

## § 3. 数据链路层

### 3.1. 数据链路层的设计问题

#### 3.1.1. 提供给网络层的服务

#### ★ 数据链路层在五层模型中的位置

应用层
传输层
网络层
数据链路层
物理层

#### ★ 虚拟通信与实际通信

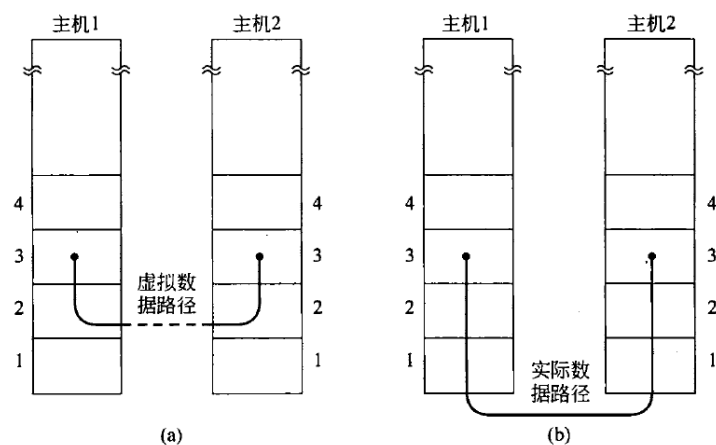
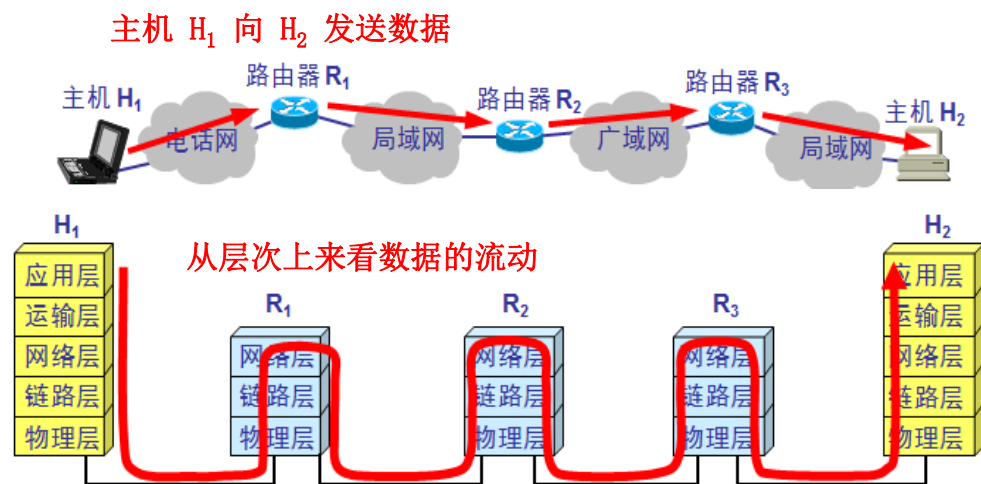
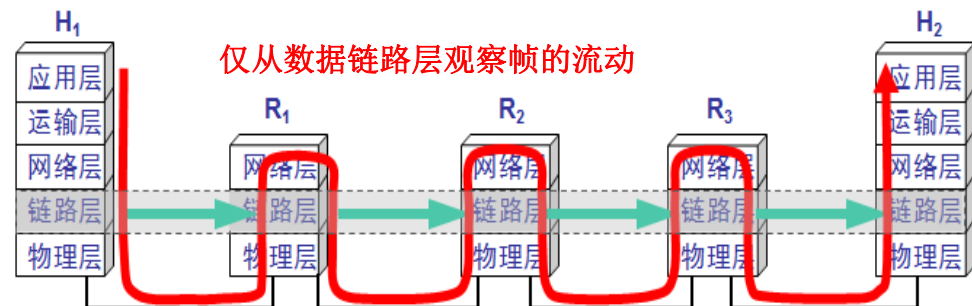


图 3-2  
(a) 虚拟通信; (b) 实际通信



从层次上来看数据的流动



仅从数据链路层观察帧的流动

## § 3. 数据链路层

### 3.1. 数据链路层的设计问题

#### 3.1.1. 提供给网络层的服务

##### ★ 数据链路层的作用

在两台相邻机器上实现可靠有效的完整信息块通信

##### ★ 数据链路层的信道类型

- 点对点信道：使用一对一的点对点通信方式
- 广播信道：使用一对多的广播通信方式，因此过程比较复杂。广播信道上连接的主机很多，因此必须使用专用的共享信道协议来协调这些主机的数据收发

##### ★ 链路和数据链路

- 链路(link)：一条无源的点到点的物理线路段，中间没有任何其他的交换结点  
=> 一条链路只是一条通路的一个组成部分

- 数据链路(data link)：除物理线路外，还必须有通信协议来控制这些数据的传输。若把实现这些协议的硬件和软件加到链路上，就构成了数据链路

=> 现在常用(不是唯一)的方法是使用适配器(网卡)来实现这些协议的硬件和软件。一般的适配器都包括了数据链路层和物理层这两层的功能

## § 3. 数据链路层

### 3.1. 数据链路层的设计问题

#### 3.1.1. 提供给网络层的服务

##### ★ 帧：数据链路层传输的基本单位

- 可以理解为在两个对等的的数据链路层之间存在一条数字管道，而在这条数字管道上传输的数据单位是帧



- 早期的数据通信协议也称通信规程(procedure)，因此在数据链路层，规程 = 协议

##### ★ 数据包和帧的关系

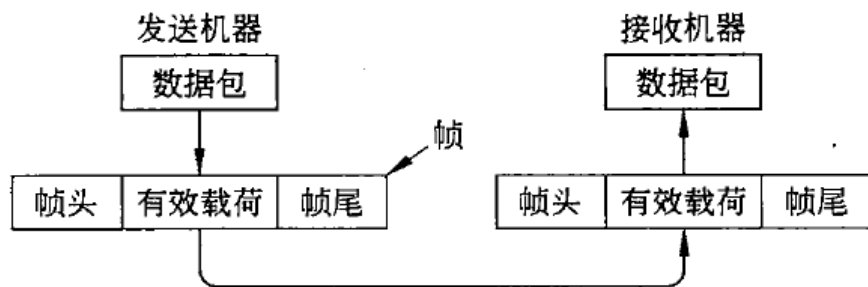
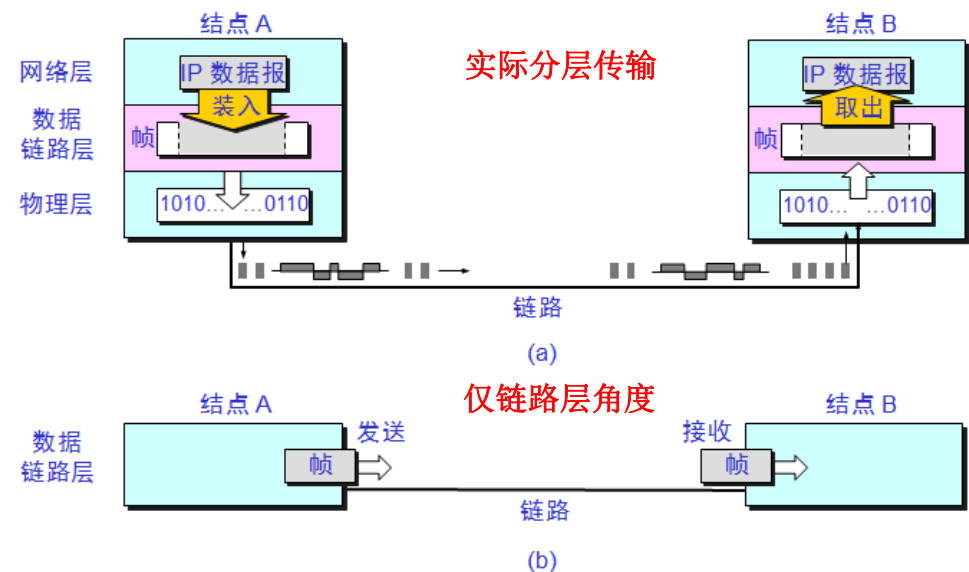


图 3-1 数据包和帧的关系

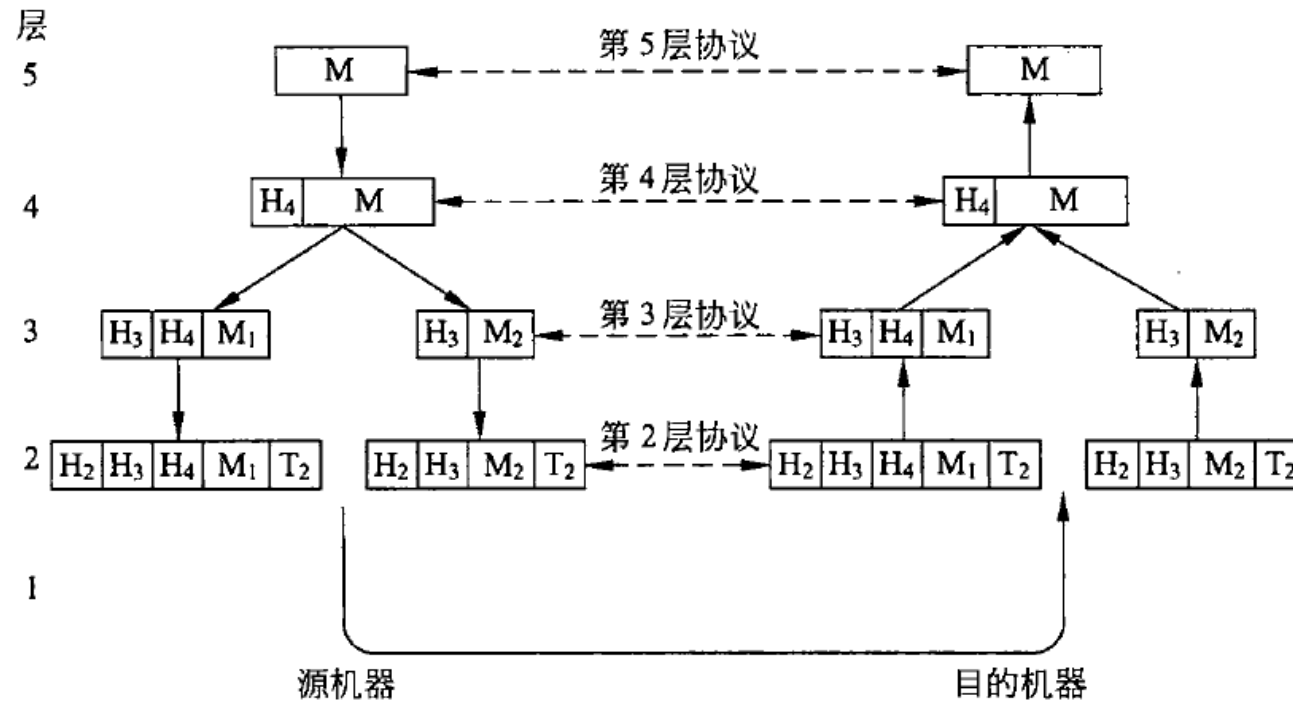


## § 3. 数据链路层

### 3.1.1. 提供给网络层的服务

#### ★ 数据包和帧的关系

- 一个完整的上层数据包可能因为长度的限制而拆分为若干帧 (第1章 P. 25 图1-15)



