计算机网络复习

费曼学习法有云： 把一个领域里艰深的公式定理或者道理讲得让小孩子也能理解听懂，那么才算是真正地理解。

而我这里的方法是：一知半解，用自己的话讲出来，后面反复修正，逐渐补全。

1.概述

列车问题： 发送-传播-转发-接收

总时延 = 发送时延 + 传播时延 + 转发时延迟

理想发送时延 = $\frac{\text{数据块长度}}{\text{带宽}}$

传播时延 = $\frac{\text{传输信道长度}}{\text{数据传输速率}}$

理想状态不考虑转发时延迟。

OSI体系结构

物数网运会表应。

- 物理层
- 数据链路层
- 网络层
- 运输层
- 会话层
- 表示层
- 应用层

这些其实就是之后章节的主题了。

2.物理层

数据的存储，转换和传输

数据的存储，是bit的形式，是数字信号。

但是它并不适合直接传输，而需要进行调制，调成模拟信号，然后再进行传输，最后再转换回数字信号。应该是为了减少信道干扰。

而似乎充当调制和解调的是光猫？猫？喵？在大概十年前我还能见到它，现在似乎见不到了。

勘误：似乎是 **光纤终端** 进行的调制和解调

信息，数据，信号

信息是物理世界的概念，数据是计算机中的二进制流存储形式，信号是数据传输时的体现。

数据一般是离散的，而信号一般是连续的。但不管数据还是信号，实际上都是对信息的采样和模拟。

信道

传输信号的通道叫做信道，是我们上述列车问题中的轨道。

理想状态信道的极限传输速率：

$C = 2W \log_2 V$

单位是bps.

C是我们要求的，极限速率。

W是信道带宽，

V是码元的种类。

为什么最大传输速率可以超出带宽，因为带宽是频率带宽度，而发送速率则是单位体积。还和码元种类（调制波形种类）有关。

W 是频率，是Hz,可以这么理解，W 是功率，而频率越高，功率越高。

实际上（高斯白噪声下）信道极限传输速率（香农提的）

$C = W \log_2(1 + \frac{S}{N})$

$\text{SNR (dB)} = 10 \log_{10} \left(\frac{S}{N}\right)$

新增参数:

S: 信号的功率，一般来说，频率越高，功率越高

N:高斯噪声信号的功率

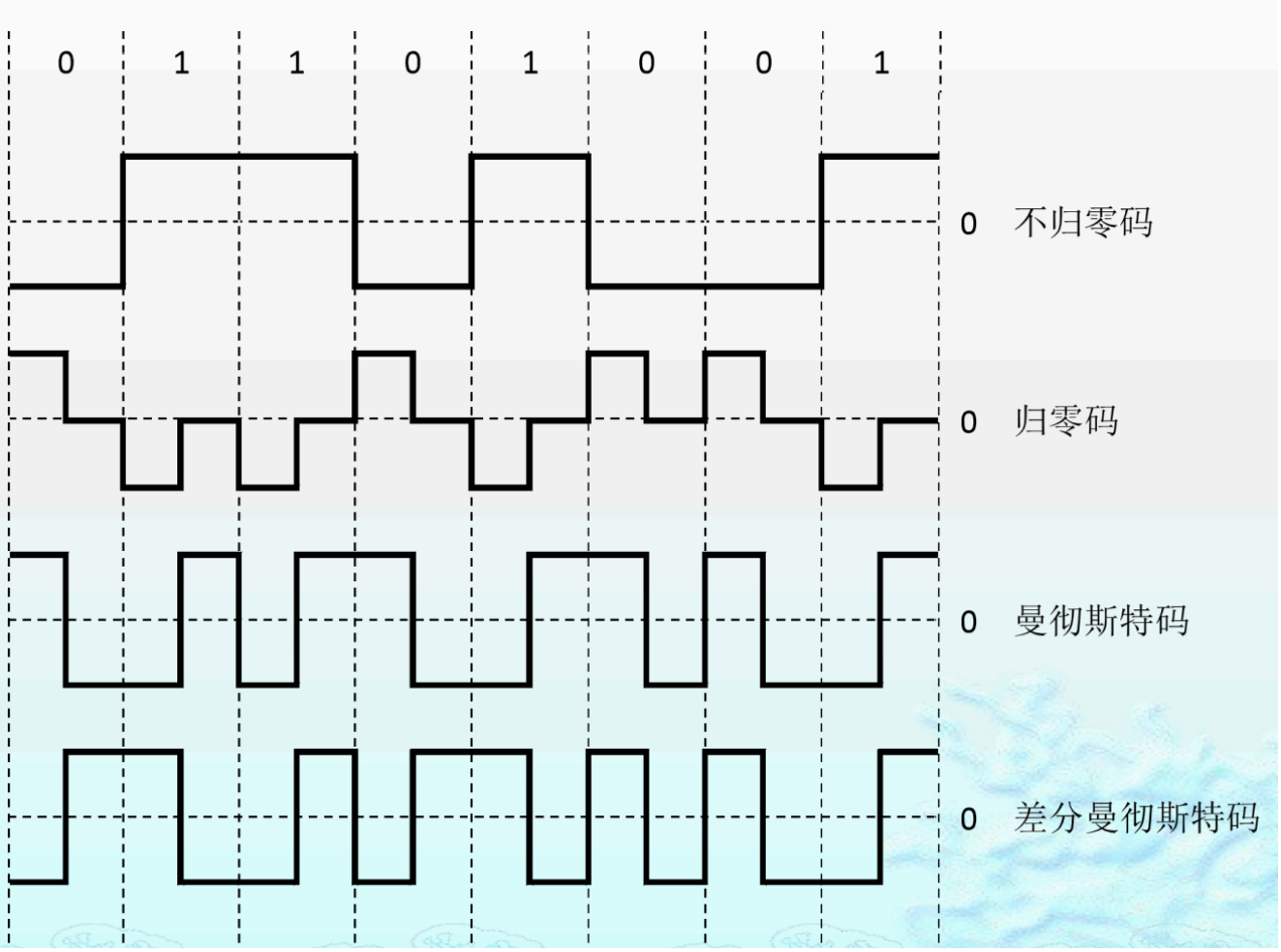
SNR:信(号)噪(声)比

计算中主要用这个。

SNR 才是我们的信噪比，告诉信噪比后还需要计算 $\frac{S}{N}$。

区别于无噪声，这里是没有 **2W** 的。而是直接**W**。

归零码（RZ），不归零码（NRZ），曼彻斯特码，差分曼彻斯特码



- 不归零码： 我觉得这种编码就很适合我这种傻子，不需要脑子的。
- 归零码： 一个bit分两半，后半总归零（名字由来），前半在上是0,前半在下是1。
- 曼彻斯特码：总是看bit中间，上升是1,下降是0
- 差分曼彻斯特： 总是看bit左边沿，但是这样上图的第一个bit是怎么看出来的。

