0.0~192.168.255.255
6. IP格式



7. 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

环回地址：127.0.0.1。 也是本机地址，等效于localhost或本机IP。一般用于测试使用。例如：ping 127.0.0.1来测试本机网络是否正常。

2.1socket概念

Linux中的**网络编程**是通过**socket接口**（套接口、套接字）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，随后的连接建立等操作都是通过这个socket描述符来实现的！

2.2socket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。
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。

2.3socket信息数据结构

1. 旧类型数据结构：

```
struct sockaddr
{
     sa_family_t  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。*/
};
```

2. 新类型的数据结构：

```
//用于ipv4的结构体（常用）
struct sockaddr_in {
     sa_family_t     sin_family;    /*地址族*/
     in_port_t       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 */
};

//用于ipv6的结构体（不常用）
struct sockaddr_in6 {
    sa_family_t     sin6_family;   /* AF_INET6 */
    in_port_t       sin6_port;     /* port number */
    uint32_t        sin6_flowinfo; /* IPv6 flow information */
    struct in6_addr sin6_addr;     /* IPv6 address */
    uint32_t        sin6_scope_id; /* Scope ID (new in 2.4) */
};

struct in6_addr {
    unsigned char   s6_addr[16];   /* IPv6 address */
};

//sin_family：赋值AF_INET表示ipv4协议，AF_INET6表示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）。

1. 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
2. 大端模式：内存的**高地址**存储数据的低字节，**低地址**存储数据的高字节

例如：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

如果某个系统所采用的字节序为**主机字节序**（本机数据存储方式），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。

Linux提拱了一些函数可以对两种字节序进行转换。

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

//将ip地址由主机字节序转换成网络字节序
uint32_t htonl(uint32_t hostlong);
//将ip地址由网络字节序转换成主机字节序
uint32_t ntohl(uint32_t netlong);

//将端口号由主机字节序转换成网络字节序
uint16_t htons(uint16_t hostshort);
//将端口号由网络字节序转换成主机字节序
uint16_t ntohs(uint16_t netshort);

/* 上面四个函数常用的是端口号的转换，其中htons是最多使用的，ip地址的转换接口基本不用 */
```

代码示例：

```c
#include <head.h>

int main(int argc, char **argv)
{
    short port = 0X1234;
    short nport = 0;

    nport = htons(port);
    printf("nport = %x\n", nport);

    port = 0;
    port = ntohs(nport);
    printf("port = %x\n", port);
    return 0;
}
```

2.5IP地址格式转换

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：

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
//将点分十进制ip转换为网络字节序，参数2保存转换后的结果
int inet_aton(const char *cp, struct in_addr *inp);

//将网络字节序转换为点分十进制ip
char *inet_ntoa(struct in_addr in);

//将点分十进制ip转换为网络字节序，该函数使用的最多
in_addr_t inet_addr(const char *cp);
```

代码示例：

```c
#include <head.h>

int main(int argc, char **argv)
{
    struct in_addr addr;
    memset(&addr, 0, sizeof(addr));

    inet_aton("192.168.130.1", &addr);
    printf("addr = %x\n", addr.s_addr);

    char *ip = inet_ntoa(addr);
    printf("ip = %s\n", ip);
    return 0;
}
```

IPv4和IPv6的函数原型：

```c
#include <arpa/inet.h>
//将点分十进制ip转换为网络字节序
int inet_pton(int af, const char *src, void *dst);

//将网络字节序转换为点分十进制ip
const char *inet_ntop(int af, const void *src, char *dst, socklen_t size);
```

代码示例：

```c
#include <head.h>

int main(int argc, char **argv)
{
    struct in_addr addr;

    //将ipv4类型的ip转换成网络二进制保存在addr结构体中
    inet_pton(AF_INET, "192.168.130.1", &addr);
    printf("addr = %x\n", addr.s_addr);

    //将ipv4类型的网络ip转换成点分十进制，保存在ip中
    char ip[64] = {0};
    const char *p = inet_ntop(AF_INET, &addr, ip, sizeof(ip));
    printf("p = %s\n", p);
    printf("ip = %s\n", ip);
    return 0;
}
```

2.6名字地址转换

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，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来得到百度公司的ip地址，如图。那么，系统是如何将www.baidu.com 这个域名转化为IP地址呢？

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。

```
#include <netdb.h>
#include <sys/socket.h>
struct hostent *gethostbyname(const char *name);
struct hostent *gethostbyaddr(const void *addr, socklen_t len, int type);
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*/
```

代码示例：

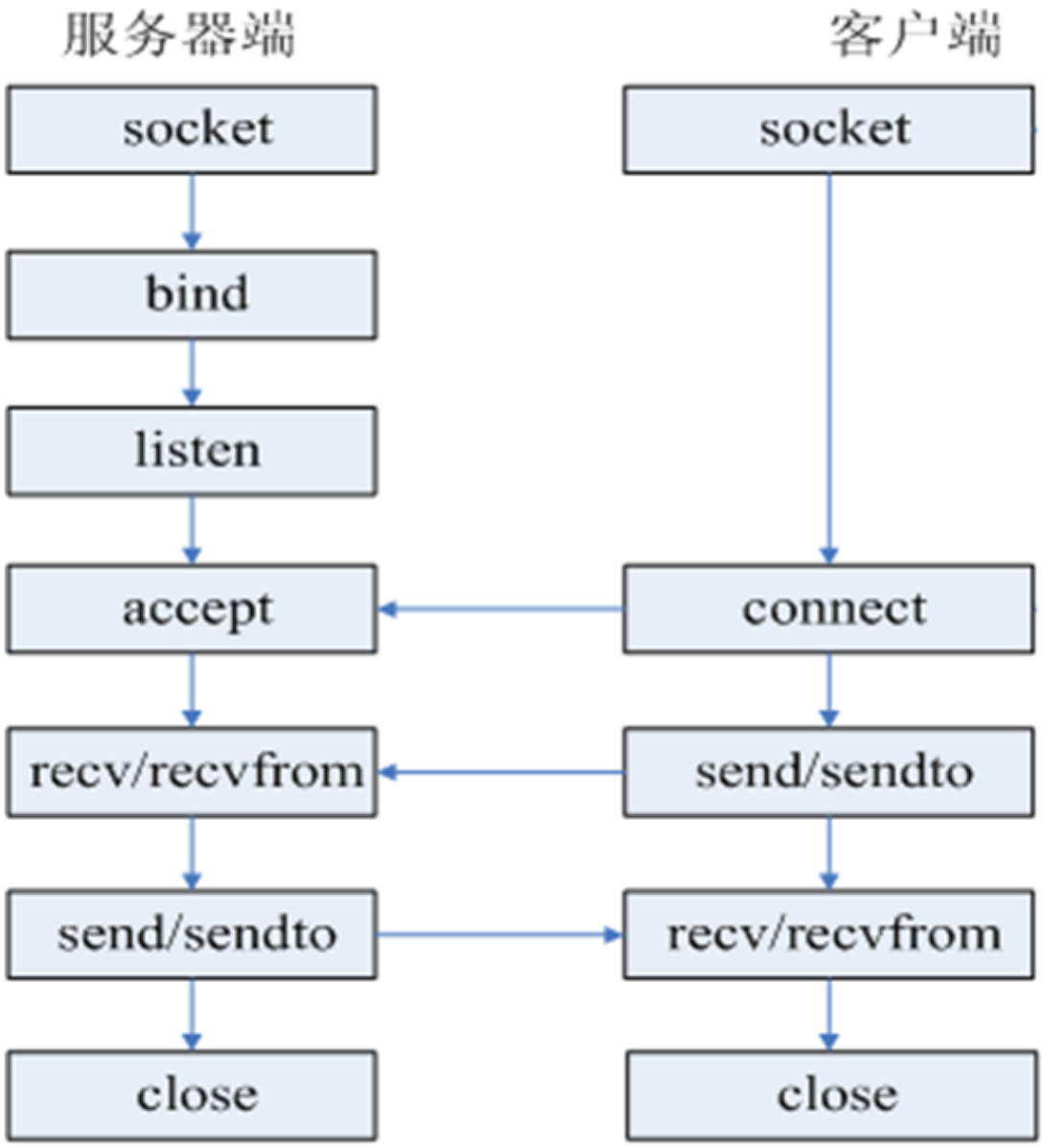
```
//将百度www.baidu.com转换为ip地址
#include <head.h>