## § 3. 数据链路层

### 3.1. 数据链路层的设计问题

#### 3.1.1. 提供给网络层的服务

##### ★ 向网络层提供的三种基本服务

- 无确认的无连接服务：目的节点收到正确的帧后不进行确认，因此无须建立和释放连接，数据链路层不检测丢帧，丢失帧由上层恢复  
=> 适用于低误码率场合、实时系统
- 有确认的无连接服务：目的节点正确收到的帧后发送确认帧，使发送方知道是否已正确到达，无须建立和释放连接，但源节点需配置存放待确认的缓冲区(确认后释放)和计时器(超时重传)  
=> 需要引入帧序号  
=> 相比于上层的确认，纠错更快(上层一个包拆分为多帧，如果只丢失部分帧，则上层重传，要全部数据包；而帧重传，只是丢失部分)  
=> 适用于不可靠信道
- 有确认的有连接服务：建立连接，初始化计时器和所需变量，对每帧进行编号后再发送，确保每个帧都被接收而且只接收一次，且按顺序接收，传输完成后断开连接，释放资源  
=> 适用于长距离且不可靠的链路

##### ★ 数据链路层的主要功能

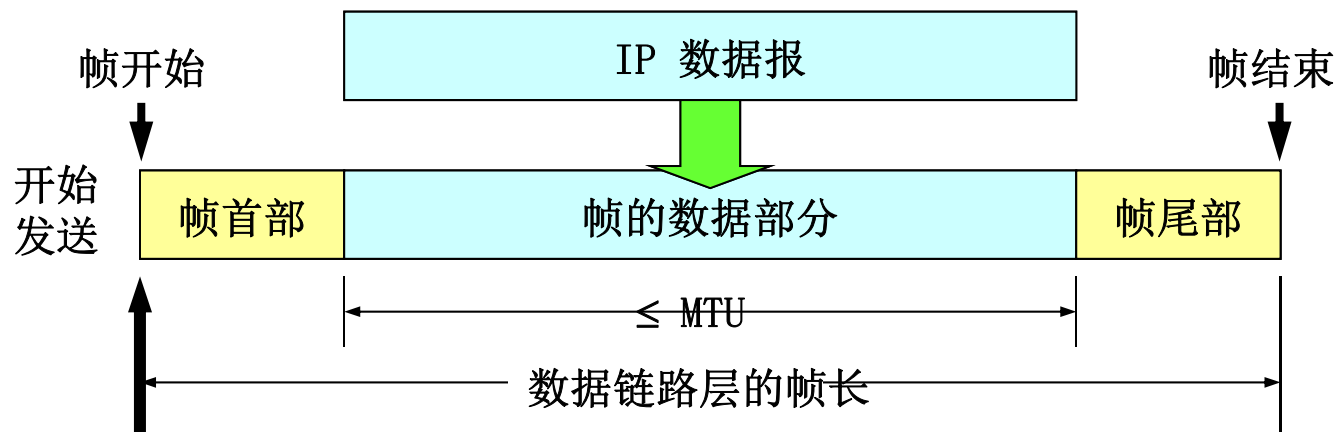
- 向网络层提供一个良好定义的服务接口
- 处理传输错误
- 调节数据流，确保慢速的接收方不会被快速的发送方淹没

## § 3. 数据链路层

### 3.1. 数据链路层的设计问题

#### 3.1.2. 成帧

★ 成帧：封装成帧 (framing) 就是在一段数据的前后分别添加首部和尾部，然后就构成了一个帧



- 首部和尾部的一个重要作用就是进行帧定界
- 帧同步：使发送/接收方均了解一帧的开始/结束

## § 3. 数据链路层

### 3.1. 数据链路层的设计问题

#### 3.1.2. 成帧

★ 字节计数法：每帧的第一个字节表示本帧的字节数

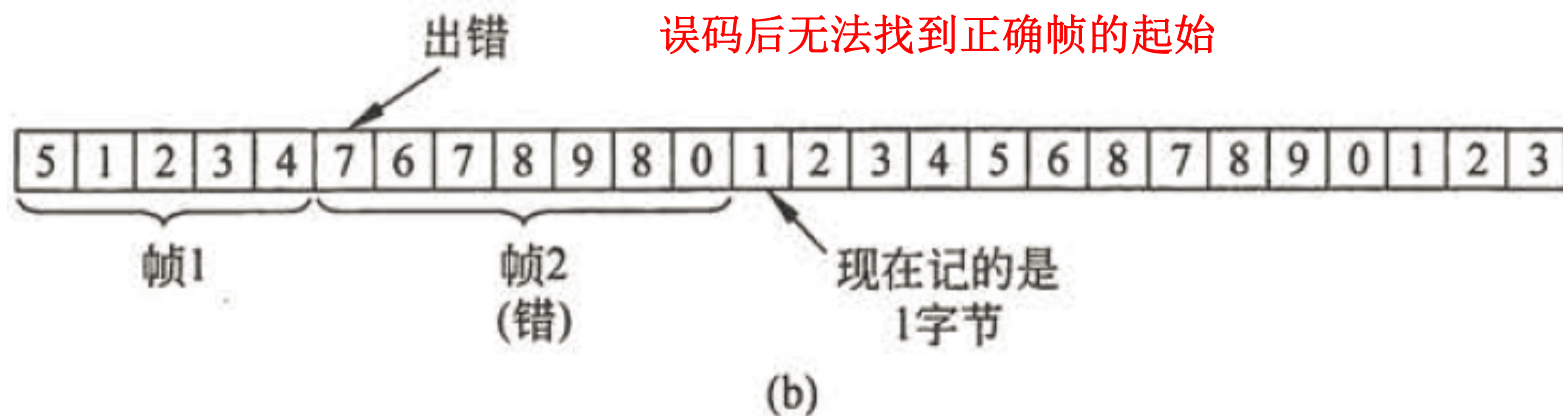
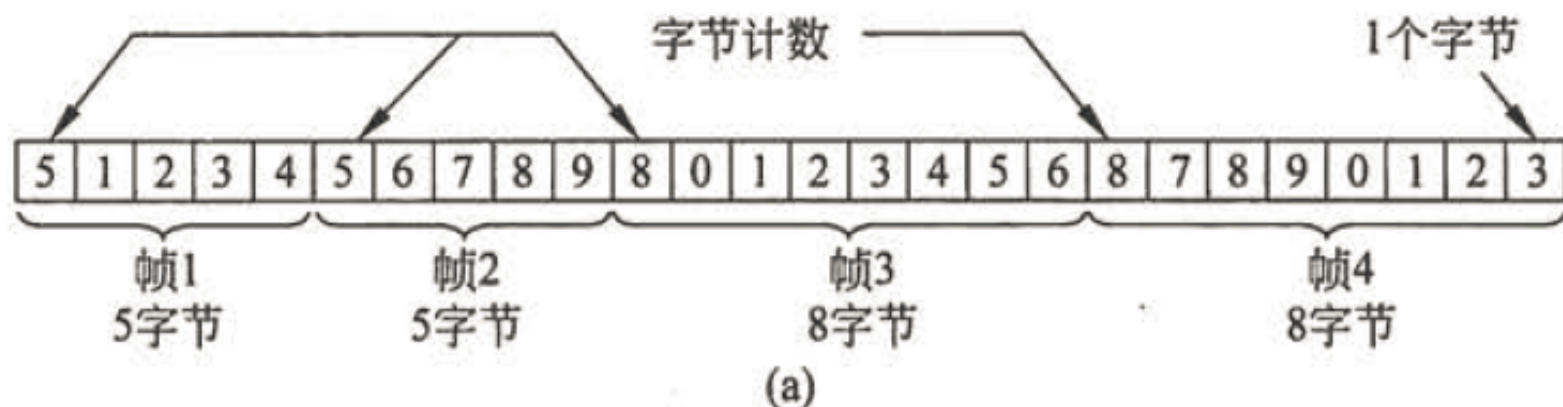


图 3-3 字节流

(a) 没有错误； (b) 有一个错误

## § 3. 数据链路层

### 3.1. 数据链路层的设计问题

#### 3.1.2. 成帧

★ 字节填充的标志字节法：特殊字节做为开始/结束

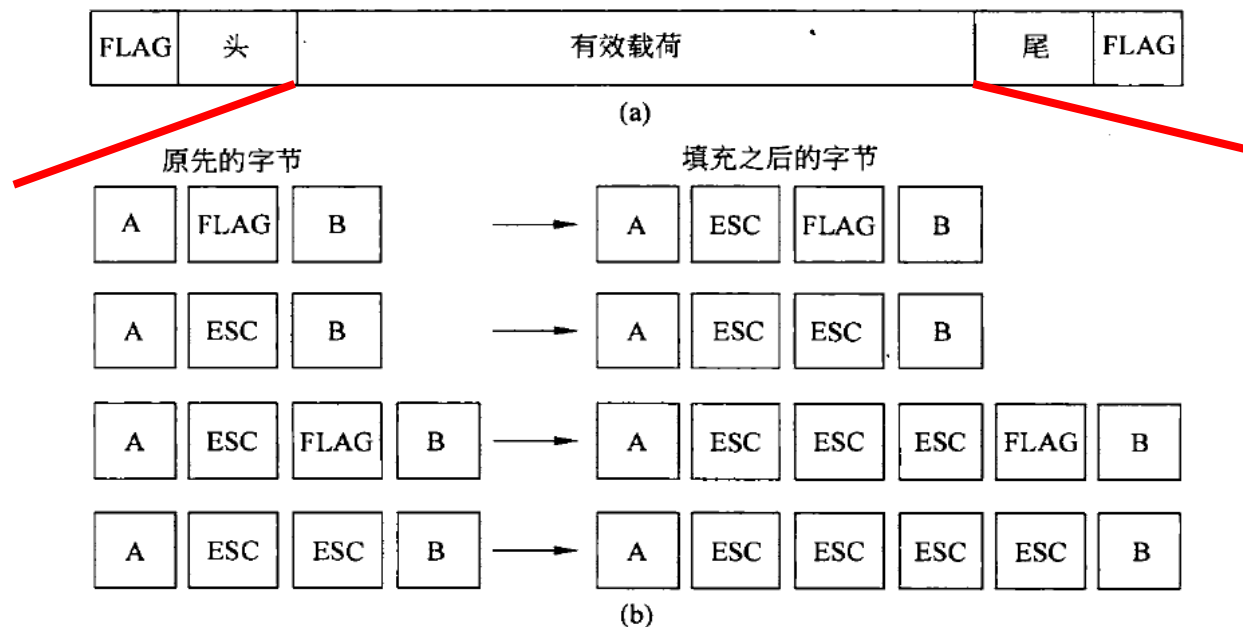


图 3-4

(a) 由标志字节分界的帧； (b) 字节填充之前和之后的字节序列示例

如果特殊字节出现在数据序列中，则需要用转义符填充，转义符自身也要被转义

和C/C++的字符串转义符理解方式相似：

“↔ FLAG : 表示串的开头和结尾

“↔ ESC : 在串中表示 “ 则应为 \“

在串中表示 \ 则应为 \\

在串中表示 \“ 则应为 \\“



## § 3. 数据链路层

### 3.1. 数据链路层的设计问题

#### 3.1.2. 成帧

★ 比特填充的标志比特法：特殊bit序列做开始/结束

0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0

(a)