这些码估计都是用来适用于某种特定情况的，但是对于我来说，第一个就很不错。

采样，量化，编码

采样和量化在计算机视觉课上也有接触过。

- 采样是指用数字信号去模拟出模拟信号。并且有一个奇怪的定理是采样频率至少得是信号频率的两倍才能还原。
- 量化是指对一些数字信号进行Normalization。比如0.9,1.8,8.7可能被量化为1,2,9。看规则，一般是根据精度截断，这里似乎又会扯到失真，但是我以前接触的那个和这个可能不是一个概念，后面碰到再说。
- 编码就是根据一定规则编成二进制。

$100BaseT$

- 基带传输(Base): 传输原始信号
- 频带传输: 传输高频信号

100应该是带宽，T是双绞线传输介质。

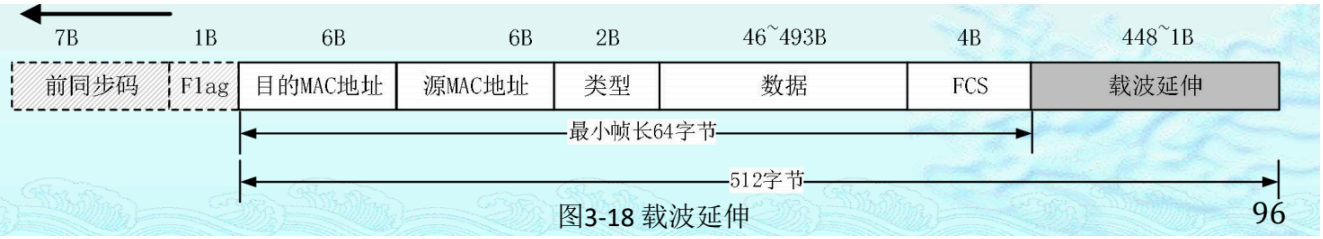
时间仓促，这样就够。

> **Note**

> $10Base2$
>
> 这个2似乎是通州电缆的粗细

3.数据链路层

MAC帧形式



IEEE 802.3的帧格式最小帧长度

每次封装MAC帧一般只管那64字节。其余的似乎并不在考虑。

为什么要封装MAC帧:

- 防止弄混接收人和发送人
- 用于校验，防止数据错乱或者丢失
- 用于标准化

为什么最小帧长度是64字节:

和 `CSMA/CD` 有关。**【Carrier Sense Multiple Access with Collision Detection/载波监听多路访问/碰撞检】**

这里要扯到全双工，半双工。

全双工：高速公路，有双向车道，有隔离带分割。相互间互不干扰。

物理实现： 对于任何一方来说都有两对双绞线或者两条光纤，一对发，一对收，但是在双绞线或者光纤内的信息流向是固定的。因为数据流向固定，所以不存在碰撞风险，不需要 `CSMA/CD` 进行碰撞检测。

半双工：只有单向车道，固定首发，收的人不能发，发的人不能收，设有碰撞检测。

物理实现：共享信道，同一时间只能有一个方向的数据流动， `CSMA/CD` 规则用于检测碰撞。

帧长度最小规定:

为了碰撞检测，检测到后会~~终止发送并且尝试接受或者重新发送。~~强化冲突。

理由，又得扯到之前的发送时延，在数据发送完毕之前，发生碰撞是可以被检测到的。当发送出去后（发送时延），就不会再管先前发送的信息了。

> **Note**
>
> **为什么可以在发送的时候检测到碰撞:**
>
> 冲突会导致电平叠加为原来两倍，电缆可以检测到。
>
> **强化冲突**
>
> 发送32比特或48比特的人为干扰信号（Jammingsignal）
>
> **指数退避**
>
> $0, 1, 3, 7, 15...$
>
> 冲突n次，从n个中选择一个k等待:
>
> 等待时间 $= k \times$ 往返时间（争用时间$or RTT$）

> 强化后立刻退避。

发送时延 $\geq 2 \times$ 传播时延

至于为什么是`X2`，这里似乎和各种情况的考量有关。这里时间不够就先跳过了。

如果用到这个公式，需要确认，发送数据块的单位一般是(B),而带宽通常是bps。

$1Byte = 8bit$

- 10BASE-T只支持半双工(CSMA/CD)
- 100/1000Mbps支持半双工(CSMA/CD)和全双工（不使用CSMA/CD）
- 10000Mbps及以上，仅支持全双工

以及这里有一个补充。

似乎半双工也仅限于数据传输速率低的时候。大概是高了之后发送时延跟不上了?