int main(int argc, char **argv)
{
    struct hostent *host = NULL;

    host = gethostbyname("www.baidu.com");
    if(NULL == host){
        printf("gethostbyname error!\n");
        return -1;
    }

    //输出正式主机名
    printf("h_name = %s\n", host->h_name);

    //输出主机别名
    for(int i = 0; host->h_aliases[i] != NULL; ++i){
        printf("h_aliases = %s\n", host->h_aliases[i]);
    }
```

```
21
22      //输出主机地址类型
23      printf("h_addr_type = %d\n", host->h_addrtype);
24
25      //输出地址长度
26      printf("h_length = %d\n", host->h_length);
27
28      //输出主机的ip地址列表
29      char buf[64] = {0};
30      for(int i = 0; host->h_addr_list[i] != NULL; ++i){
31          memset(buf, 0, sizeof(buf));
32          inet_ntop(host->h_addrtype, host->h_addr_list[i], buf, sizeof(buf));
33          printf("h_addr = %s\n", buf);
34      }
35
36      return 0;
37  }
```

```
1   #include <head.h>
2
3   int main(int argc, char **argv)
4   {
5       struct hostent *host = NULL;
6       struct in_addr addr;
7       memset(&addr, 0, sizeof(addr));
8       //经测试，并不是所有可见ip都能转换
9       addr.s_addr = inet_addr("35.232.111.17");
10
11      //参数2是ip的长度，参数3是ip类型
12      host = gethostbyaddr(&addr, 4, AF_INET);
13      if(NULL == host){
14          printf("host is NULL\n");
15          return -1;
16      }
17      printf("name = %s\n", host->h_name);
18      return 0;
19  }
```

3.socket编程

3.1使用tcp协议的流程



服务端

1. socket函数：生成一个套接口描述符

```
#include <sys/types.h>          /* See NOTES */
 #include <sys/socket.h>
int socket(int domain, int type, int protocol);
//domin: ip协议类型，AF_INET:ipv4网络协议；AF_INET6:ipv6网络协议
//type:网络协议类型，SOCK_STREAM:tcp协议；SOCK_DGRAM:udp协议
//protocol: socket使用的传输协议编号，通常为0
//成功返回套接字，失败返回-1

示例：
int sfd = socket(AF_INET, SOCK_STREAM, 0);
```

2. bind函数：将端口号和ip地址绑定起来，与socket生成的套接字相关联

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
//sockfd: socket返回的sfd
//addr:结构体指针，指针指向的结构体里面存储需要绑定的ip和端口
```

```
//addrlen: sockaddr_in结构体的长度

//注意：sockaddr_in结构体存储的是网络字节序的ip和端口，所以需要使用htons和inet_addr等函数进程字节序的转化

示例：
struct sockaddr_in addr;
memset(&addr, 0, sizeof(addr));
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = inet_addr("需要转换的ip"); /* 如果为INADDR_ANY，系统会自动填充本机IP地址(全零) */
addr.sin_port = htons(需要转换的端口); /* 如果填0，函数会自动为你选择一个未占用的端口来使用 */
//需要做结构体类型转化
bind(sfd, (struct sockaddr*)&addr, sizeof(addr));
```

3. 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

```
int listen(int sockfd, int backlog);
//sockfd: socket返回值
//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

示例：
listen(sfd, 10);
```

4. 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标识符来处理，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。

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
//addr: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
//成功返回一个新的socket描述符，用于与客户端在建立连接后进行数据的交互

示例：
int newFd = accept(sfd, NULL, NULL);
```

5. 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

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
//sockfd: accept函数返回的新的文件描述符newFd
//buf:表示缓冲区,接收的数据存放在buf里
//len: buf的长度
//flags: recv的属性设置，通常为0
//成功返回实际接收到的字节数，失败返回-1，返回0时表示对端断开连接

示例：
char buf[64] = {0};
recv(newFd, buf, sizeof(buf), 0);
```

6. send函数：用新的套接字发送数据给指定的远端主机

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
//成功返回实际发送的字节数，失败返回-1

示例：
send(newFd, "hello", 5, 0);
```

7. 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

```
#include <unistd.h>
int close(int fd);

示例：
close(newFd);
```

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

```
int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
//sockfd: socket函数的返回值
//addr: 结构体指针，指向的结构体里保存的是服务端的ip和端口
//addrlen: 结构体的大小

示例：
connect(sfd, &addr, sizeof(addr));
```

代码示例：

服务端 (仅实现通信) ：

```
#include <head.h>

int main(int argc, char **argv)
{
    int sfd = socket(AF_INET, SOCK_STREAM, 0);
    ERROR_CHECK(sfd, -1, "socket");

    //保存本机的ip和端口
    struct sockaddr_in serAddr;
    memset(&serAddr, 0, sizeof(serAddr));
    serAddr.sin_family = AF_INET;

    //采用传参的方式，传入ip和端口
    serAddr.sin_addr.s_addr = inet_addr(argv[1]);
    serAddr.sin_port = htons(atoi(argv[2]));

    //绑定本机的IP和端口，绑定到sfd上
    int ret = 0;
    ret = bind(sfd, (struct sockaddr*)&serAddr, sizeof(serAddr));