0 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 1 0 1 0 0 1 0

填充比特

(b)

01111110：标志

数据中连续5个1，则填入0

0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 1 0

(c)

图 3-5 比特填充

(a) 原始数据： (b) 出现在线路上的数据：

(c) 存储在接收方内存中的数据

## § 3. 数据链路层

### 3. 1. 数据链路层的设计问题

#### 3. 1. 2. 成帧

★ 物理层编码违禁法：使用在物理层违例的编码作为定界标志

## § 3. 数据链路层

### 3.1. 数据链路层的设计问题

#### 3.1.3. 差错控制

- ★ 差错：在传输过程中可能会产生比特差错(1 可能会变成 0，而 0 也可能变成 1)
- ★ 误码率(BER = Bit Error Rate)：在一段时间内，传输错误的比特占所传输比特总数的比率  
称为误码率
  - 误码率与信噪比有很大的关系
  - 为了保证数据传输的可靠性，在计算机网络传输数据时，必须采用各种差错检测措施
- ★ 差错种类
  - 随机错、突发错

#### 3.1.4. 流量控制

- ★ 功能
  - 解决高速发送方与低速接收方的同步问题
- ★ 解决方法
  - 基于反馈的流量控制
    - 接收方通知发送方发送速率是否可接受(本章)
  - 基于速率的流量控制
    - 由协议限制发送方可以发送的速率(第五章)

## § 3. 数据链路层

### 3.2. 差错检测和纠正

#### ★ 差错控制编码

纠错码：包含足够多的冗余信息，使接收方能发现错误并推断出正确数据（不需要重传）

检错码：包含冗余信息，使接收方能发现错误，但无法推断出正确数据（需要重传）

★ 编码效率：若码字长 $n$ 位，其中信息位长 $m$ 位，冗余位长 $r$ 位，则编码效率 
$$R = \frac{m}{n} = \frac{m}{m + r}$$

#### ★ 差错控制方式：

##### ● 自动请求重发（ARQ）

- 采用检错码
- 采用双向信道
- 发送ACK或NAK
- 发送方需缓冲区
- 需帧号
- 发送方需计时器

##### ● 前向纠错（FEC）

- 采用纠错码
- 发送方无需缓冲区、计时器和双向信道

##### ● ARQ和FEC结合

## § 3. 数据链路层

### 3.2. 差错检测和纠正

#### 3.2.1. 纠错码

##### ★ 常用种类

海明码

二进制卷积码

里德所罗门码

低密度奇偶校验码

##### ★ 常用编码方法：块码

块码(block code)：将冗余校验信息加入待发送信息中，若干bit分为一组，每组n位总长，其中包含m位数据和r位校验，本组的校验位仅和本组的数据位有关，称为块码，也称为分组码

两种发送方法：

┌ 前m位为数据，后r位为校验，顺序发送  
└ r位校验码做为m个数据位的函数计算得到 => 函数为线性则称为线性码 (计算机效率高)

## § 3. 数据链路层

### 3.2. 差错检测和纠正

#### 3.2.1. 纠错码

##### ★ 海明 (hamming) 距离

一个编码系统中任意两个合法编码 (码字) 间不同的 bit 位数称为两个码字的码距, 整个系统中任意两个码字的最小距离就是该编码系统的码距, 称为海明距离

例1: 某编码系统, 用3个bit位来表示8种不同信息

则: 0-1/2-3/6-7/... 码距: 1

1-2/1-4/3-5/... 码距: 2

0-7/1-6/2-5/... 码距: 3

=> 海明距离: 1

	a2	a1	a0
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

假设在发送/接收过程中发生了错误

001 -> 000 / 011 / 101 : 1 bit位发生错误

001 -> 010 / 111 / 100 : 2 bit位发生错误

001 -> 110 : 3 bit位发生错误

=> 接收时, 任何一位/多位发生变化, 均被认为是合法码字, 因此无法发现任何错误

## § 3. 数据链路层

### 3.2. 差错检测和纠正

#### 3.2.1. 纠错码

##### ★ 海明 (hamming) 距离

一个编码系统中任意两个合法编码 (码字) 间不同的 bit 位数称为两个码字的码距, 整个系统中任意两个码字的最小距离就是该编码系统的码距, 称为海明距离

	a3	a2	a1	a0
0	0	0	0	0
1	1	0	0	1
2	1	0	1	0
3	0	0	1	1
4	1	1	0	0
5	0	1	0	1
6	0	1	1	0
7	1	1	1	1

例2: 用如图所示4个bit位表示8种不同信息

=> 海明距离: 2

假设发送/接收过程中有错

1001 -> 0001 / 1101 / 1011 / 1000 : 1 bit位错误

=> 1bit错, 全部是非法码字, 能判断有错

1001 -> 1010 / 1111 / 0101 / 1100 / 0011 / 0000 : 2 bit位错误

=> 2bit错, 全部是合法码字, 不能判断有错

1001 -> 1110 / 0111 / 0010 / 0100 : 3 bit位错误

=> 3bit错, 全部是非法码字, 能判断有错

1001 -> 0110 : 4 bit位错误

=> 4bit错, 全部是合法码字, 不能判断有错

=> 能判断奇数个错, 不能判断偶数个错

n=4的码字, 假设每位的错误独立, 误码率位 $10^{-4}$ , 则错1/2/3/4bit的概率分别为:  $10^{-4}, 10^{-8}, 10^{-12}, 10^{-16}$  => 能检出1bit错, 就抓住了主要矛盾

## § 3. 数据链路层

### 3.2. 差错检测和纠正

#### 3.2.1. 纠错码

##### ★ 海明 (hamming) 距离

一个编码系统中任意两个合法编码 (码字) 间不同的 bit 位数称为两个码字的码距, 整个系统中任意两个码字的最小距离就是该编码系统的码距, 称为海明距离

例2: 用如图所示4个bit位表示8种不同信息

=> 海明距离: 2

<p>续问1: 能否纠正1 bit的错误? 假设发送/接收过程中有1bit错, 收到的是1011 则发送可能是 1001/1111/0011/1010 =&gt; 无法纠正 (因为四种发送码字的1bit错有重复)</p>					<p>续问3: 多少bit对应8种编码能否做到1bit错误唯一 3bit-&gt;8种: <math>2^3=8</math>, 8种合法编码, 0种非法编码 =&gt; 无法检错+无法纠错</p> <p>4bit-&gt;8种: <math>2^4=16</math>, 8种合法编码, 8种非法编码 =&gt; 每种4bit合法编码错1bit位, 共<math>4*8=32</math>种 =&gt; <math>8&lt;32</math> (8种非法编码表示32种错), 必定有重复 =&gt; 可以检错+无法纠错</p> <p>5bit-&gt;8种: <math>2^5=32</math>, 8种合法编码, 24种非法编码 =&gt; 每种5bit合法编码错1bit位, 共<math>5*8=40</math>种 =&gt; <math>24&lt;40</math>, 必定有重复 =&gt; 可以检错+无法纠错</p> <p>6bit-&gt;8种: <math>2^6=64</math>, 8种合法编码, 56种非法编码 =&gt; 每种6bit合法编码错1bit位, 共<math>6*8=48</math>种 =&gt; <math>56&gt;48</math>, 可做到无重复=&gt;可以检错+可以纠错</p>				
<p>续问2: 要想纠正1 bit的错误, 要满足什么条件? 各种发送码字的1bit错相互没有重复, 则可以纠错 例如: 1011 这个错误码, 只能对应1001这个发送 =&gt; 目前4bit对应8种编码无法做到</p>									
	a3	a2	a1	a0					
0	0	0	0	0					
1	1	0	0	1					
2	1	0	1	0					
3	0	0	1	1					
4	1	1	0	0					
5	0	1	0	1					
6	0	1	1	0					
7	1	1	1	1					



## § 3. 数据链路层

### 3.2. 差错检测和纠正

#### 3.2.1. 纠错码

##### ★ 海明 (hamming) 距离

一个编码系统中任意两个合法编码(码字)间不同的bit位数称为两个码字的码距，整个系统中任意两个码字的最小距离就是该编码系统的码距，称为海明距离

	a5	a4	a3	a2	a1	a0
0	0	0	0	0	0	0
1	0	1	1	0	0	1
2	1	0	1	0	1	0
3	1	1	0	0	1	1
4	1	1	0	1	0	0
5	1	0	1	1	0	1
6	0	1	1	1	1	0
7	0	0	0	1	1	1

例3: 用如图所示6个bit位表示8种不同信息

=> 海明距离: 3

假设发送/接收过程中有错

011001 -> 011000 / 011011 / 011101 / 010001 / 001001 / 111001

=> 1bit错，均为非法码字

=> 可判断出错 (多bit不考虑)

同理: 000000等7种合法编码，每种错1bit位有6种情况，共42种，且与011001的6种非法码字不同

=> 收到 011000 / 011011 / 011101 / 010001 / 001001 / 111001 这六种均认为是收到了 011001

=> 纠错成功

结论: 海明距离至少要等于3，才能纠错1位

## § 3. 数据链路层

### 3.2. 差错检测和纠正

#### 3.2.1. 纠错码

##### ★ 海明 (hamming) 距离

一个编码系统中任意两个合法编码 (码字) 间不同的 bit 位数称为两个码字的码距, 整个系统中任意两个码字的最小距离就是该编码系统的码距, 称为海明距离

例3: 用如图所示6个bit位表示8种不同信息

=> 海明距离: 3

	a5	a4	a3	a2	a1	a0
0	0	0	0	0	0	0
1	0	1	1	0	0	1
2	1	0	1	0	1	0
3	1	1	0	0	1	1
4	1	1	0	1	0	0
5	1	0	1	1	0	1
6	0	1	1	1	1	0
7	0	0	0	1	1	1

问题1: 不失一般性,  $m$  位的合法编码, 要扩展到多少位, 才能纠错1位?

=> 假设加入  $r$  位, 共扩展到  $n$  位 ( $n = m + r$ )

=> 扩展到  $n$  位后, 每种编码错1位的情况有  $n$  种, 加上正确情况, 一共  $n+1$  种

=>  $m$  位, 共  $2^m$  种合法编码, 要保证所有合法/非法编码不重复, 则  $2^m * (n+1) \leq 2^n$

=>  $2^m * (n+1) \leq 2^m * 2^r$

=>  $n+1 \leq 2^r$

=>  $m+r+1 \leq 2^r$

=>  $2^{r-r} \geq m+1$

3bit合法编码:  
 $2^{r-r} \geq 3+1$ ,  $r=3$  (共6bit)

4bit合法编码:  
 $2^{r-r} \geq 4+1$ ,  $r=3$  (共7bit)

5bit合法编码:  
 $2^{r-r} \geq 5+1$ ,  $r=4$  (共9bit)

问题2: 海明距离和纠错/检错位数之间的关系?