因为之前看过，最大传输速率和信道带宽（ **W** ）有关，那个是频带宽度，而发送和发送频率（也叫带宽）有关，可能是发送频率跟不上最大传输速率，发展，导致堵塞或者等的时间不断变长（最短帧不断变长），效率低了所以就取消半双工。

另外，碰撞和监听也并不只有`CSMA/CD`。

`CSMA/CD`并不适用于无线通信。且无线不会冲突，因为信道也共用。

另外有一个`CDMA`,是无线的，不要搞混。

无线网络

发送AP和接收AP格式:

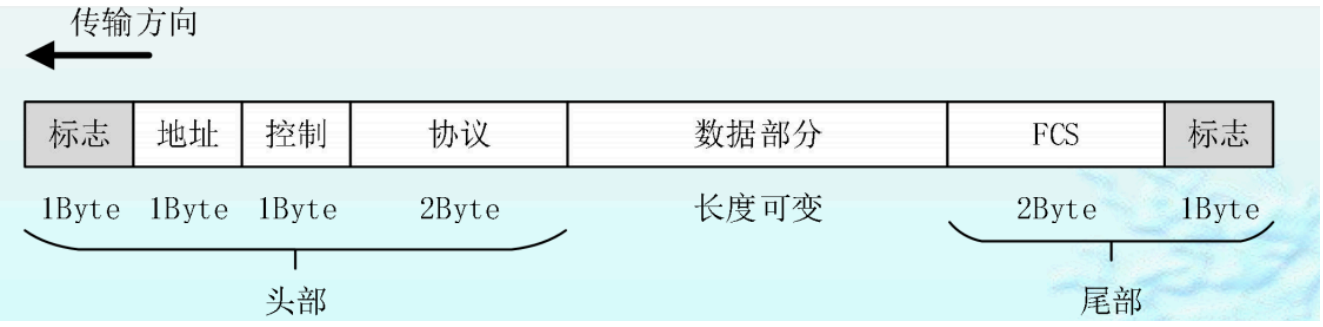
发送/接收	地址1	地址2	地址3
（源）发送AP	AP地址	源地址	第三方地址（接收地址）
（源）接收AP	源地址	AP地址	第三方地址（发送地址）

FLAG

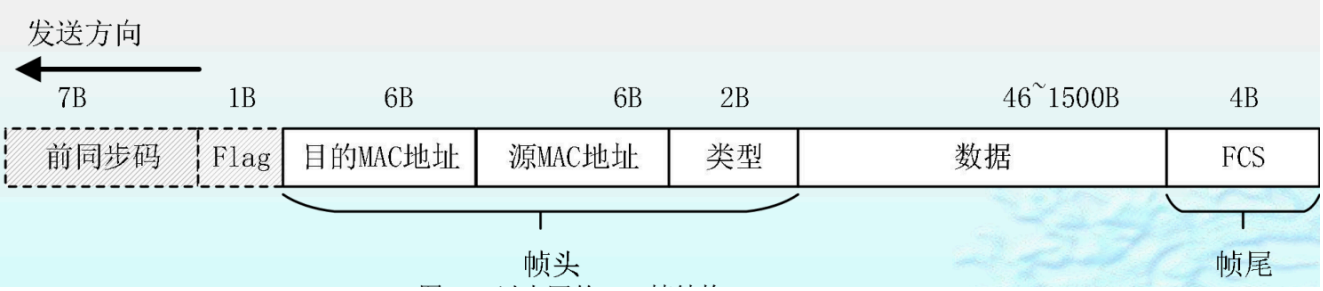
字符填充，`SOH`,`EOT`,本身是一个字符，控制字符，不可打印。

位填充，`01111110`，如果数据中存在相同的字符串，那么每5个连续的1后面插入一个0。解释的时候，每5个连续1后面去掉一个0。

PPP帧结构



MAC帧结构



MAC帧的部分主要是为了校验，和规范化。

MAC地址分成单，多，广（全为1）。

传统局域网

与传统以太网相比，交换式以太网具有以下特点:

- 隔离了冲突域，增大了网络跨距。
- 处于一个广播域，可能产生广播风暴。

冲突域和广播域

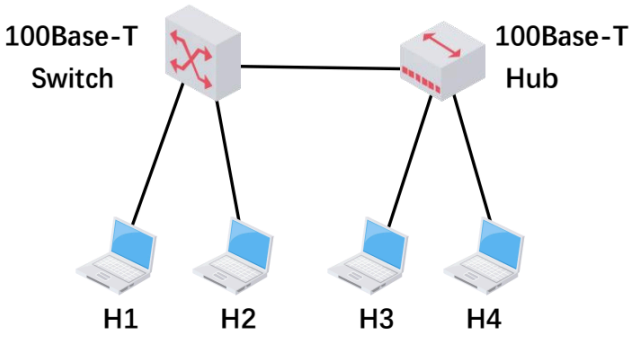
路由器隔离广播域。（不同于路由器）

交换机隔离冲突域。（划分信道）

> **Note**
>
> 转发时，无缓存，先广播（发送者除外）。
>
> 确认时，直接回发送者。

【2016年 题35】若下图中的主机H2向主机H4发送1个数据帧，主机H4向主机H2立即发送一个确认帧，则除H4外，从物理层上能够收到该确认帧的主机还有（　）。

A. 仅H2　　B. 仅H3　　C. 仅H1、H2　　D. 仅H2、H3



回去时，只回发送者，所以H1收不到。

而Hub(集线器)比交换机蠢一点。

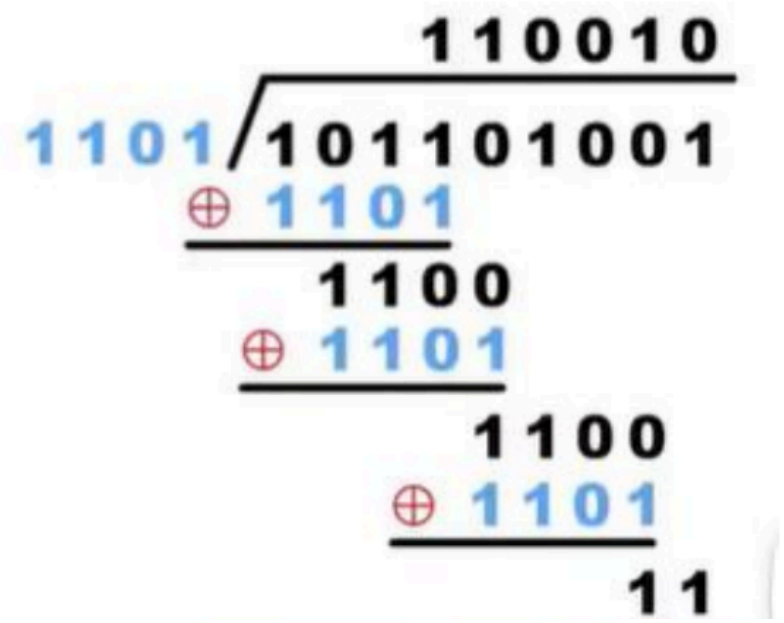
集线器不隔离（共用信道）

LAN

WAN

CRC校验

二进制除法：从高位向低位做xor

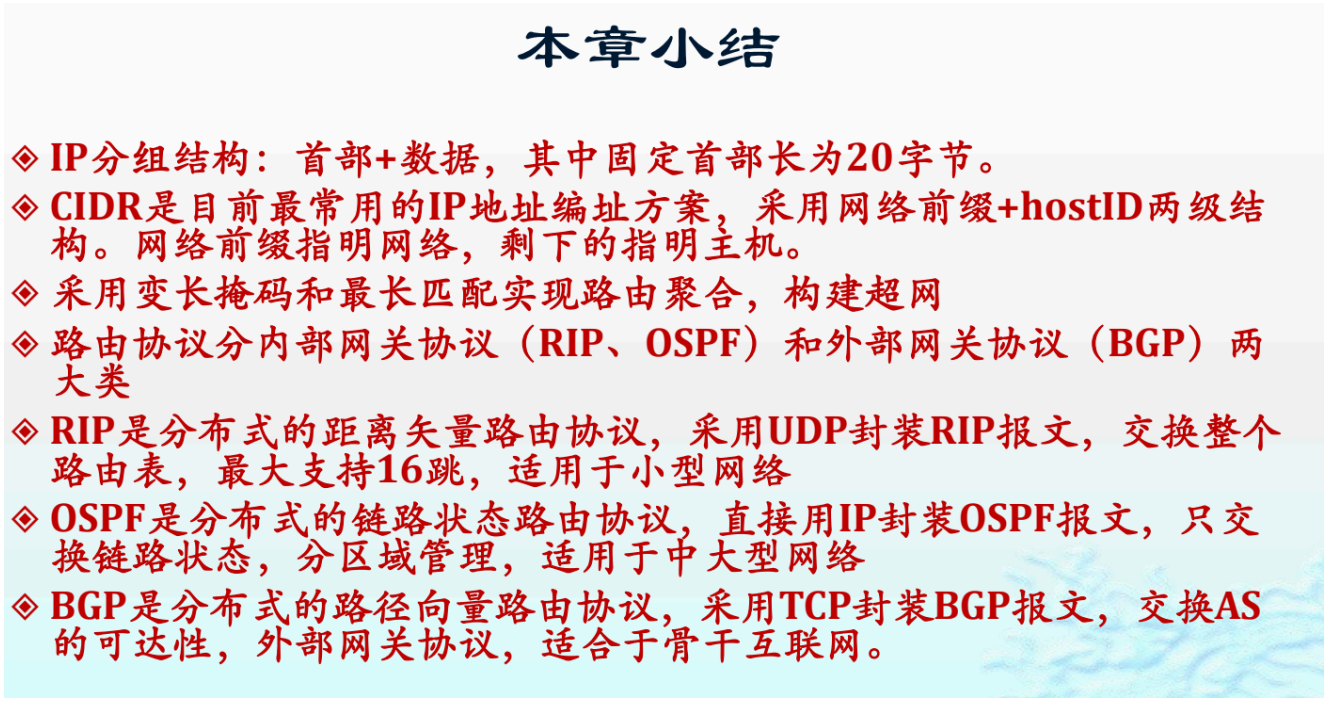


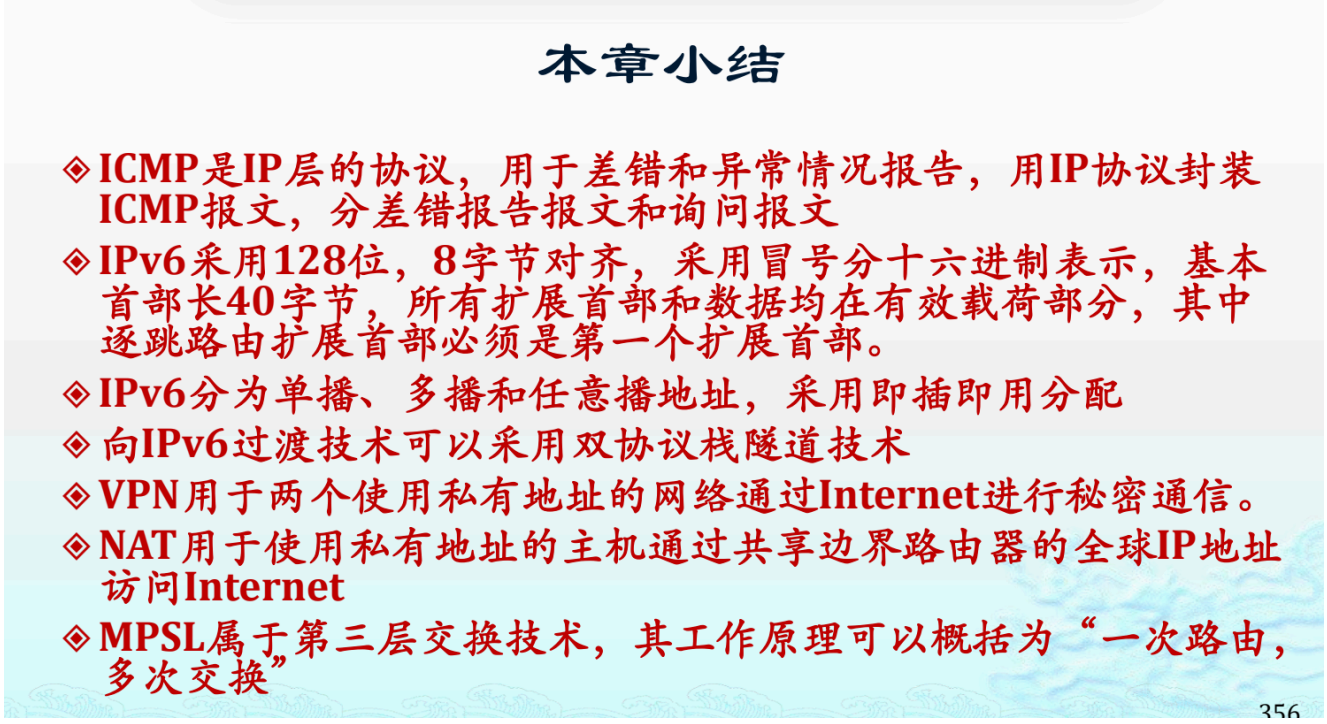
余数为0,则没有误码。

4.网络层

本章小结

- 网络层提供无连接的、不可靠的数据报服务
- 网络层的主要任务是实现网络互连和路由选择，IP实现网络互连，路由协议实现路由选择。
- 分类的全球IP地址分为A、B、C三类，每类IP地址由net_id和host_id组成。D类用于多播，E类保留。
- 子网划分是将host_id的前若干位借来当subnet_id，并没有改变net_id的位数。
- 子网掩码用于查找网络地址，包括子网地址
- IP地址用于标识一个接口而不是一台设备。
- 网络层及以上使用IP地址，链路层及以下用MAC地址
- IP地址到MAC地址的映射由ARP协议完成





子网起点终点

- 0 不用， 1 不用
- 确定分子网的位数
- 32位，确定起点终点。

示例 `168.16.80.24/20`：

转换32位二进制数 -> 后十二位置0确定主机端口 -> 后十二位置1确定广播端口。

可用端口号起点~终点为排除了0和1的所有端口。

子网和子网掩码和默认网关

`/20` 有12位掩码：

`255.255.11110000.0000`

`255.255.240.0`

而默认网关通常作为第一个可以使用的IP地址。（不包含0）

但是并不一定，只要是可用IP（排除0和1），都可以作为默认网关。

`IPv4` 和 `IPv6`

`IPv4` 首部固定20字节。

`IPv6` 首部长度不固定。

`ICMP`

ICMP（Internet Control Message Protocol）