    //监听的最大连接数是10
    ret = listen(sfd, 10);
    ERROR_CHECK(ret, -1, "listen");
```

```
    //接收连接,返回新的文件描述符，新的描述符作用，与对端进行数据交互使用的
    int newFd = accept(sfd, NULL, NULL);
    ERROR_CHECK(newFd, -1, "accept");

    char buf[64] = {0};
    ret = recv(newFd, buf, sizeof(buf), 0);
    ERROR_CHECK(ret, -1, "recv");
    printf("buf = %s\n", buf);

    ret = send(newFd, "helloclient", 11, 0);
    ERROR_CHECK(ret, -1, "send");

    close(sfd);
    close(newFd);

    return 0;
}
```

客户端

```
#include <head.h>

int main(int argc, char **argv)
{
    int sfd = socket(AF_INET, SOCK_STREAM, 0);
    ERROR_CHECK(sfd, -1, "socket");

    //保存服务端的ip和端口
    struct sockaddr_in serAddr;
    memset(&serAddr, 0, sizeof(serAddr));
    serAddr.sin_family = AF_INET;
    serAddr.sin_addr.s_addr = inet_addr(argv[1]);
    serAddr.sin_port = htons(atoi(argv[2]));

    //连接服务器
    int ret = 0;
    ret = connect(sfd, (struct sockaddr*)&serAddr, sizeof(serAddr));
    ERROR_CHECK(ret, -1, "connect");

    ret = send(sfd, "helloserver", 11, 0);
    ERROR_CHECK(ret, -1, "send");

    char buf[64] = {0};
    ret = recv(sfd, buf, sizeof(buf), 0);
    ERROR_CHECK(ret, -1, "recv");
    printf("buf = %s\n", buf);

    close(sfd);

    return 0;
}
```

升级版（可以实现聊天）

```
#include <head.h>
```

```c

int main(int argc, char **argv)
{
    int sfd = socket(AF_INET, SOCK_STREAM, 0);
    ERROR_CHECK(sfd, -1, "socket");

    //保存本机的ip和端口
    struct sockaddr_in serAddr;
    memset(&serAddr, 0, sizeof(serAddr));
    serAddr.sin_family = AF_INET;
    serAddr.sin_addr.s_addr = inet_addr(argv[1]);
    serAddr.sin_port = htons(atoi(argv[2]));

    //绑定本机的IP和端口，绑定到sfd上
    int ret = 0;
    ret = bind(sfd, (struct sockaddr*)&serAddr, sizeof(serAddr));

    //监听的最大连接数是10
    ret = listen(sfd, 10);
    ERROR_CHECK(ret, -1, "listen");

    //接收连接,返回新的文件描述符，新的描述符作用，与对端进行数据交互使用的
    int newFd = 0;

    //创建监听集合
    fd_set rdset;
    fd_set needMonitorSet;
    FD_ZERO(&needMonitorSet);
    FD_ZERO(&rdset);

    FD_SET(STDIN_FILENO, &needMonitorSet);
    FD_SET(sfd, &needMonitorSet);

    char buf[64] = {0};
    int readyNum = 0;
    while(1){
        //把需要监听的描述符加入到集合当中
        memcpy(&rdset, &needMonitorSet, sizeof(fd_set));
        readyNum = select(10, &rdset, NULL, NULL, NULL);
        ERROR_CHECK(readyNum, -1, "select");
        for(int i = 0; i < readyNum; ++i){

            //如果是标准输入就绪，就代表需要发送数据给对端
            if(FD_ISSET(STDIN_FILENO, &rdset)){
                memset(buf, 0, sizeof(buf));
                read(STDIN_FILENO, buf, sizeof(buf));
                send(newFd, buf, strlen(buf)-1, 0);
            }

            //如果是newFd就绪，就说明对端有数据发送给我们
            if(FD_ISSET(newFd, &rdset)){
                memset(buf, 0, sizeof(buf));
                ret = recv(newFd, buf, sizeof(buf)-1, 0);
                if(0 == ret){
                    close(newFd);
                    FD_CLR(newFd, &needMonitorSet);
                    printf("bey bey\n");
                    continue;
```

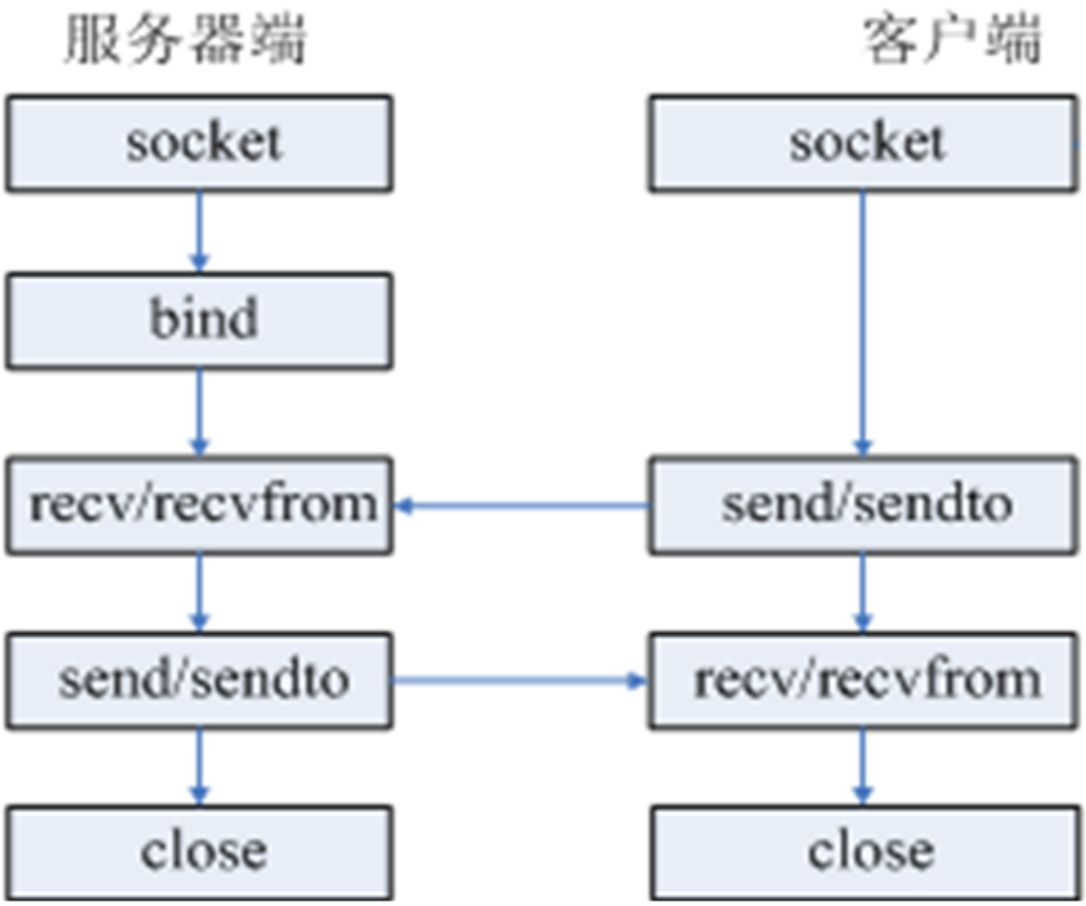
```
                }
                printf("buf = %s\n", buf);
            }
            if(FD_ISSET(sfd, &rdset)){
                newFd = accept(sfd, NULL, NULL);
                ERROR_CHECK(newFd, -1, "accept");
                printf("client connect\n");