=> 为了检测  $d$  位错误, 需要距离为  $d+1$  的编码方案 ( $d$  个1位错误不可能把一个有效码字变为另一个有效码字)

=> 为了纠正  $d$  位错误, 需要距离为  $2d+1$  的编码方案 (合法码字之间距离足够远, 即使  $d$  位变化, 仍是离原来的码字最近)

海明距离为3的编码, 能

纠正1位错误

或者发现2位错误

注意: 不能同时纠错和检错!

## § 3. 数据链路层

### 3.2. 差错检测和纠正

#### 3.2.1. 纠错码

##### ★ 最简单的奇偶检验码

奇偶性：一组 $n$ 个二进制bit位，如果有奇数个1，则称具有奇性，否则称为具有偶性

奇性检测方法：奇性 =  $a_n \oplus a_{n-1} \oplus a_{n-2} \oplus \cdots \oplus a_1$

=> 所有码元(bit)模2加/半加/异或(XOR)运算

=> 例：1011 => 奇性 (便于计算机运算)

奇偶校验：给每组增加一个校验位，是编码中的1的个数为奇数(奇校验)或偶数(偶校验)

=> 例：1011 => 奇校验 => 10110

偶校验 => 10111

=> 假设偶校验，则接收端：

$$S = a_n \oplus a_{n-1} \oplus a_{n-2} \oplus \cdots \oplus a_1 \oplus a_0 = 0 \text{ (正确)}$$

1 (错误)

=> S只有两种状态，因此只能表示正确/错误

=> 接收时共1/3..位翻转，则能发现错误

共2/4..位翻转，则不能发现错误

=> 只能发现错误，无法纠错

续问：检测出哪一位有错(纠错)，至少需要多少位的校验位？

=>  $d=1$ ,  $2d+1=3$

=> 至少需要海明距离为3的编码方案

## § 3. 数据链路层

### 3.2. 差错检测和纠正

#### 3.2.1. 纠错码

##### ★ 海明码 (Hamming码)

海明码是一种多重(复式)奇偶检错系统, 它将信息用逻辑形式编码, 以便能够检错和纠错。用在海明码中的全部传输码字是由原来的信息和附加的奇偶校验位组成的。每一个这种奇偶位被编在传输码字的特定位置上。当传输出现单bit错误时, 无论错误位置是信息位还是校验位, 都能够被检测出来

- 功能: 用多个奇偶校验来纠正单比特错
- 校验位位数定理: 设有 $m$ 位数据位,  $r$ 位校验位, 传输总位数为 $n$ , 若欲纠正单比特错, 则海明码校验位的位数为 $(m+r+1) \leq 2^r$
- 推导并使用长度为 $m$ 位的码字的海明码的步骤
  - 1、确定最小的校验位数 $k$ , 记为 $P_1$ 、 $P_2$ 、...、 $P_k$ , 每个校验位符合不同的奇偶测试规定
  - 2、原有信息和 $k$ 个校验位一起编成长为 $n=m+k$ 位的新码字, 选择 $k$ 校验位(0或1)以满足必要的奇偶条件
  - 3、对所接收的信息作所需的 $k$ 个奇偶检查
  - 4、如果所有的奇偶检查结果均为正确的, 则认为信息无错误
  - 5、如果发现有一个或多个校验位检查结果有错, 则由检查结果来确定唯一的错误位

例：数据位数 $m=4$ 的海明码推导过程

1、由公式  $(m+r+1) \leq 2^r$  可得  $r=3$   $n=7$  (简称为7, 4码)

2、将检验位放在1/2/4/8/16...等处，其余为数据位

码字位置	1	2	3	4	5	6	7
检验位	x	x		x			
数据位			x		x	x	x
复合码字	P1	P2	D1	P3	D2	D3	D4


- 理论上可放任意位置，放在 $2^k$ 位置的方便性稍后讲
- 为方便讲解，实际顺序与机内顺序反
- 不失一般性，假设偶校验，则：

P1 位填写D1、D2、D4位 (3/5/7) 的偶检验值

P2 位填写D1、D3、D4位 (3/6/7) 的偶检验值

P3 位填写D2、D3、D4位 (5/6/7) 的偶检验值

3、得到的海明码如右图所示



复合码字	P1	P2	D1	P3	D2	D3	D4
0	0	0	0	0	0	0	0
1	1	1	0	1	0	0	1
2	0	1	0	1	0	1	0
3	1	0	0	0	0	1	1
4	1	0	0	1	1	0	0
5	0	1	0	0	1	0	1
6	1	1	0	0	1	1	0
7	0	0	0	1	1	1	1
8	1	1	1	0	0	0	0
9	0	0	1	1	0	0	1
10	1	0	1	1	0	1	0
11	0	1	1	0	0	1	1
12	0	1	1	1	1	0	0
13	1	0	1	0	1	0	1
14	0	0	1	0	1	1	0
15	1	1	1	1	1	1	1

## 例：数据位数m=4的海明码推导过程

<p>假设偶校验，则：</p> <p>P1 位填写D1、D2、D4位 (3/5/7)</p> <p>P2 位填写D1、D3、D4位 (3/6/7)</p> <p>P3 位填写D2、D3、D4位 (5/6/7)</p> <p>问：应该如何选择校验位对应的数据位？</p> <p>=&gt; 校验位占用 1/2/4/8/16...</p> <p>数据位占用 3/5/6/7...</p> <p>=&gt; 数据位 3 = 1+2，出现在 P1/P2中</p> <p>数据位 5 = 1+4，出现在 P1/P3中</p> <p>数据位 6 = 2+4，出现在 P2/P3中</p> <p>数据位 7=1+2+4，出现在 P1/2/3中</p> <p>=&gt; 数据位编号拆为 <math>2^{P1} + 2^{P2} + \dots + 2^{Pk}</math></p> <p>校验位占用 <math>2^k</math> 位置的好处</p>	<p>假设传输 2 (0101010)</p> <p>且正确</p> <p><math>A = 1+3+5+7 = 0</math></p> <p><math>B = 2+3+6+7 = 0</math></p> <p><math>C = 4+5+6+7 = 0</math></p> <p>=&gt; 校验正确</p>
<p>假设偶校验，则：</p> <p>P1 位填写D1、D2、D4位 (3/5/7)</p> <p>P2 位填写D1、D3、D4位 (3/6/7)</p> <p>P3 位填写D2、D3、D4位 (5/6/7)</p> <p>问：应该如何选择校验位对应的数据位？</p> <p>问：怎么确定是哪位出错？</p> <p>=&gt; 校验位按从大到小顺序排列 (P3P2P1)</p> <p>对应每个检验位的验算结果 (C B A)</p> <p>=&gt; CBA = 110 (6)</p> <p>=&gt; 第6位出错</p>	<p>假设传输 2 (0101010)</p> <p>且D3位错误 (01010<b>0</b>0)</p> <p><math>A = 1+3+5+7 = 0</math></p> <p><math>B = 2+3+<b>6</b>+7 = 1</math></p> <p><math>C = 4+5+<b>6</b>+7 = 1</math></p> <p>=&gt; 校验出错</p> <p>错在哪位？</p> <p>A正确 (无D3)</p> <p>B/C错误 (有D3)</p> <p>=&gt; D3位错误</p> <p>=&gt; 纠正为 01010<b>1</b>0</p>

复合码字	P1	P2	D1	P3	D2	D3	D4
0	0	0	0	0	0	0	0
1	1	1	0	1	0	0	1
2	0	1	0	1	0	1	0
3	1	0	0	0	0	1	1
4	1	0	0	1	1	0	0
5	0	1	0	0	1	0	1
6	1	1	0	0	1	1	0
7	0	0	0	1	1	1	1
8	1	1	1	0	0	0	0
9	0	0	1	1	0	0	1
10	1	0	1	1	0	1	0
11	0	1	1	0	0	1	1
12	0	1	1	1	1	0	0
13	1	0	1	0	1	0	1
14	0	0	1	0	1	1	0
15	1	1	1	1	1	1	1

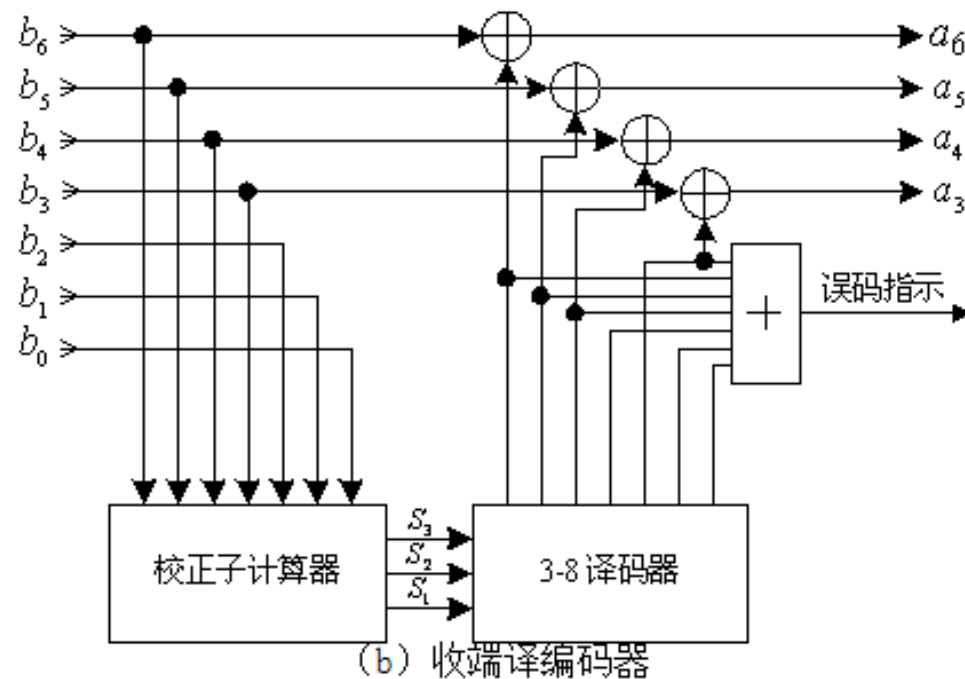
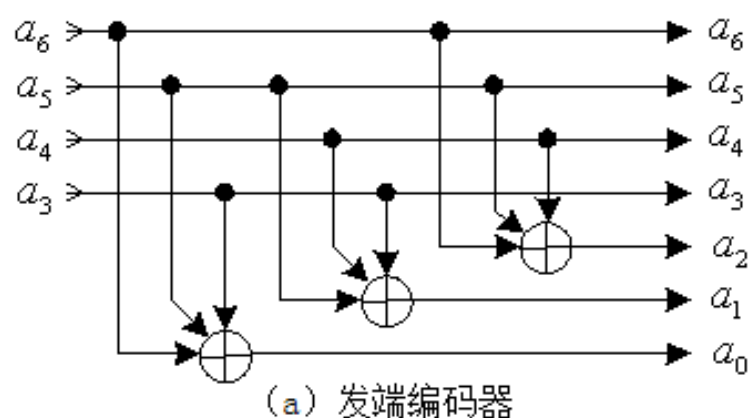
## § 3. 数据链路层