ICMP 是互联网控制消息协议，它是TCP/IP协议族中的一个重要组成部分，主要用于在IP网络中发送错误报告、控制信息和网络诊断信息。以下是ICMP的一些关键点：

- **目的**：
  - 报告网络中的错误情况，例如目标不可达、超时、参数问题等。
  - 提供网络诊断功能，例如Ping命令就是利用ICMP的Echo Request和Echo Reply消息来测试网络连通性。
  - 流量控制和拥塞控制，例如源抑制消息可以通知发送方减慢发送速率。
- **常见ICMP消息类型**：
  - **Echo Request (Type 8)** 和 **Echo Reply (Type 0)**：用于Ping操作。
  - **Destination Unreachable (Type 3)**：表示数据包无法到达目的地，包含多种不可达原因。
  - **Time Exceeded (Type 11)**：通常在数据包的TTL（生存时间）减少到0时发送。
  - **Redirect (Type 5)**：告知发送方应使用不同的路由。
  - **Source Quench (Type 4)**：请求发送方减慢发送速率，已不常用。

报告错误，规范化参数，控制拥塞。

不可达（404），超时（403）？似乎和这类有关。

`RIP`

RIP（Routing Information Protocol）

RIP 是路由信息协议，是一种用于小型网络的动态路由协议。以下是RIP的主要特点：

- **工作机制**：
  - RIP使用距离矢量路由算法，基于跳数（hop count）来决定路由路径，其中每个跳数代表一个路由器。
  - RIP每隔30秒广播一次整个路由表给邻居路由器。
  - 最大跳数限制为15，任何超过15跳的路径被视为不可达。

`traceroute`、`tracert`。

限制跳数，和决定跳的方向，广播路由表。

5.运输层

TCP 和 UDP

联想一下http和https。

- TCP更复杂：

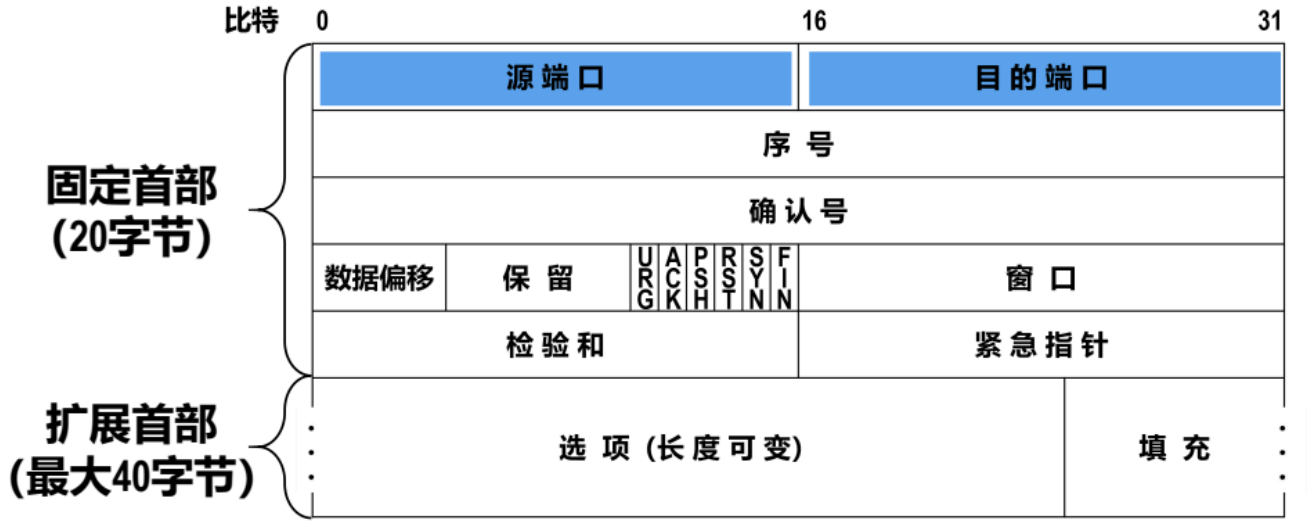
  确认机制、超时重传、流量控制、以及拥塞控制
- UDP更简单。

http基于TCP。

https基于TCP+SSL。

用户数据报协议UDP (User Datagram Protocol)	传输控制协议TCP (Transmission Control Protocol)
☐ 无连接	☐ 面向连接
☐ 支持“一对一”、“一对多”、“多对一”和“多对多”交互通信。	☐ 每一条TCP连接只能有两个端点EP，只能是一对一通信。
☐ 面向应用报文	☐ 面向字节流
☐ 尽最大努力交付，即不可靠；不使用流量控制和拥塞控制。	☐ 可靠传输，使用流量控制和拥塞控制。
☐ 首部开销小，仅8字节。	☐ 首部最小20字节，最大60字节。