                FD_SET(newFd, &needMonitorSet);
            }
        }
    }
    close(sfd);
    close(newFd);

    return 0;
}
```

客户端

```
#include <head.h>

int main(int argc, char **argv)
{
    int sfd = socket(AF_INET, SOCK_STREAM, 0);
    ERROR_CHECK(sfd, -1, "socket");

    //保存服务端的ip和端口
    struct sockaddr_in serAddr;
    memset(&serAddr, 0, sizeof(serAddr));
    serAddr.sin_family = AF_INET;
    serAddr.sin_addr.s_addr = inet_addr(argv[1]);
    serAddr.sin_port = htons(atoi(argv[2]));

    int ret = 0;
    ret = connect(sfd, (struct sockaddr*)&serAddr, sizeof(serAddr));
    ERROR_CHECK(ret, -1, "connect");

    //创建监听集合
    fd_set rdset;
    FD_ZERO(&rdset);

    char buf[64] = {0};
    int readyNum = 0;
    while(1){
        //把需要监听的描述符加入到集合当中
        FD_SET(STDIN_FILENO, &rdset);
        FD_SET(sfd, &rdset);
        readyNum = select(sfd + 1, &rdset, NULL, NULL, NULL);
        for(int i = 0; i < readyNum; ++i){

            //如果是标准输入就绪，就代表需要发送数据给对端
            if(FD_ISSET(STDIN_FILENO, &rdset)){
                memset(buf, 0, sizeof(buf));
                read(STDIN_FILENO, buf, sizeof(buf));
                send(sfd, buf, strlen(buf)-1, 0);
            }
```

```
38
39              //如果是sfd就绪，就说明对端有数据发送给我们
40              if(FD_ISSET(sfd, &rdset)){
41                  memset(buf, 0, sizeof(buf));
42                  ret = recv(sfd, buf, sizeof(buf)-1, 0);
43                  if(0 == ret){
44                      printf("server close\n");
45                      close(sfd);
46                      return 0;
47                  }
48                  printf("buf = %s\n", buf);
49              }
50          }
51      }
52      close(sfd);
53
54      return 0;
55  }
```

3.2使用UDP协议的流程图



服务端

1. sendto函数：发送数据

```
ssize_t sendto(int sockfd, const void *buf, size_t len, int flags, const
struct sockaddr *dest_addr, socklen_t addrlen);

//前四个参数和send函数相同
//dest_addr: 结构体指针，指向的结构体保存对端的ip和端口
```

2. recvfrom函数：接收数据

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct
sockaddr *src_addr, socklen_t *addrlen);

//前四个参数和recv函数相同
//src_addr: 结构体指针，指向的结构体保存对端的ip和端口
```

注意：send/recv这对函数常用于tcp通信；sendto/recvfrom这对函数常用于udp通信也可用于tcp通信

代码示例：

服务端（仅实现通信）：

```
#include <head.h>

int main(int argc, char **argv)
{
    //udp的通信
    int sfd = socket(AF_INET, SOCK_DGRAM, 0);
    ERROR_CHECK(sfd, -1, "socket");

    //保存本机的ip和端口
    struct sockaddr_in serAddr;
    memset(&serAddr, 0, sizeof(serAddr));
    serAddr.sin_family = AF_INET;
    serAddr.sin_addr.s_addr = inet_addr(argv[1]);
    serAddr.sin_port = htons(atoi(argv[2]));

    //绑定本机的IP和端口，绑定到sfd上
    int ret = 0;
    ret = bind(sfd, (struct sockaddr*)&serAddr, sizeof(serAddr));
    ERROR_CHECK(ret, -1, "bind");

    struct sockaddr_in cliAddr;
    socklen_t len = sizeof(cliAddr);
    memset(&cliAddr, 0, len);

    //服务端需要先recvfrom接收对端的消息，并保存对端的ip和端口号
    char buf[64] = {0};
    ret = recvfrom(sfd, buf, sizeof(buf), 0, (struct sockaddr*)&cliAddr,
&len);
    ERROR_CHECK(ret, -1, "recv");
    printf("buf = %s\n", buf);

    ret = sendto(sfd, "helloclient", 11, 0, (struct sockaddr*)&cliAddr,
len);
    ERROR_CHECK(ret, -1, "send");

```

```
    close(sfd);

    return 0;
}
```

客户端：

```
#include <head.h>

int main(int argc, char **argv)
{
    //udp的通信
    int sfd = socket(AF_INET, SOCK_DGRAM, 0);
    ERROR_CHECK(sfd, -1, "socket");

    //保存服务端的ip和端口
    struct sockaddr_in serAddr;
    memset(&serAddr, 0, sizeof(serAddr));
    serAddr.sin_family = AF_INET;
    serAddr.sin_addr.s_addr = inet_addr(argv[1]);
    serAddr.sin_port = htons(atoi(argv[2]));

    int ret = 0;
    socklen_t len = sizeof(serAddr);

    //客户端需要先发送数据给服务端
    char buf[64] = {0};
    ret = sendto(sfd, "helloserver", 11, 0, (struct sockaddr*)&serAddr,
len);
    ERROR_CHECK(ret, -1, "send");

    ret = recvfrom(sfd, buf, sizeof(buf), 0, (struct sockaddr*)&serAddr,
&len);
    ERROR_CHECK(ret, -1, "recv");
    printf("buf = %s\n", buf);

    close(sfd);

    return 0;
}
```

代码示例（可以实现聊天）：

服务端：

```
#include <head.h>

int main(int argc, char **argv)
{
    //udp的通信
    int sfd = socket(AF_INET, SOCK_DGRAM, 0);
    ERROR_CHECK(sfd, -1, "socket");

    //保存本机的ip和端口
    struct sockaddr_in serAddr;
    memset(&serAddr, 0, sizeof(serAddr));
    serAddr.sin_family = AF_INET;
```

```c
    serAddr.sin_addr.s_addr = inet_addr(argv[1]);
    serAddr.sin_port = htons(atoi(argv[2]));

    //绑定本机的IP和端口，绑定到sfd上
    int ret = 0;
    ret = bind(sfd, (struct sockaddr*)&serAddr, sizeof(serAddr));
    ERROR_CHECK(ret, -1, "bind");

    struct sockaddr_in cliAddr;
    socklen_t len = sizeof(cliAddr);
    memset(&cliAddr, 0, len);

    fd_set rdset;
    FD_ZERO(&rdset);

    int readyNum = 0;
    char buf[64] = {0};

    while(1){
        FD_SET(STDIN_FILENO, &rdset);
        FD_SET(sfd, &rdset);
        readyNum = select(sfd + 1, &rdset, NULL, NULL, NULL);
        ERROR_CHECK(readyNum, -1, "select");
        for(int i = 0; i < readyNum; ++i){
            if(FD_ISSET(STDIN_FILENO, &rdset)){
                memset(buf, 0, sizeof(buf));
                read(STDIN_FILENO, buf, sizeof(buf));
                ret = sendto(sfd, buf, strlen(buf)-1, 0, (struct
sockaddr*)&cliAddr, len);
                ERROR_CHECK(ret, -1, "send");
            }
            if(FD_ISSET(sfd, &rdset)){
                memset(buf, 0, sizeof(buf));
                ret = recvfrom(sfd, buf, sizeof(buf)-1, 0, (struct
sockaddr*)&cliAddr, &len);
                ERROR_CHECK(ret, -1, "recv");
                printf("buf = %s\n", buf);
            }
        }

    }

    close(sfd);

    return 0;
}
```