### 3.2. 差错检测和纠正

#### 3.2.1. 纠错码

##### ★ 海明码 (Hamming码)

海明码是一种多重(复式)奇偶检错系统, 它将信息用逻辑形式编码, 以便能够检错和纠错。用在海明码中的全部传输码字是由原来的信息和附加的奇偶校验位组成的。每一个这种奇偶位被编在传输码字的特定位置上。当传输出现单bit错误时, 无论错误位置是信息位还是校验位, 都能够被检测出来



## § 3. 数据链路层

### 3.2. 差错检测和纠正

#### 3.2.1. 纠错码

#### 3.2.2. 检错码

#### ★ 常用种类

奇偶

检验和

循环冗余校验 (Cyclic Redundancy Check)

#### ★ 循环冗余检验 (CRC) 的计算方法

- 在发送端把数据划分为组，每组  $m$  个比特
- 假设待传送的一组数据  $M$ ，在  $M$  的后面再添添加供差错检测用的  $r$  位冗余码一起发送 ( $n=m+r$ )
- 用二进制的模 2 运算进行  $2^r$  乘  $M$  的运算，相当于在  $M$  后面添加  $r$  个 0
- 得到的  $(m+r)$  位的数除以事先选定好的长度为  $(r+1)$  位的除数  $P$ ，得出商是  $Q$  而余数是  $R$ ，余数  $R$  比除数  $P$  少 1 位，即  $R$  是  $r$  位。
- 把余数  $R$  作为冗余码添加在数据  $M$  的后面发送出去。发送的数据是： $2^r * M + R$

例：每组  $m$  位  $m = 6$   
待传数据位  $M = 101001$   
设  $r = 3$   
选定除数  $P = 1101$   
则：被除数  $2^r * M = 101001000$

模 2 除法运算结果：

商  $Q = 110101$

余数  $R = 001$

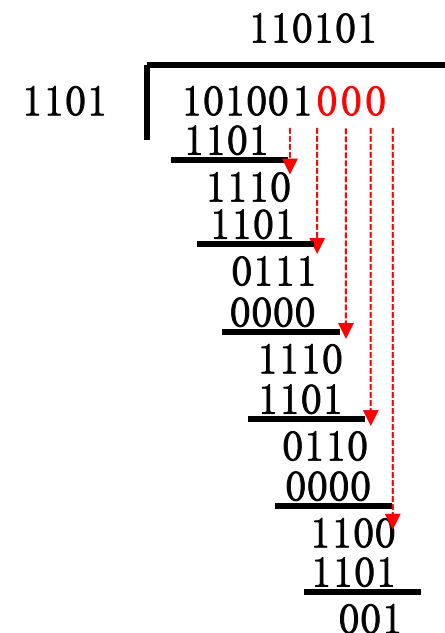
把余数  $R$  作为冗余码添加在数据  $M$  的后面 ( $2^r * M + R$ )

则：发送数据是  $101001001$

共  $(m+r)$  位

简易法则：

商=本次运算的被除数最高位  
减法不借位 (XOR)





## § 3. 数据链路层

### 3.2. 差错检测和纠正

#### 3.2.2. 检错码

##### ★ 循环冗余检验(CRC)的校验方法

m位数据 : 101001  
选定的除数 : 1101  
发送的数据 : 101001001

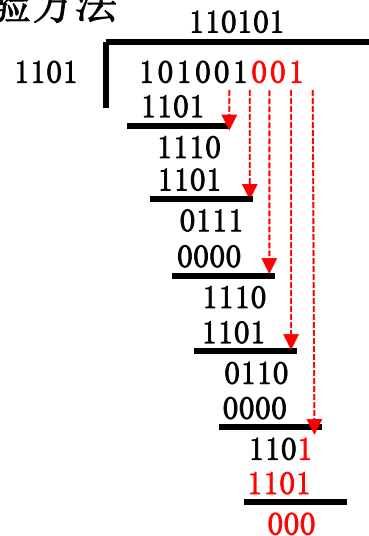
假设接收数据为: 101001001

检验方法:

接收数据/除数

=> 余数为0: 正确

=> 余数非0: 有错



m位数据 : 101001  
选定的除数 : 1101  
发送的数据 : 101001001

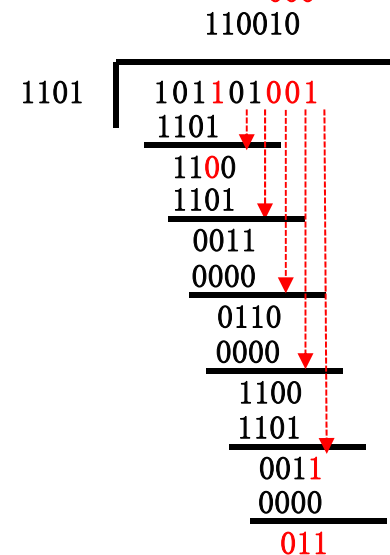
假设接收数据为: 101101001

检验方法:

接收数据/除数

=> 余数为0: 正确

=> 余数非0: 有错



## § 3. 数据链路层

### 3.2. 差错检测和纠正

#### 3.2.2. 检错码

##### ★ 循环冗余检验(CRC)的原理

- 将位串看成是系数为0/1的多项式 $M(x)$ ，一个 $m$ 位的帧被看做是 $m-1$ 次多项式的系数列表

例:  $M=101001 \Rightarrow M(x)=1x^5+0x^4+1x^3+0x^2+0x^1+1x^0$

- 多项式的算数运算规则

以2为模，加法不进位，减法不借位(都相当于 $\oplus$ )

除法按长除法规则(多项式除法, 其中减法为模2)

- 对多项式进行编码时，发送/接收双方按预先商定一个生成多项式 $G(x)$ ，要求该多项式的最高位和最低位必须是1

- P. 166 CRC计算方法:

(1) 假设 $G(x)$ 的阶位 $r$ ，在帧的最低位端加上 $r$ 个0，使得该帧现在包含 $m+r$ 位，对应多项式为 $x^rM(x)$

(2) 利用模2除法，用对应于 $G(x)$ 的位串去除对应于 $x^rM(x)$ 的位串

(3) 利用模2减法，从对应于 $x^rM(x)$ 的位串中减去余数(总是小于等于 $r$ 位)，结果就是将被传输的带校验和的帧。它的多项式不妨设为 $T(x)$

例: 数据  $M = 101001$        $M(x)=1x^5+0x^4+1x^3+0x^2+0x^1+1x^0$  (阶= $m-1$ )

设  $r = 3$

除数  $P = 1101$        $G(x)=1x^3+1x^2+0x^1+1x^0$  (阶=3)

则: 被除数  $2^r * M = 101001000$        $x^rM(x)$

商  $= 110101$        $Q(x) = x^rM(x) / G(x)$

余数  $= 001$        $R(x) = x^rM(x) \% G(x)$

带CRC的帧       $T(x) = x^rM(x) - R(x)$

101001000
-      001
101001001

## § 3. 数据链路层

### 3.2. 差错检测和纠正

#### 3.2.2. 检错码

##### ★ 循环冗余检验(CRC)的原理

商(被忽略)

由CRC算法可知,  $x^rM(x) = G(x)Q(x) + R(x)$

=> 因为 $R(x)$ 是 $x^rM(x)$ 除以 $G(x)$ 的余数, 记为  $R(x) = x^rM(x) \% G(x)$

=> 发送方将  $T(x) = x^rM(x) - R(x)$  传输到信道上

(因为XOR性质, 也可以理解为 $x^rM(x) + R(x)$ )

=> 接收方收到 $T'(x)$

注: 若 $T(x)$ 能被 $G(x)$ 整除, 记为 $T(x)/G(x)=0$   
否则记为 $T(x)/G(x) \neq 0$

=> 接收正确, 则 $T'(x) = T(x)$

=> 做 $T'(x)/G(x)$

=>  $T(x)/G(x)$

=>  $(x^rM(x) - R(x))/G(x)$

=>  $((G(x)Q(x) + R(x)) - R(x))/G(x)$

=>  $G(x)Q(x)/G(x)$

=>  $Q(x)$

=>  $T'(x)$ 一定能被 $G(x)$ 整除

(被除数-余数, 差值一定能被除数除尽)

=> 接收错误, 则 $T'(x) \neq T(x)$

=>  $T'(x) = T(x) + E(x)$  (正确错误多项式叠加)

=> 做 $T'(x)/G(x)$

=>  $(T(x) + E(x))/G(x)$

=>  $T(x)/G(x) + E(x)/G(x)$

=> 必0 +  $E(x)/G(x)$

=> 若 $E(x)/G(x) \neq 0$ , 则检测出错误

$E(x)/G(x) = 0$ , 则漏检

=> 接收正确, 则余数 为0

=> 接收错误, 则余数不为0

为0 (存在漏检可能)

发送: 101001001	$E(x)$
接收: 101101001	
=> 101001001 + 100000	
接收: 101101101	
=> 101001001 + 100100	
接收: 111101101	
=> 101001001 + 10100100	

## § 3. 数据链路层

### 3.2. 差错检测和纠正

#### 3.2.2. 检错码

##### ★ 循环冗余检验(CRC)的性质

- 若 $G(x)$ 中含有 $(x+1)$ 因子, 则检出所有奇数位错

证: 由假设条件可知,  $G(x) = (x+1) \cdot G'(x)$

$\Rightarrow$  假设漏检, 则  $E(x)/G(x) = 0$

$\Rightarrow E(x) = G(x) \cdot Q(x) = (x+1) \cdot G'(x) \cdot Q(x)$

$\Rightarrow$  假设 $E(x)$ 是奇数位错, 则必含有奇数个项

$\Rightarrow E(x) = 1 \longrightarrow$  矛盾

$\Rightarrow$  用1代入上式, 则

$\Rightarrow E(1) = (1+1) \cdot G'(1) \cdot Q(1) = 0$

$\Rightarrow E(x)/G(x) \neq 0$  (不能整除), 即可检出所有此类

- 若 $G(x)$ 中不含有 $x$ 的因子, 即 $G(x)$ 中含有常数项1, 则可检测所有突发长度 $\leq r$ 的突发错

证: 此时  $E(x) = x^i + \dots + x^j$  (突发错: 错误全部在 $i-j$ 间)

$\Rightarrow x^j(x^{i-j} + \dots + 1)$  ( $i-j \leq r-1$ )

$\Rightarrow$  因为  $G(x)$  是 $r$ 次多项式 ( $G(x)$  最高/最低必1)