TCP报文头部



确认序号和序号

- 确认序号是对方下次要发的首部字节序号
- 序号等于对方的确认序号否则就是中间就发送丢失
- 对于各自来说，发送的字节序号上是连续的（你连续你的，我连续我的）

【2009年 题38】主机甲与主机乙之间已建立一个TCP连接，主机甲向主机乙发送了两个连续的TCP段，分别包含300字节和500字节的有效载荷，第一个段的序列号为200，主机乙正确接收到两个段后，发送给主机甲的确认序列号是（D）。

A. 500　　B. 700　　C. 800　　D. 1000

200 – 499 – 500 – 999

确认序号1000,代表下次甲应该发来这个。

【2013年 题39】主机甲与主机乙之间已建立一个TCP连接，双方持续有数据传输，且数据无差错与丢失。若甲收到1个来自乙的TCP段，该段的序号是1913、确认序号为2046、有效载荷为100字节，则甲立即发送给乙的TCP段的序号和确认序号分别是（B）。

A. 2046、2012　　B. 2046、2013　　C. 2047、2012　　D. 2047、2013

各自连续。

乙：1913(确认2046) – 2012

甲:2046(确认2013) – 甲末尾序号

乙:2013(甲末尾序号 + 1) − 乙末尾序号

【2011年 题40】主机甲与主机乙之间已建立一个TCP连接，主机甲向主机乙发送了3个连续的TCP段，分别包含300字节、400字节和500字节的有效载荷，第3个段的序号为900，若主机乙仅正确接收到第1和第3个段，则主机乙发送给主机甲的确认序号是（B）。