客户端：

```c
#include <head.h>

int main(int argc, char **argv)
{
    //udp的通信
    int sfd = socket(AF_INET, SOCK_DGRAM, 0);
    ERROR_CHECK(sfd, -1, "socket");

```

```
    //保存服务端的ip和端口
    struct sockaddr_in serAddr;
    memset(&serAddr, 0, sizeof(serAddr));
    serAddr.sin_family = AF_INET;
    serAddr.sin_addr.s_addr = inet_addr(argv[1]);
    serAddr.sin_port = htons(atoi(argv[2]));

    fd_set rdset;
    FD_ZERO(&rdset);
    socklen_t len = sizeof(serAddr);
    int ret = 0;

    int readyNum = 0;
    char buf[64] = {0};

    while(1){
        FD_SET(STDIN_FILENO, &rdset);
        FD_SET(sfd, &rdset);
        readyNum = select(sfd + 1, &rdset, NULL, NULL, NULL);
        ERROR_CHECK(readyNum, -1, "select");
        for(int i = 0; i < readyNum; ++i){
            if(FD_ISSET(STDIN_FILENO, &rdset)){
                memset(buf, 0, sizeof(buf));
                read(STDIN_FILENO, buf, sizeof(buf));
                ret = sendto(sfd, buf, strlen(buf)-1, 0, (struct
sockaddr*)&serAddr, len);
                ERROR_CHECK(ret, -1, "send");
            }
            if(FD_ISSET(sfd, &rdset)){
                memset(buf, 0, sizeof(buf));
                ret = recvfrom(sfd, buf, sizeof(buf)-1, 0, (struct
sockaddr*)&serAddr, &len);
                ERROR_CHECK(ret, -1, "recv");
                printf("buf = %s\n", buf);
            }
        }

    }

    close(sfd);

    return 0;
}
```

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，那么在填充完内存后，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。

3.3设置套接口的选项setsockopt的用法

```
#include <sys/types.h>
#include <sys/socket.h>
int setsockopt(int sockfd, int level, int optname, const void *optval, socklen_t optlen);
//sockfd: 需要设置的套接口
//level: 选项定义的层次
//optname: 需要设置的选项（根据对应层次设置选项）
//optval: 指针，指向存放选项值的缓冲区，选项值是对选项是否生效做说明，选项值大于0，代表选项生效
//optlen: optval缓冲区长度
```

层次：SOL_SOCKET

选项名称	说明	数据类型
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

层次：IPPROTO_IP

选项名称	说明	数据类型
IP_HDRINCL	在数据包中包含IP首部	int
IP_OPTINOS	IP首部选项	int
IP_TOS	服务类型	
IP_TTL	**生存时间**	int

层次：IPPRO_TCP

选项名称	说明	数据类型
TCP_MAXSEG	TCP最大数据段的大小	int
TCP_NODELAY	不使用Nagle算法	int

注意：设置套接口选项的时候，需要在bind绑定前设置。

代码示例：设置地址可重用

```
//设置地址可重用
int reuse = 1; //设置值大于0，表示生效
setsockopt(sfd, SOL_SOCKET, SO_REUSEADDR, &reuse, sizeof(int));
```

3.4获取套接口信息getsockopt

```
int getsockopt(int sockfd, int level, int optname, void *optval, socklen_t
*optlen);
```

代码示例：获取发送缓冲区的小大

```
socklen_t sndBuf = 0;
socklen_t len = sizeof(sndBuf);
getsockopt(sfd, SOL_SOCKET, SO_SNDBUF, &sndBuf, &len);
```

3.5单播、广播、多播（组播）（了解）

多播广播是用于建立分步式系统：例如网络游戏、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。

3.6DDos攻击，防护（了解）

基本概念

分布式拒绝服务攻击(英文意思是Distributed Denial of Service，简称DDoS)是指处于不同位置的多个攻击者同时向一个或数个目标发动攻击，或者一个攻击者控制了位于不同位置的多台机器并利用这些机器对受害者同时实施攻击。由于攻击的发出点是分布在不同地方的，这类攻击称为分布式拒绝服务攻击，其中的攻击者可以有多个。

分布式拒绝服务攻击可以使很多的计算机在同一时间遭受到攻击，使攻击的目标无法正常使用，分布式拒绝服务攻击已经出现了很多次，导致很多的大型网站都出现了无法进行操作的情况，这样不仅仅会影响用户的正常使用，同时造成的经济损失也是非常巨大的。

分布式拒绝服务攻击方式在进行攻击的时候，可以对源IP地址进行伪造，这样就使得这种攻击在发生的时候隐蔽性是非常好的，同时要对攻击进行检测也是非常困难的，因此这种攻击方式也成为了非常难以防范的攻击。

攻击方式

1、SYN Flood攻击

SYN Flood攻击是当前网络上最为常见的DDoS攻击，它利用了TCP协议实现上的一个缺陷。通过向网络服务所在端口发送大量的伪造源地址的攻击报文，就可能造成目标服务器中的半开连接队列被占满，从而阻止其他合法用户进行访问。

2、UDP Flood攻击

UDP Flood是日渐猖厥的流量型DDoS攻击，原理也很简单。常见的情况是利用大量UDP小包冲击DNS服务器或Radius认证服务器、流媒体视频服务器。由于UDP协议是一种无连接的服务，在UDP Flood攻击中，攻击者可发送大量伪造源IP地址的小UDP包。

3、ICMP Flood攻击

ICMP Flood攻击属于流量型的攻击方式，是利用大的流量给服务器带来较大的负载，影响服务器的正常服务。由于目前很多防火墙直接过滤ICMP报文。因此ICMP Flood出现的频度较低。

4、Connection Flood攻击

Connection Flood是典型的利用小流量冲击大带宽网络服务的攻击方式，这种攻击的原理是利用真实的IP地址向服务器发起大量的连接。并且建立连接之后很长时间不释放，占用服务器的资源，造成服务器上残余连接(WAIT状态)过多，效率降低，甚至资源耗尽，无法响应其他客户所发起的链接。