$\Rightarrow E(x)/G(x) \neq 0$  (不能整除)

$\Rightarrow$  可检出所有错

## § 3. 数据链路层

### 3.2. 差错检测和纠正

#### 3.2.2. 检错码

##### ★ 循环冗余检验(CRC)的性质

- 若 $G(x)$ 中不含有 $x$ 的因子, 且对任何 $0 < k \leq n-1$ 的 $k$ , 除不尽 $x^k+1$ , 则能检测出所有独立的双bit错

证: 独立双错对应的  $E(x) = x^i + x^j$

$$\Rightarrow x^j(x^{i-j} + 1) \quad 0 < i-j \leq n-1$$

$\Rightarrow$  由已知条件可得, 必有 $E(x)/G(x) \neq 0$  (不能整除)

- 若 $G(x)$ 中不含有 $x$ 的因子, 则对突发长度为 $r+1$ 的突发错的漏检率为 $1/2^{(r-1)}$

证: 此时对应的差错多项式为  $E(x) = x^i + \dots + x^j$

$$\Rightarrow x^j(x^{i-j} + \dots + 1)$$

$$\Rightarrow x^j(x^r + \dots + 1) \quad (x^r + \dots + 1 \text{ 是 } r \text{ 次多项式})$$

$\Rightarrow$  由于 $G(x)$ 为 $r$ 次多项式,

当且仅当 $G(x) = x^r + \dots + 1$ 时,  $E(x)/G(x) = 0$  (能整除, 漏检)

$\Rightarrow$  又: 多项式 $x^r + \dots + 1$ 中间有 $r$ 项, 系数均为0/1

$\Rightarrow$  共有 $2^{r-1}$ 种不同的突发长度为 $r+1$ 的突发错, 其中仅一种检测不出

$\Rightarrow$  漏检率 $= 1/2^{r-1}$

## § 3. 数据链路层

### 3.2. 差错检测和纠正

#### 3.2.2. 检错码

##### ★ 循环冗余检验(CRC)的性质

● 若 $G(x)$ 中不含有 $x$ 的因子, 则对突发长度 $b > r+1$ 的突发错的漏检率为 $1/2^r$

证: 此时差错多项式为  $E(x) = x^i + \dots + x^j$

$$\Rightarrow x^j(x^{i-j} + \dots + 1)$$

$$\Rightarrow x^j(x^{b-1} + \dots + 1) \quad \text{其中 } b-1 > r$$

$$\Rightarrow \text{当 } (x^{b-1} + \dots + 1) = G(x) \cdot Q(x) \text{ 时, 漏检}$$

$$\Rightarrow G(x) \text{ 为 } r \text{ 次多项式, 且常数项为 } 1$$

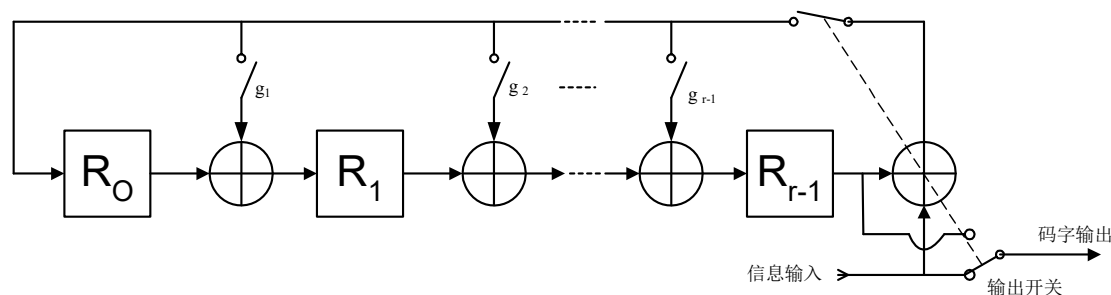
$$\Rightarrow Q(x) = x^{b-r-1} + \dots + 1$$

$$\Rightarrow \text{有 } 2^{(b-r-1)-1} = 2^{b-r-2} \text{ 种不同的可能性漏检}$$

$$\Rightarrow \text{突发长度为 } b \text{ 的差错有 } 2^{b-2} \text{ 种}$$

$$\Rightarrow \text{漏检率} = 2^{b-r-2} / 2^{b-2} = 2^{-r}$$

##### ★ 循环冗余检验(CRC)的硬件实现



## § 3. 数据链路层

### 3.3. 基本数据链路层协议

#### ★ 底层通信模型的基本假设

- 各层均为独立进程，通过消息传递进行通信

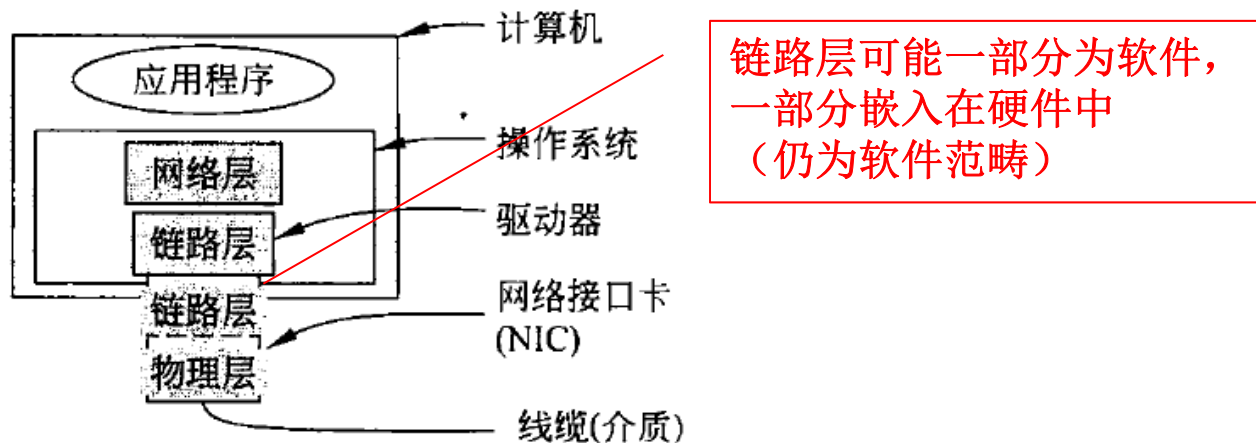


图 3-10 物理层、数据链路层和网络层的实现

## § 3. 数据链路层

### 3.3. 基本数据链路层协议

#### ★ 底层通信模型的基本假设

- 各层均为独立进程，通过消息传递进行通信
- 通信采用的是可靠的、面向连接的服务
- 上层（网络层）有足够的数据供发送
- 除通信错误外，不考虑其它错误（机器崩溃等）
- 数据链路层只处理一个信道（无并发、中断等）
- 网络层和数据链路层存在着严格的接口，网络层只处理**纯数据**（对于更上层的传输层、应用层而言，数据仍可能是带格式的），不会得到任何数据链路层的控制信息
- 通信双方的数据链路层需要等待对方时，采用阻塞等待方式，由事件激活（在数据链路层不考虑其上下层采用何种方式等待对方）

#### ★ 帧的基本格式

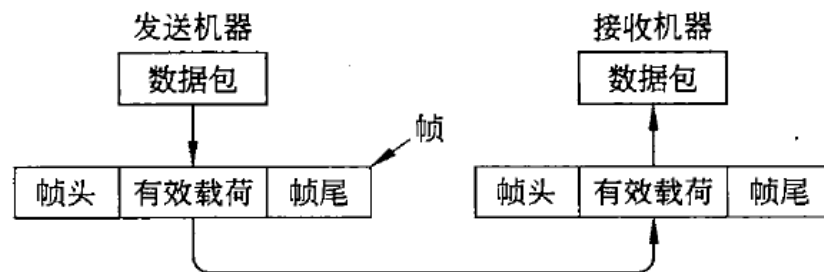


图 3-1 数据包和帧的关系

- 帧头含控制信息（包括帧的类型、发送序号、接收序号等），帧尾含校验和



## § 3. 数据链路层

### 3.3. 基本数据链路层协议

★ 用于描述协议的一些基本结构 (P. 169 图3-11)

```
#define MAX_PKT 1024 //数据包长度, 可按需改
```

```
typedef enum {  
    false,  
    true  
} boolean; //状态枚举量(false=0/true=1)
```

```
typedef unsigned int seq_nr; //发送序号
```

```
typedef struct {  
    unsigned char data[MAX_PKT];  
} packet; //数据包, 纯数据
```

```
typedef enum {  
    data, //数据包  
    ack, //确认包  
    nak //否定确认包  
} frame_kind; //帧类型枚举量
```

```
typedef struct {  
    frame_kind kind; //帧类型  
    seq_nr seq; //发送序号  
    seq_nr ack; //接收序号  
    packet info; //数据包  
} frame; //帧结构
```

- 不考虑头尾的FLAG字节、校验和等
- kind/seq/ack属于帧头的控制信息部分

```
typedef enum {  
    frame_arrival, //帧到达  
    cksum_err, //检验和错  
    timeout, //发送超时  
    network_layer_ready, //网络层就绪  
    ack_timeout //确认包超时
```

```
} event_type; //事件类型枚举量
```

- 每个算法对应的event\_type有所不同, 此处只是一个说明, 后面会实际定义
- P. 169 图3-11中无, 在后面每个协议中具体定义

## § 3. 数据链路层

### 3.3. 基本数据链路层协议

★ 协议描述中用到的基本函数(P. 169 图3-11)

● 所有函数均假设已实现，可以直接用

```
void wait_for_event( event_type *event);  
    //阻塞函数，等待事件发生  
  
void from_network_layer(packet *p);  
    //发送方从网络层得到纯数据包  
  
void to_network_layer(packet *p);  
    //接收方向网络层发送纯数据包  
    //去掉帧的类型、发送/确认序号等控制信息  
  
void from_physical_layer(packet *p);  
    //接收方从物理层取得帧  
    //帧头尾的FLAG字节、数据中的字节填充均已去掉  
    //调用本函数前已验证过校验和，若发生错误  
    //则发送cksum_err事件，因此只有帧正确的  
    //情况下会调用本函数  
  
void to_physical_layer(packet *p);  
    //发送方向物理层发送帧  
    //帧头尾加FLAG字节、数据中进行字节填充  
    //计算校验和放入帧尾
```

```
void start_timer(seq_nr k);  
    //启动第k帧的定时器  
  
void stop_timer(seq_nr k);  
    //停止第k帧的定时器  
  
void start_ack_timer(void);  
    //启动确认包定时器  
  
void stop_ack_timer(void);  
    //停止确认包定时器  
  
void enable_network_layer(void);  
    //解除网络层阻塞  
    //使可以产生新的network_layer_ready事件  
  
void disable_network_layer(void);  
    //使网络层阻塞  
    //不再产生新的network_layer_ready事件  
  
#define inc(k) if(k<MAX_SEQ) k=k+1; else k=0;  
    //使k在[0 ~ MAX_SEQ-1]间循环增长  
    //如果MAX_SEQ=1，则0/1互换
```