A. 300　　B. 500　　C. 1200　　D. 1400

确认序号一旦对不上，后续不再确认。

是被迫的，因为不知道对方结束点，就不能发确认序号，那么就不知道后续在哪里开始，就不能组成报文头部。

seq、ack

seq实际上就是序列长度

$ack = seq + 1$实际上就是确认序号

ACK

ACK正常情况总为1,表示确认。

URG

`URG=1` 则紧急立刻发送

RST

`Reset`

SYN

`SYN=1` 表示请求建立连接

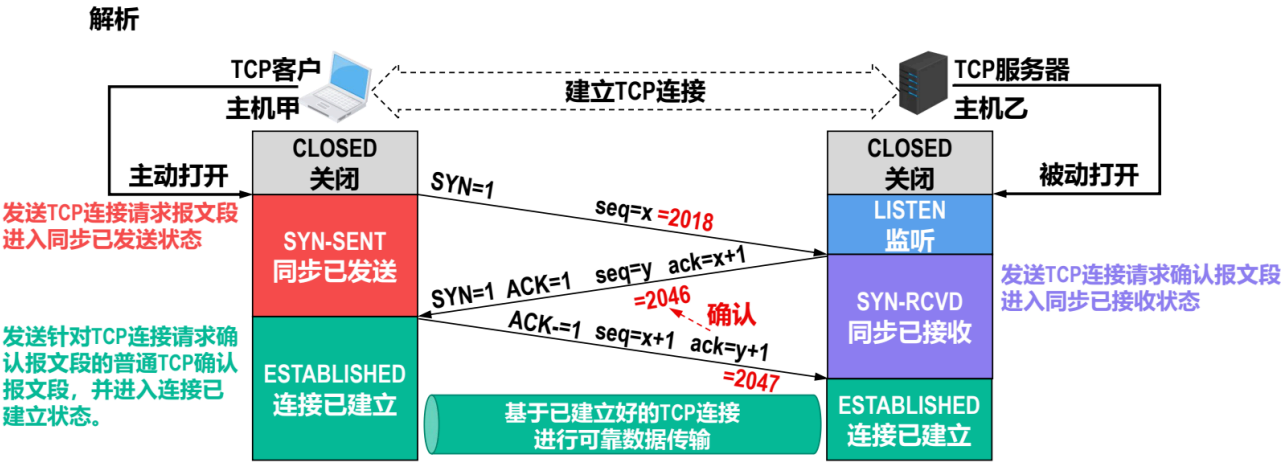
FIN

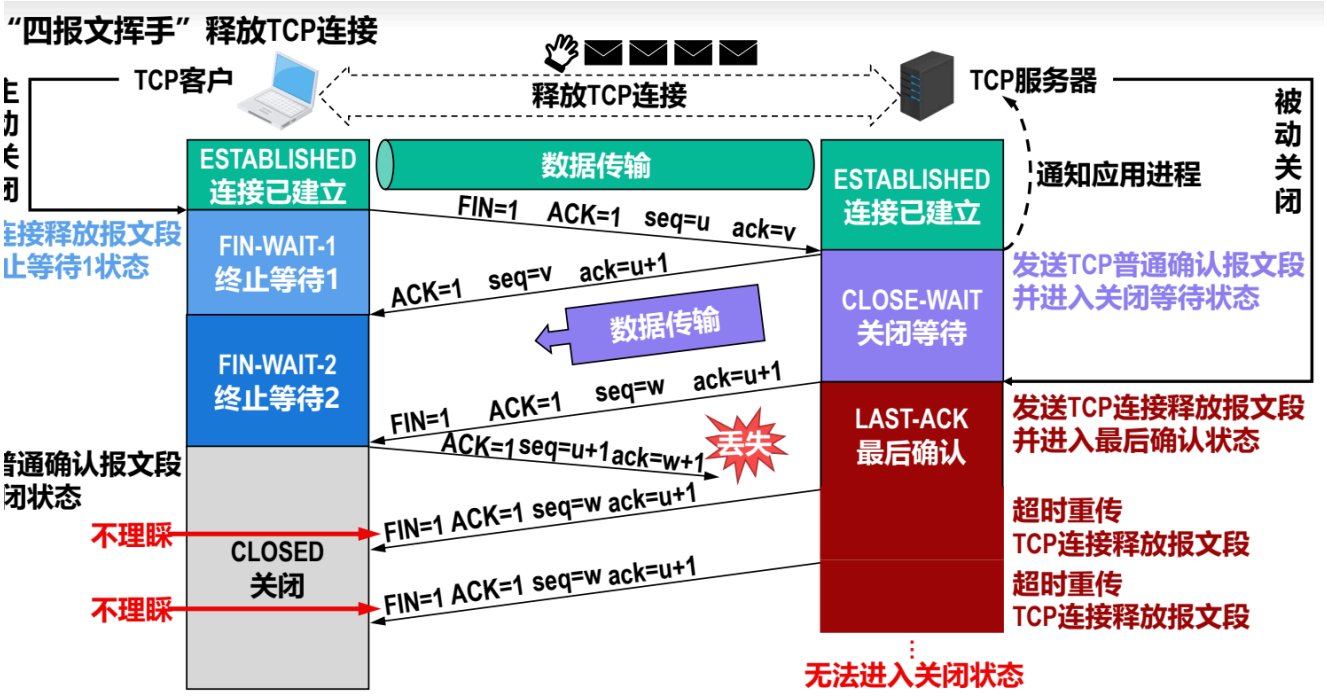
`Final`,结束连接

TCP三报文握手（建立连接）和四报文握手（断开连接）

总是一个发，一个立刻确认，然后接着发，接着确认。

只有在第一次发送时，$ACK = 0, SYN = 1$





回退N帧、选择重传、停止等待

都具有一个特点：

一旦确认序号对不上序号，不发送ACK。

回退N帧：一旦对不上，后续不再ACK

窗口前移，到缺失地方重新发送。

具有累积ACK的特点，没有ACK不一定没有收到，比如0,2,3有ACK,那么1也是被确认的，不然就不会有2,3。另外终点也会ACK,可以确认终点是3。

选择重传：对不上，后续接着ACK

重发缺失ACK的部分。

接收到一个立刻发一个ACK。

停止等待：发一个，等一下，没收到ACK就不接着发。

超时重传。

流量控制

【2010年 题39】主机甲和主机乙之间建立了一个TCP连接，TCP最大段长度为1000字节。若主机甲的当前拥塞窗口为4000字节，在主机甲向主机乙连续发送两个最大段后，成功收到主机乙发送的第一个段的确认段，确认段中通告的接收窗口大小为2000字节，则此时主机甲还可以向主机乙发送的最大字节数是（A）。

A. 1000　　B. 2000　　C. 3000　　D. 4000

解析

TCP发送方的发送窗口值 = min[TCP发送方的拥塞窗口值，TCP接收方的接收窗口值]

发送窗口值 $=\min\{\text{拥塞窗口值}, \text{接收窗口值}\}$

拥塞控制

请求超时断开连接实际上就是一种拥塞控制的体现。

6.应用层

很多在看《网络是怎样连接的》的时候其实已经看过了，所以这里就不看了。

DHCP 服务器

记录物理地址对应的IP地址。

IP端口可以手动设置，也可以请求DHCP配置。

如果手动设置的话，不需要经过DHCP。

但是DHCP在被请求和进行分配的时候，会尝试广播和确认目标IP地址是否已经有主机占用。

DHCP 报文类型

Discover

Offer

Request

ACK

7.程序设计题:

MAC帧封装:

前导码 – 帧定界符 – 目的地址 – 源地址 – 长度字段(数据字段长度) – 数据字段（46～1500） – 校验字段（*CRC*）

```
D5 55 55 55 55 55 55 D5 00 11 22 33 44 55 AA BB CC DD EE FF 00 2E
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 01 23 45 67
```

通常而言长这样。一个2位16进制数代表一个字节。

```
struct MacFrame
{
    unsigned char ucArrPrecode[7]; //7字节前导码
    unsigned char ucBoundary; //帧定界 1字节
    unsigned char UcArrDecAdd[6]; //目的地址
    unsigned char UcArrSourceAdd[6]; //源地址
    unsigned short nLen; //长度 46~1500
    unsigned char* ucArrData; //不定长的数据块
    unsigned char CRC; //一字节的校验码
}
// 去他妈妈的葡萄牙命名法
void FillMacFrame(
    MacFrame* pMacFrame,
    unsigned char* ucArrDecAdd,
```

```
    unsigned char* ucArrSourceAdd,
    unsigned short nLen,
    unsigned char* pData,
    unsigned char CRC,
)
{
    for(int i=0;i<7;i++)
        pMacFrame.ucArraryPrecode[i] = 01010101;
    pMacFrame.ucBoundary = 01010101;
    for(int i=0;i<6;i++)
    {
        pMacFrame.UcArrDecAdd[i]=ucArrDecAdd[i]
        pMacFrame.UcArrSourceAdd[i]=ucArrSourceAdd[i]
    }
    pMacFrame.nLen = (nLen<46)?46:nLen;
    // 用malloc初始化是垃圾值，没有赋值的话是未知
    pMacFrame.ucArrData = malloc(pMacFrame.nLen+64-46)
    for(int i=0;i<sizeof(UcData)&&i<nLen;i++)
    {
        pMacFrame.ucArrData[i]=UcData[i];
    }
    for(int i=sizeof(UcData);i<nLen;i++)
    {
        // Data不足46字节，需要补充0
        pMacFrame.ucArrData[i]=0;
    }
    CalCulateCRC（pMacFrame， CRC）;
}
```

- 前导码56位(7字节)的1010101...1010比特序列组成，帧定界符为1字节，结构为10101011。
- 目的地址和源地址均采用6字节。(48bit)

校验和

```
unsigned short CalculteCheckSum(unsigned short *pSendWordBuff, unsigned int nWordTotal)
{
    unsigned short usCheckSum; //校验和
    unsinged long ulTempCheckSum = 0; //sum
    int i;
    for(i=0;i<nWordTotal;i++) //按照byte进行校验计算，对每个short进行校验
        ulTempCheckSum = ulTempCheckSum  + pSendWordBuff[i]; //sum
    usCheckSum = ~( (ulTempCheckSum  & 0x0000FFFF) + (ulTempCheckSum   >> 16));
    // 低16位+高16位取反得到校验和
    return usCheckSum ;
}
```