5、HTTP Get攻击

这种攻击主要是针对存在ASP、JSP、PHP、CGI等脚本程序，特征是和服务器建立正常的TCP连接，并不断的向脚本程序提交查询、列表等大量耗费数据库资源的调用。这种攻击的特点是可以绕过普通的防火墙防护，可通过Proxy代理实施攻击，缺点是攻击静态页面的网站效果不佳，会暴露攻击者的IP地址。

6、UDP DNS Query Flood攻击

UDP DNS Query Flood攻击采用的方法是向被攻击的服务器发送大量的域名解析请求，通常请求解析的域名是随机生成或者是网络世界上根本不存在的域名。域名解析的过程给服务器带来了很大的负载，每秒钟域名解析请求超过一定的数量就会造成DNS服务器解析域名超时。

防御措施

不但是对DDoS，而且是对于所有网络的攻击，都应该是采取尽可能周密的防御措施，同时加强对系统的检测，建立迅速有效的应对策略。应该采取的防御措施有：

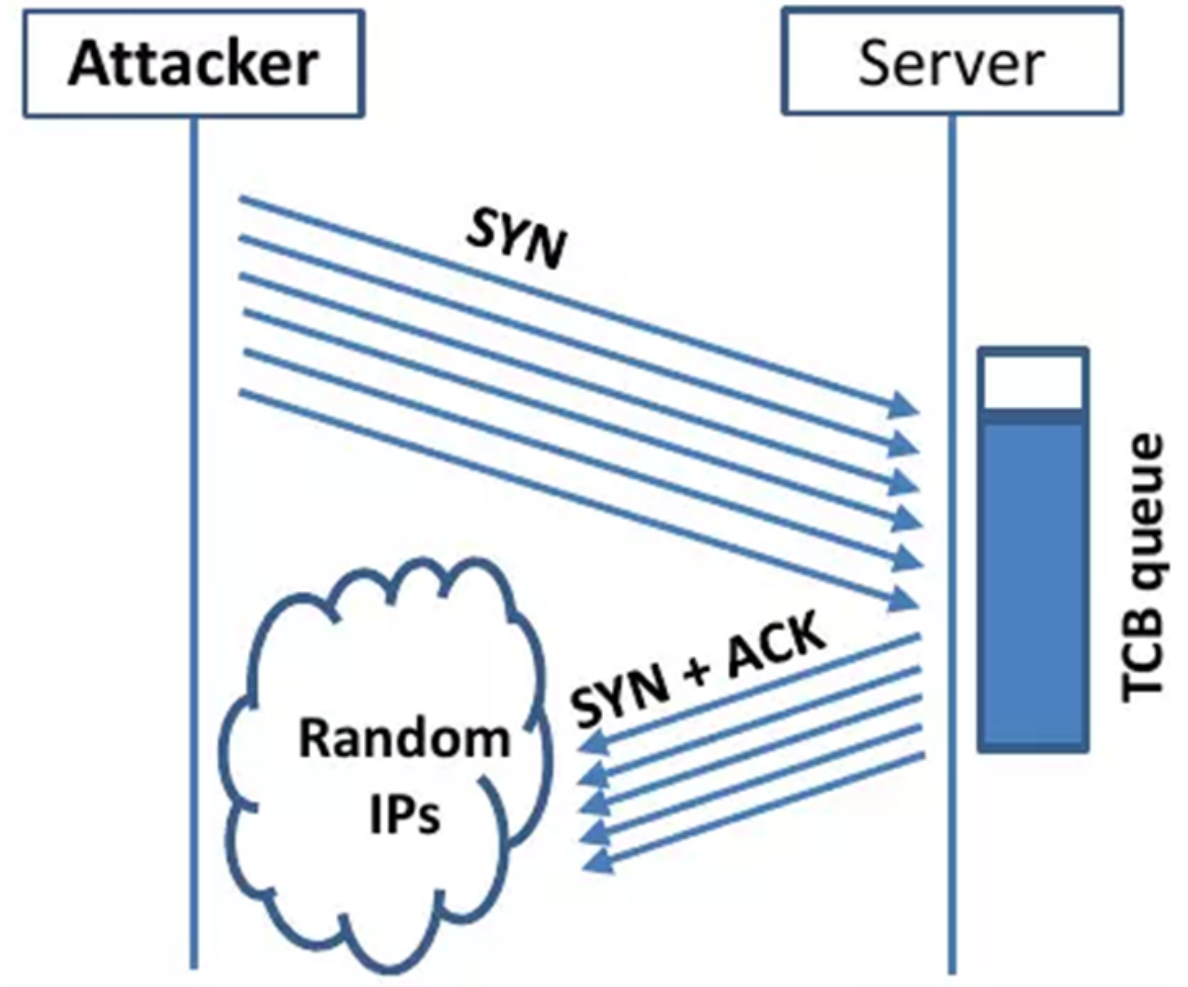
1. 全面综合地设计网络的安全体系，注意所使用的安全产品和网络设备。
2. 提高网络管理人员的素质，关注安全信息，遵从有关安全措施，及时地升级系统，加强系统抗击攻击的能力。
3. 在系统中加装防火墙系统，利用防火墙系统对所有出入的数据包进行过滤，检查边界安全规则，确保输出的包受到正确限制。
4. 优化路由及网络结构。对路由器进行合理设置，降低攻击的可能性。
5. 优化对外提供服务的主机，对所有在网上提供公开服务的主机都加以限制。
6. 安装入侵检测工具(如NIPC、NGREP)，经常扫描检查系统，解决系统的漏洞，对系统文件和应用程序进行加密，并定期检查这些文件的变化。

SYN Flood攻击示例

攻击方式

攻击者构造网络包，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

攻击示意图



防护手段

1. 增加半连接队列长度

2. 减小SYN+ACK重传次数
3. 启用syn cookie（SYN Cookie是对TCP服务器端的三次握手协议作一些修改，专门用来防范SYN Flood攻击的一种手段，该方式也是更有效的防御方式，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）