多个定时器采用链表方式，具体见下页

## § 3. 数据

### 3.3. 基本数据链路层协议

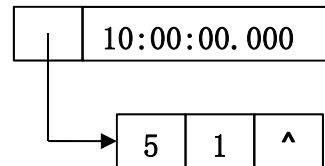
#### ★ 协议描述中用到的基本函数(P. 169 图3-11)

例：当前时间为10:00:00.000（精确到ms）硬件时钟每1ms产生一个中断，当前链表为空

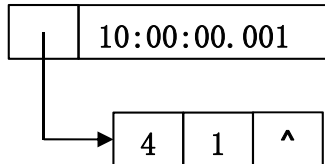
```
void start_timer(seq_nr k); //启动第k帧的定时器
void stop_timer(seq_nr k); //停止第k帧的定时器
```

- 以链表形式处理多帧同时存在的若干定时器  
启动定时器：放入链表对应位置(按超时时间隔排)  
停止定时器：从链表中移除  
发生超时：从链表中移除并发送timeout事件
- 每个链表结点有两个数据域，分别是超时时间和帧号，其中超时时间记录与前个结点的时间差
- 硬件时钟周期性产生中断，每次中断则更新链表(为高效，只更新首元结点)

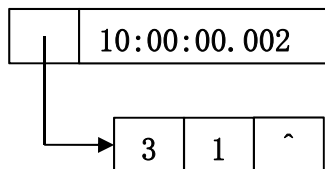
(1) 现在 1#帧 需要启动定时器，超时间隔为5ms



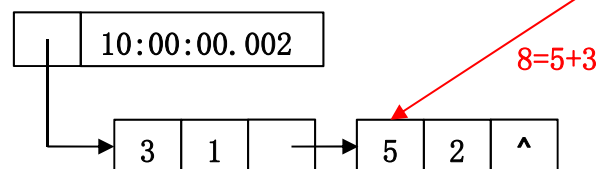
(2) 1ms 后



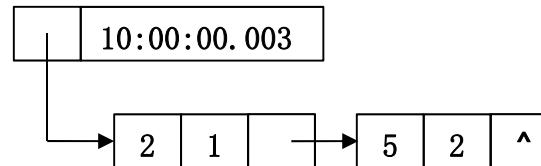
(3) 2ms后



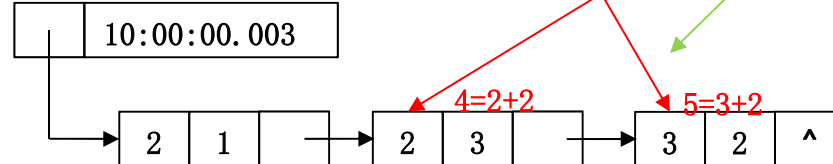
(4) 现在 2#帧 需要启动定时器，超时间隔为8ms



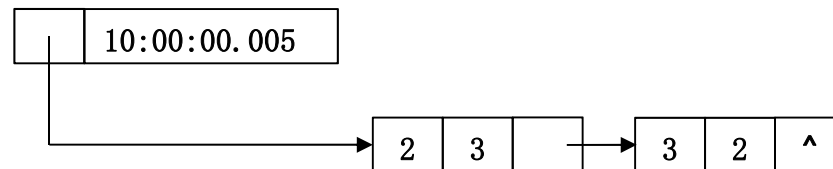
(5) 1ms



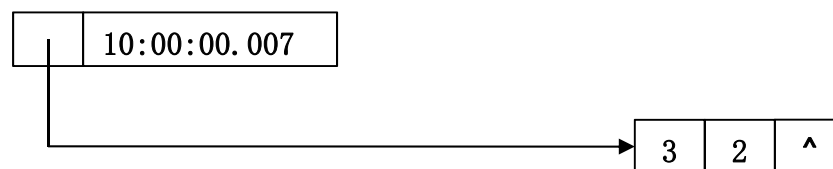
(6) 现在 3#帧 需要启动定时器，超时间隔为4ms



(7) 再1ms后，1#帧超时到，将节点移出链表



(8) 再2ms后，3#帧超时到，将节点移出链表



● 实际使用中，因为帧是按序发送，超时应该依次排列，不会出现插入在中间，实际总是尾部  
(本例仅是就算法而论)

## § 3. 数据

### 3.3. 基本数据链路层协议

★ 协议描述中用到的基本函数(P. 169 图3-11)

- 书P. 185图3-20，是3个定时器同时开始的情况，超时分别是5、13、19

```
void start_timer(seq_nr k);    //启动第k帧的定时器  
void stop_timer(seq_nr k);    //停止第k帧的定时器
```

- 以链表形式处理多帧同时存在的若干定时器  
启动定时器：放入链表对应位置(按超时时间间隔排)  
停止定时器：从链表中移除  
发生超时：从链表中移除并发送timeout事件
- 每个链表结点有两个数据域，分别是超时时间和帧号，其中超时时间记录与前个结点的时间差
- 硬件时钟周期性产生中断，每次中断则更新链表(为高效，只更新首元结点)

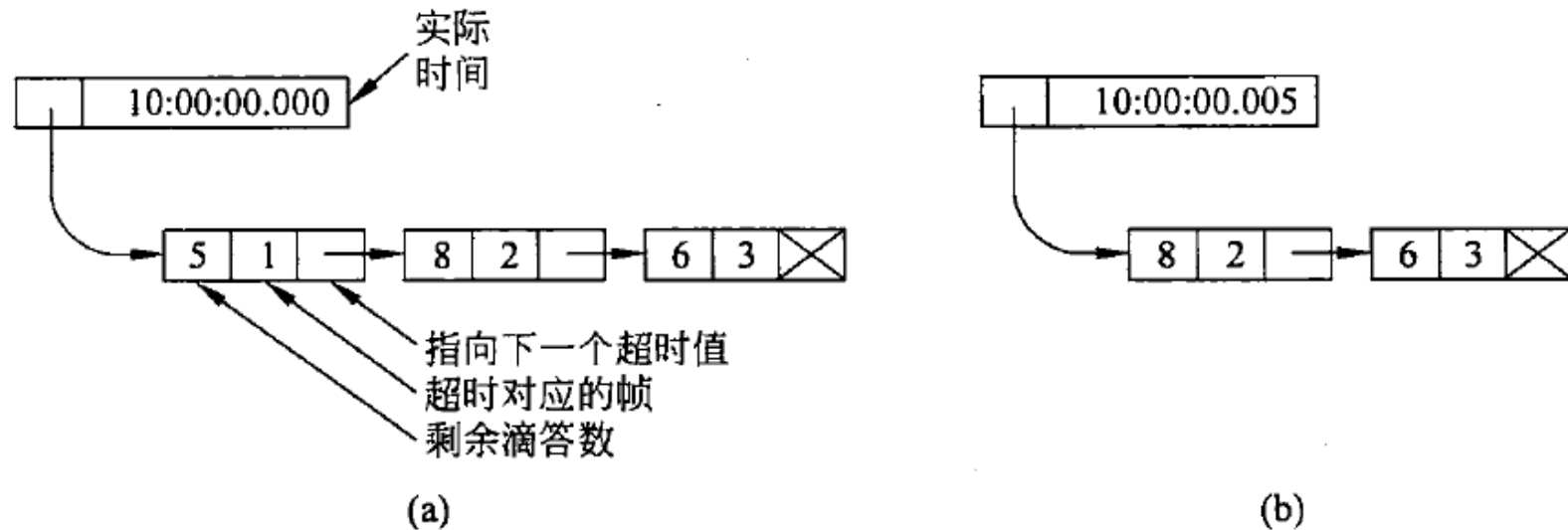


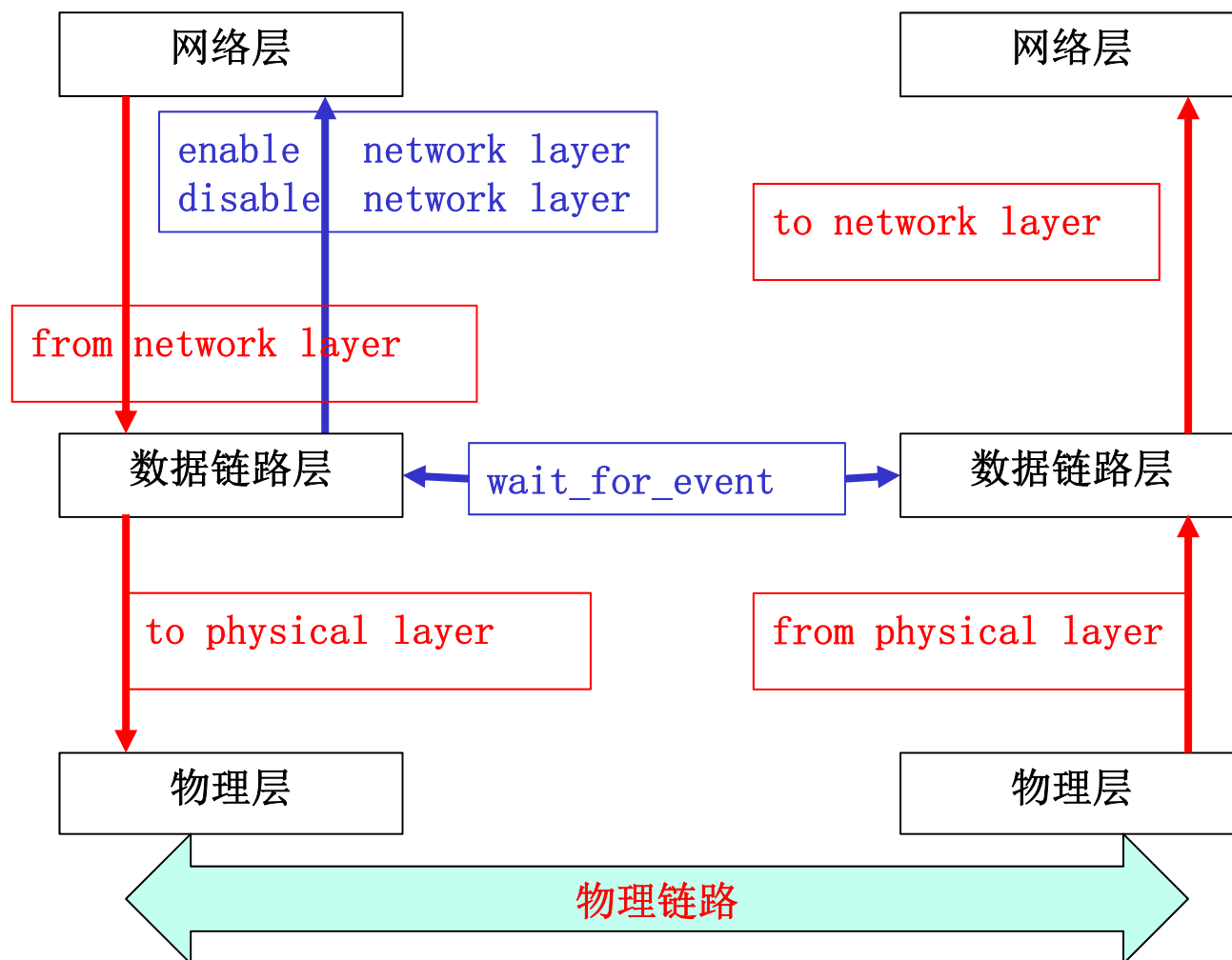
图 3-20 软件模拟多个计时器

(a) 队列中的计时器； (b) 第一个计时器超时后的情形

## § 3. 数据链路层

### 3.3. 基本数据链路层协议

#### ★ 基本通信模型与函数间的关系



## § 3. 数据链路层

### 3.3. 基本数据链路层协议

#### 3.3.1. 一个乌托邦式的单工协议

##### ★ 前提

- 数据单向传输
- 永远不出错(不必考虑错误处理)
- 上/下层永远处于就绪状态(缓存无限大)
- 数据的处理速度足够快, 时间可以忽略不计(只考虑传输时间)