Socket编程

```
```cpp
#include <iostream>
#include <cstring>
#include <sys/socket.h>
#include <netinet/ip_icmp.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/time.h>
#include <unistd.h>
#include <netdb.h>
#include <chrono>
```
```

```cpp
// ICMP 校验和函数
unsigned short checksum(void* b, int len) {
    unsigned short* buf = (unsigned short*)b;
    unsigned int sum = 0;
    unsigned short result;
    for (sum = 0; len > 1; len -= 2)
        sum += *buf++;
    if (len == 1)
        sum += *(unsigned char*)buf;
    sum = (sum >> 16) + (sum & 0xFFFF);
    sum += (sum >> 16);
    result = ~sum;
    return result;
}

// 计算时间差（返回毫秒）
long time_diff(struct timeval start, struct timeval end) {
    return (end.tv_sec - start.tv_sec) * 1000 + (end.tv_usec - start.tv_usec) / 1000;
}

// 构建并发送 ICMP Echo Request
void send_icmp(int sockfd, sockaddr_in* addr, int ttl) {
    char packet[64];
    struct icmphdr* icmp_hdr = (struct icmphdr*)packet;
    // 填充 ICMP 头
    memset(packet, 0, sizeof(packet));
    icmp_hdr->type = ICMP_ECHO;
    icmp_hdr->code = 0;
    icmp_hdr->un.echo.id = getpid();
    icmp_hdr->un.echo.sequence = ttl;
    icmp_hdr->checksum = checksum(icmp_hdr, sizeof(packet));
    // 设置 TTL
    setsockopt(sockfd, IPPROTO_IP, IP_TTL, &ttl, sizeof(ttl));
    // 发送数据包
    sendto(sockfd, packet, sizeof(packet), 0, (sockaddr*)addr, sizeof(*addr));
}

// 接收 ICMP Echo Reply 或 Time Exceeded 消息
bool receive_icmp(int sockfd, sockaddr_in* reply_addr, long& rtt) {
    char buffer[1024];
    socklen_t addrlen = sizeof(*reply_addr);
    struct timeval start, end;
    gettimeofday(&start, nullptr);
    int received_bytes = recvfrom(sockfd, buffer, sizeof(buffer), 0, (sockaddr*)reply_addr, &addrlen);
    gettimeofday(&end, nullptr);
    if (received_bytes > 0) {
        struct iphdr* ip_hdr = (struct iphdr*)buffer;
        struct icmphdr* icmp_hdr = (struct icmphdr*)(buffer + (ip_hdr->ihl * 4));
        rtt = time_diff(start, end);
        // 判断 ICMP 类型，是否为 Echo Reply 或 TTL 超时
        if (icmp_hdr->type == ICMP_ECHOREPLY) {
            return true; // 收到目标的回复
        } else if (icmp_hdr->type == ICMP_TIME_EXCEEDED) {
            return true; // 收到中间路由器的 TTL 超时
        }
    }
    return false;
}

int main(int argc, char* argv[]) {
    if (argc != 2) {
        std::cerr << "Usage: tracert <IP Address/Host Name>" << std::endl;
        return 1;
    }
```

```
    struct addrinfo hints, *res;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family = AF_INET; // 使用 IPv4
    // 解析目标主机名或 IP 地址
    if (getaddrinfo(argv[1], nullptr, &hints, &res) != 0) {
        std::cerr << "Error resolving hostname" << std::endl;
        return 1;
    }

    sockaddr_in dest_addr = *(sockaddr_in*)res->ai_addr;
    char ip_str[INET_ADDRSTRLEN];
    inet_ntop(AF_INET, &(dest_addr.sin_addr), ip_str, INET_ADDRSTRLEN);
    std::cout << "Tracing route to " << argv[1] << " [" << ip_str << "]" << std::endl;
    std::cout << "Over a maximum of 30 hops:\n" << std::endl;

    // 创建原始套接字
    int sockfd = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP);
    if (sockfd < 0) {
        std::cerr << "Error creating socket" << std::endl;
        return 1;
    }

    // 设置接收超时为 3 秒
    struct timeval timeout;
    timeout.tv_sec = 3;
    timeout.tv_usec = 0;
    setsockopt(sockfd, SOL_SOCKET, SO_RCVTIMEO, &timeout, sizeof(timeout));

    const int max_hops = 30;
    for (int ttl = 1; ttl <= max_hops; ++ttl) {
        std::cout << "Sending ICMP packet with TTL = " << ttl << std::endl;
        send_icmp(sockfd, &dest_addr, ttl);
        sockaddr_in reply_addr;
        long rtt = 0;
        bool success = receive_icmp(sockfd, &reply_addr, rtt);
        if (success) {
            char reply_ip[INET_ADDRSTRLEN];
            inet_ntop(AF_INET, &(reply_addr.sin_addr), reply_ip, INET_ADDRSTRLEN);
            std::cout << ttl << " " << rtt << "ms " << reply_ip << std::endl;
            // 如果到达目标主机，则完成
            if (reply_addr.sin_addr.s_addr == dest_addr.sin_addr.s_addr) {
                break;
            }
        } else {
            std::cout << ttl << " * * * Request timed out." << std::endl;
        }
    }

    close(sockfd);
    return 0;
}
```

`socket()` 创建套接字:

```
//TCP
int tcp_socket = socket(AF_INET, SOCK_STREAM, 0);
if (tcp_socket == -1) {
    perror("socket creation failed");
    // 处理错误
}

//UDP
int tcp_socket = socket(AF_INET, SOCK_DGRAM, 0);
if (tcp_socket == -1) {
```

```
    perror("socket creation failed");
    // 处理错误
}

//ICMP
int tcp_socket = socket(AF_INET, SOCK_RAW, 0);
if (tcp_socket == -1) {
    perror("socket creation failed");
    // 处理错误
}
```