3.7描述符属性修改及文件描述符的传递

修改文件描述符属性

fcntl函数简介：fcntl函数可以改变已打开的文件描述符性质

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

int fcntl(int fd, int cmd, ... /* arg */ );
//fd: 需要设置的文件描述符
//cmd: 需要操作的指令
//可边长参数: 根据不同的cmd，有不同的填法
//fcntl的返回值与命令有关。如果出错，所有命令都返回-1，如果成功则返回某个其他值
```

fcntl函数有5种功能：

1. 复制一个现有的描述符（cmd=F_DUPFD）
2. 获得 / 设置文件描述符标记(cmd=F_GETFD或F_SETFD)
3. 获得 / 设置文件描述符状态(cmd=F_GETFL或F_SETFL)
4. 获得 / 设置异步I/O所有权(cmd=F_GETOWN或F_SETOWN)
5. 获得 / 设置记录锁(cmd=F_GETLK,F_SETLK或F_SETLKW)

cmd有如下选项设置：

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符，并且复制参数fd的文件描述符。执行成功则返回新复制的文件描述符。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（在计算机科学中，旗标被理解为一个单个整型值，结合一对被称为P和V的函数。）。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 取得文件描述符状态旗标，此旗标为open（）的参数flags。

F_SETFL 设置文件描述符状态旗标，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。

F_GETLK 取得文件锁定的状态。

F_SETLK 设置文件锁定的状态。此时flcok 结构的l_type 值必须是F_RDLCK、F_WRLCK或F_UNLCK。如果无法建立锁定，则返回-1，错误代码为EACCES 或EAGAIN。

F_SETLKW F_SETLK 作用相同，但是无法建立锁定时，此调用会一直等到锁定动作成功为止。若在等待锁定的过程中被信号中断时，会立即返回-1，错误代码为EINTR。

代码示例：

```
#include <head.h>

void setNonBlock(int fd)
```

```
{
    int status = 0;

    //获取fd的状态
    status = fcntl(fd, F_GETFL);

    //设置状态为非阻塞
    status |= O_NONBLOCK;
    fcntl(fd, F_SETFL, status);
}
int main(int argc, char **argv)
{
    setNonBlock(STDIN_FILENO);

    char buf[64] = {0};
    read(STDIN_FILENO, buf, sizeof(buf));

    printf("buf = %s\n", buf);
    return 0;
}
```

进程间传递文件描述符

进程间传递描述符需要使用三个函数

socketpair函数简介：建立一对匿名的已经连接的套接字

```
int socketpair(int domain, int type, int protocol, int sv[2]);
//domain: 填AF_LOCAL
//sv: 填大小为2的数组，保存两个套接字
```

sendmsg函数简介：用于发送消息到另一个套接字

```
ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
//msg: 指针指向的结构体存储需要发送的数据
//flags: 通常为0

struct msghdr {
    void         *msg_name;       /* optional address */  套接口地址，应用于udp
    socklen_t    msg_namelen;     /* size of address */ 套接口地址长度
    struct       iovec *msg_iov;  /* scatter/gather array */ 指向iovec结构体数组
    size_t       msg_iovlen;      /* # elements in msg_iov */ 结构体数组大小
    void         *msg_control;    /* ancillary data, see below */ 附属数据，指向cmsghdr结构体
    size_t       msg_controllen;  /* ancillary data buffer len */  cmsghdr结构体的长度
    int          msg_flags;       /* flags (unused) */ 标志位，没有使用（用于接收特定的标志位）
};

/* iovec必须赋值 */
struct iovec {
    void  *iov_base;    /* Starting address */ 指向缓冲区的起始位置，缓冲区用于存储writev所接收/要发送的数据
    size_t iov_len;     /* Number of bytes to transfer */要传输的字节数
};
```

```
20
21 /* 存放附属数据结构体 */
22 struct cmsghdr {
23     socklen_t cmsg_len;    /* data byte count, including header */结构体大小、这个值可由CMSG_LEN()宏计算
24     int       cmsg_level;  /* originating protocol */原始协议，填SOL_SOCKET
25     int       cmsg_type;   /* protocol-specific type */特定协议类型，填SCM_RIGHTS
26     /* followed by
27     unsigned char cmsg_data[]; */可变长数组，存放附属数据，使用CMSG_DATA()接收
28 };
```

recvmsg函数简介：从一个套接字上接收数据

```
1 ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
```

iovec结构体是保存数据的，如果创建的结构体数组，那么读写数据的时候需要使用到writev和readv两个函数

```
1 ssize_t writev(int fd, const struct iovec *iov, int iovcnt);
2 ssize_t readv(int fd, const struct iovec *iov, int iovcnt);
3 //iov: 指向结构体
4 //iovcnt: 结构体数组大小
```

代码示例：iovec的使用

```
1  #include <head.h>
2
3  int main(int argc, char **argv)
4  {
5      struct iovec iov[2];
6      memset(iov, 0, sizeof(iov));
7
8      char buf1[64] = {0};
9      strcpy(buf1, "hello");
10
11     iov[0].iov_base = buf1;
12     iov[0].iov_len = strlen(buf1);
13
14     char buf2[64] = {0};
15     strcpy(buf2, "world");
16     iov[1].iov_base = buf2;
17     iov[1].iov_len = strlen(buf2);
18
19     writev(STDOUT_FILENO, iov, 2);
20     return 0;
21 }
```

4.epoll多路复用

4.1背景简介

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，将用户关心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

```
#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. epoll_create(int size)：创建一个epoll的实例，并返回指向该实例的句柄（也是文件描述符），size参数已经没有意义，但是必须大于0，一般填1即可。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，所以在使用完epoll后，必须调用close()关闭，否则可能导致fd被耗尽。
2. epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)：epoll的事件注册函数，它不同于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：

   1. EPOLL_CTL_ADD：注册新的fd到epfd中；
   2. EPOLL_CTL_MOD：修改已经注册的fd的监听事件；
   3. 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 */
};

typedef union epoll_data {
    void        *ptr;
    int          fd;
    uint32_t     u32;
    uint64_t     u64;
} epoll_data_t;
```

   events可以是以下几个宏的集合：

   1. EPOLLIN ：表示对应的文件描述符可以读（包括对端SOCKET正常关闭）；
   2. EPOLLOUT：表示对应的文件描述符可以写；
   3. 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）；
   4. EPOLLERR：表示对应的文件描述符发生错误；
   5. EPOLLHUP：表示对应的文件描述符被挂断；
   6. 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。

7. 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

3. 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。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。

4.3epoll对文件描述符操作的两种模式

LT模式：Level_trigger(水平触发)：当被监控的文件描述符上有可读写事件发生时，epoll_wait()会通知处理程序去读写。如果这次没有把数据一次性全部读写完(如读写缓冲区太小)，那么下次调用epoll_wait()时，它还会通知你在上次没读写完的文件描述符上继续读写

ET模式：Edge_trigger(边缘触发)：当被监控的文件描述符上有可读写事件发生时，epoll_wait()会通知处理程序去读写。如果这次没有把数据全部读写完(如读写缓冲区太小)，那么下次调用epoll_wait()时，它不会通知你，也就是它只会通知你一次，直到该文件描述符上出现第二次可读写事件才会通知你

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。

代码示例：服务端

```
#include <head.h>

int main(int argc, char **argv)
{
    int sfd = socket(AF_INET, SOCK_STREAM, 0);
    ERROR_CHECK(sfd, -1, "socket");

    //保存本机的ip和端口
    struct sockaddr_in serAddr;
    memset(&serAddr, 0, sizeof(serAddr));
    serAddr.sin_family = AF_INET;
    serAddr.sin_addr.s_addr = inet_addr(argv[1]);
    serAddr.sin_port = htons(atoi(argv[2]));

    //绑定本机的IP和端口，绑定到sfd上
    int ret = 0;
    ret = bind(sfd, (struct sockaddr*)&serAddr, sizeof(serAddr));

    //监听的最大连接数是10
    ret = listen(sfd, 10);
    ERROR_CHECK(ret, -1, "listen");

    //接收连接,返回新的文件描述符，新的描述符作用，与对端进行数据交互使用的
    int newFd = accept(sfd, NULL, NULL);
    ERROR_CHECK(newFd, -1, "accept");

    //创建epoll,参数必须大于0
    int epfd = epoll_create(1);
    ERROR_CHECK(epfd, -1, "epoll_create");

    struct epoll_event events, evs[2];
    memset(&events, 0, sizeof(events));
    //告诉内核监听读事件
    events.events = EPOLLIN;
```

```
    events.data.fd = STDIN_FILENO;
    ret = epoll_ctl(epfd, EPOLL_CTL_ADD, STDIN_FILENO, &events);
    ERROR_CHECK(ret, -1, "epoll_ctl");

    events.events = EPOLLIN;
    events.data.fd = newFd;
    ret = epoll_ctl(epfd, EPOLL_CTL_ADD, newFd, &events);
    ERROR_CHECK(ret, -1, "epoll_ctl");


    char buf[64] = {0};
    int readyNum = 0;
    while(1){
        readyNum = epoll_wait(epfd, evs, 2, -1);
        for(int i = 0; i < readyNum; ++i){

            //如果是标准输入就绪，就代表需要发送数据给对端
            if(evs[i].data.fd == STDIN_FILENO){
                memset(buf, 0, sizeof(buf));
                read(STDIN_FILENO, buf, sizeof(buf));
                send(newFd, buf, strlen(buf)-1, 0);
            }

            //如果是newFd就绪，就说明对端有数据发送给我们
            if(evs[i].data.fd == newFd){
                memset(buf, 0, sizeof(buf));
                ret = recv(newFd, buf, sizeof(buf)-1, 0);
                if(0 == ret){
                    close(newFd);
                    printf("bey bey\n");
                    continue;
                }
                printf("buf = %s\n", buf);
            }
        }
    }
    close(sfd);
    close(newFd);

    return 0;
}
```

客户端：

```
#include <head.h>

int main(int argc, char **argv)
{
    int sfd = socket(AF_INET, SOCK_STREAM, 0);
    ERROR_CHECK(sfd, -1, "socket");

    //保存服务器端的ip和端口
    struct sockaddr_in serAddr;
    memset(&serAddr, 0, sizeof(serAddr));
    serAddr.sin_family = AF_INET;
    serAddr.sin_addr.s_addr = inet_addr(argv[1]);
    serAddr.sin_port = htons(atoi(argv[2]));
```

```
14
15      int ret = 0;
16      ret = connect(sfd, (struct sockaddr*)&serAddr, sizeof(serAddr));
17      ERROR_CHECK(ret, -1, "connect");
18
19      int epfd = epoll_create(1);
20      ERROR_CHECK(epfd, -1, "epoll_create");
21
22      struct epoll_event event, evs[2];
23      memset(&event, 0, sizeof(event));
24      memset(evs, 0, sizeof(evs));
25
26      event.events = EPOLLIN;
27      event.data.fd = STDIN_FILENO;
28      ret = epoll_ctl(epfd, EPOLL_CTL_ADD, STDIN_FILENO, &event);
29      ERROR_CHECK(ret, -1, "epoll_ctl1");
30
31      event.data.fd = sfd;
32      ret = epoll_ctl(epfd, EPOLL_CTL_ADD, sfd, &event);
33      ERROR_CHECK(ret, -1, "epoll_ctl2");
34
35      char buf[64] = {0};
36      int readyNum = 0;
37      while(1){
38          readyNum = epoll_wait(epfd, evs, 2, -1);
39          for(int i = 0; i < readyNum; ++i){
40
41              //如果是标准输入就绪，就代表需要发送数据给对端
42              if(evs[i].data.fd == STDIN_FILENO){
43                  memset(buf, 0, sizeof(buf));
44                  read(STDIN_FILENO, buf, sizeof(buf));
45                  send(sfd, buf, strlen(buf)-1, 0);
46              }
47
48              //如果是sfd就绪，就说明对端有数据发送给我们
49              if(evs[i].data.fd == sfd){
50                  memset(buf, 0, sizeof(buf));
51                  recv(sfd, buf, sizeof(buf)-1, 0);
52                  printf("buf = %s\n", buf);
53              }
54          }
55      }
56      close(sfd);
57
58      return 0;
59  }
```

4.4epoll与select的对比

4.4.1select原理：

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。

4.4.2epoll的原理

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。

1. 执行epoll_create时，创建了红黑树和就绪链表；
2. 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;
3. 执行epoll_wait时立刻返回准备就绪链表里的数据即可。

4.4.3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*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_ctl的时候把描述符添加到红黑树，不需要每次拷贝。

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。如果没有大量的空闲连接或者无效链接，epoll的效率并不会比select/poll高很多，但是当遇到大量的空闲连接，就会发现epoll的效率大大高于select/poll

5.五种I/O模型（了解）

5.1基本概念

同步、异步、阻塞、非阻塞

5.2模型

5.2.1. 阻塞I/O

几乎所有套接字默认的都是阻塞的，以recvfrom系统调用为例子，它要等到有数据报到达且被复制到应用进程的缓冲区中或者发生了错误才返回。若没有数据到达那么将一直会阻塞。

5.2.2. 非阻塞I/O

进程将一个套接字设置为非阻塞就是通知内核：当前所请求的IO操作在请求的过程不需要把进程投入睡眠，而是返回一个错误。（注意这里是指请求IO操作，不是进行IO操作）。

5.2.3. I/O复用

通过系统调用select、poll、epoll等实现IO复用模型。此时进程就会组赛在这些系统调用上，而不是阻塞在真正的IO操作上，直到有就绪事件了，这些系统调用就会返回哪些套接字可读写，然后就可以进行把数据包复制到应用进程缓冲区了。

5.2.4. 信号驱动I/O (SIGIO)

用信号让内核在文件描述符准备就绪的时候通知用户进程，即是告知我们什么时候可以启动IO操作。就如数据准备好了，内核就会以一种形式通知用户进程。

5.2.5. 异步I/O

异步 I/O 的读写操作总是立即返回，而不论 I/O 是否是阻塞的，真正的读写操作由内核接管。

异步IO由POSIX规范定义的。一般地说，这些函数的工作机制就是：由用户进程告知内核启动一个操作，并且由内核去操作，操作完后给用户进程发一个通知，通知用户进程操作完了（包括数据从内核缓冲区拷贝到用户缓冲区的过程）。该模型与信号驱动式IO模型不同的就是，异步IO模型中，是由内核通知IO操作什么时候完成，而信号驱动式IO是由内核告知何时启动IO操作。