##### ★ 算法

发送方无尽发送

接收方用阻塞+事件通知方式无尽接收

## ★ 算法

```
typedef enum {  
    frame_arrival  
} event_type; //只有接收端有一种事件产生
```

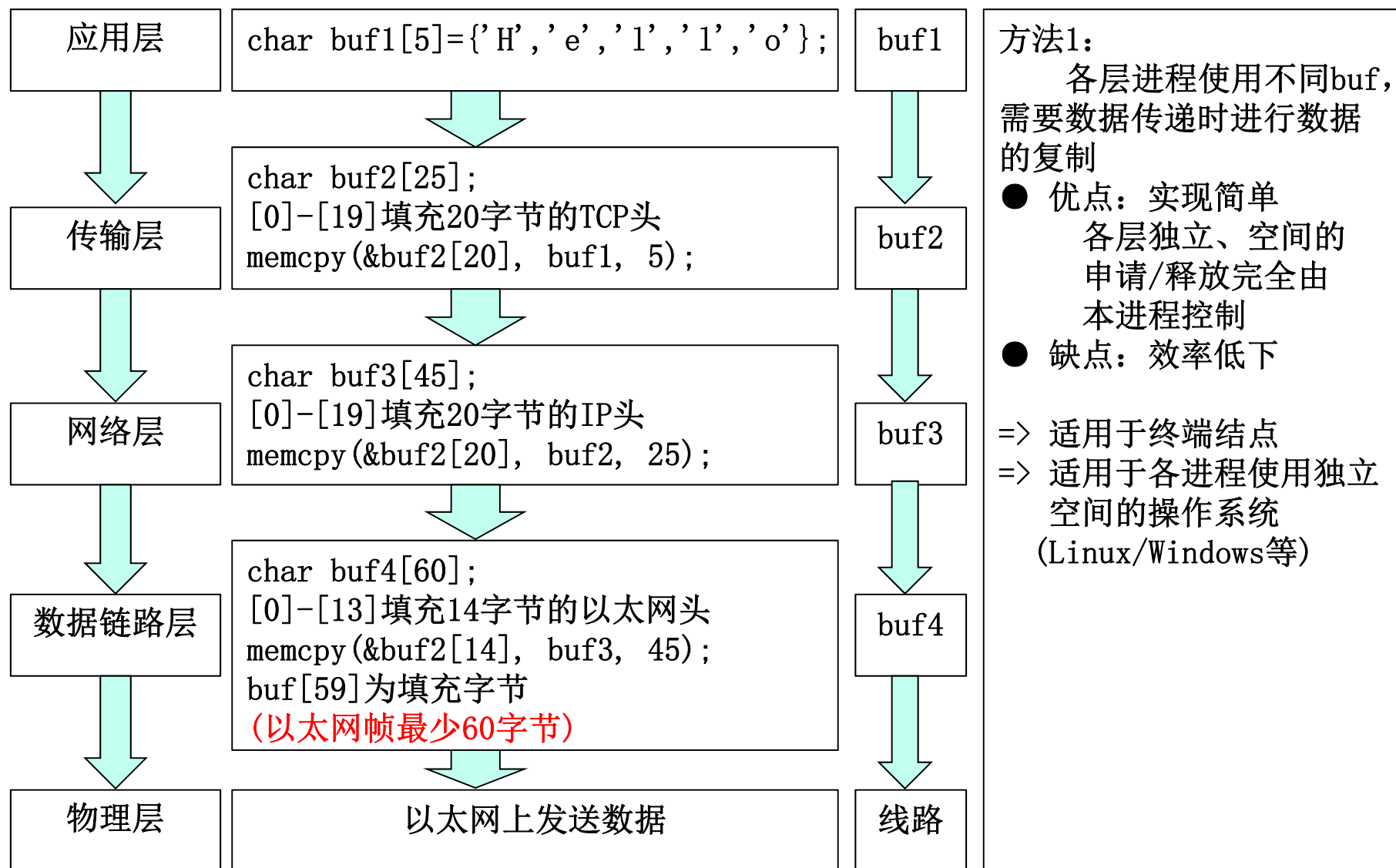
```
void sender1(void)  
{  
    frame s;  
    packet buffer;  
  
    while(true) {  
        from_network_layer(&buffer);  
        s.info = buffer;  
        to_physical_layer(&s);  
    }  
}
```

```
void receiver1(void)  
{  
    frame r;  
    event_type event;  
  
    while(true) {  
        wait_for_event(&event);  
        from_physical_layer(&r);  
        to_network_layer(&r.info);  
    }  
}
```

s.kind  
s.seq  
s.ack  
在本算法中均不使用，  
直接忽略即可(后续同)

注意：此处仅是算法表示  
若用C语言实际实现，则是  
错误的，因为字符串拷贝  
不能直接赋值  
(后续所有位置均不再说明)

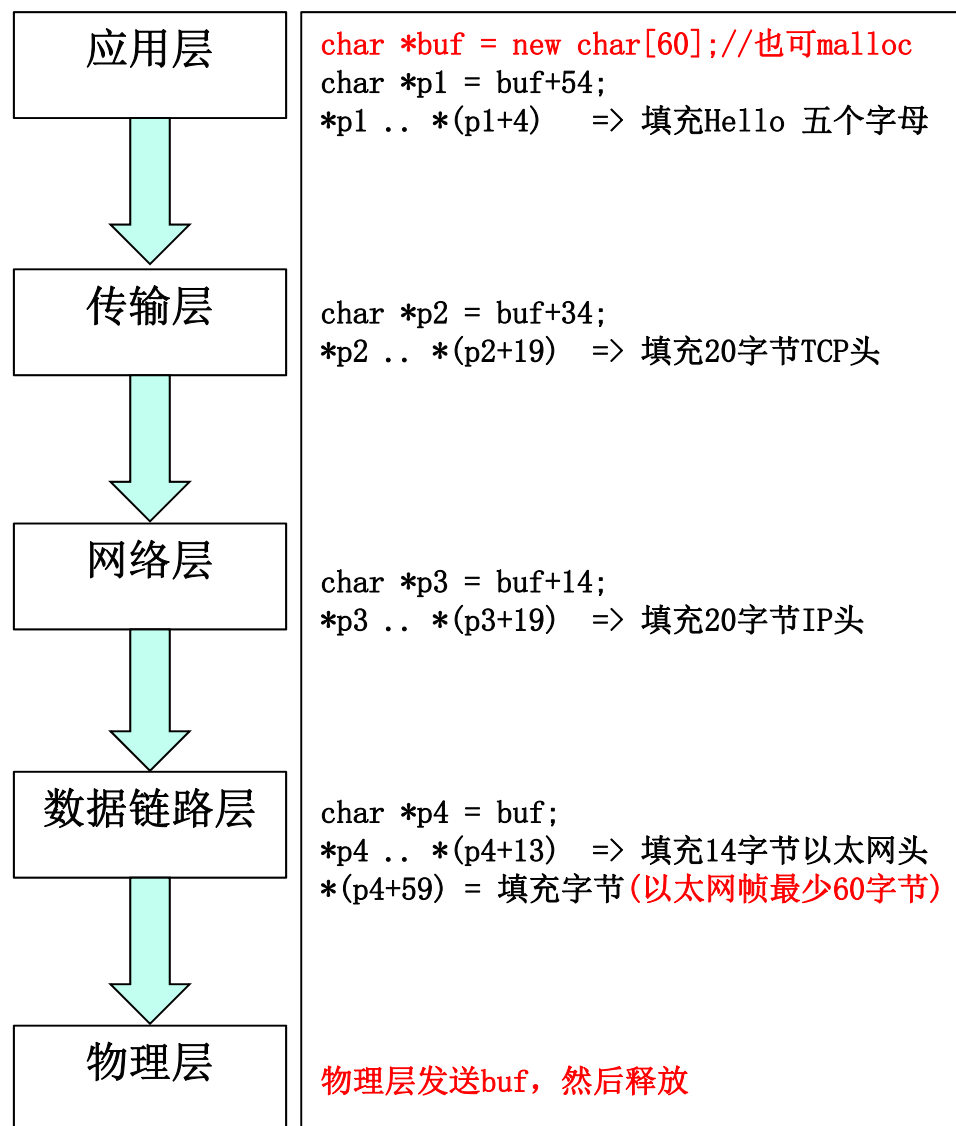
★ 补充：如何在一个系统的多层次的多个进程中传递数据？



如果接收，则反向进行



★ 补充：如何在一个系统的多层次的多个进程中传递数据？



申请 buf

各层在  
buf的  
相应位置  
填写  
自身负责  
的内容

发送buf  
释放buf

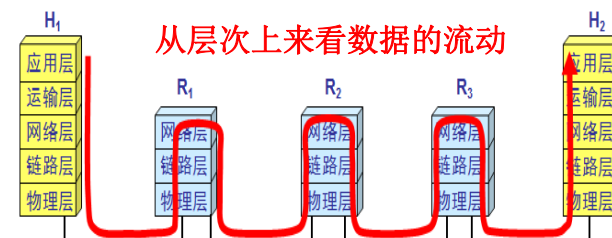
方法2:

在最上层进程中申请空间，将该空间的指针在不同进程间传递，最后由最下层空间释放(如果出错也可能由中间进程释放)

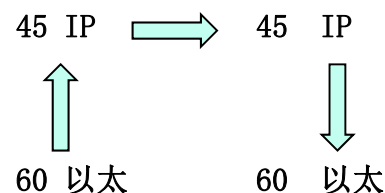
- 优点：效率高
- 缺点：实现复杂

=> 适用于转发结点  
(路由器/交换机等)

=> 适用于各进程使用统一空间的操作系统  
(VxWorks等)



**路由器R1:**  
假设以太口E0收数据"hello", E1转发  
则: E0收数据时申请60字节  
E1发送完成后释放60字节



如果接收，则反向进行

## § 3. 数据链路层

### 3.3. 基本数据链路层协议

#### 3.3.2. 无错信道上的单工停-等式协议

##### ★ 前提

- 数据单向传输
- 永远不出错(不必考虑错误处理)
- 接收方数据链路层缓存有限，处理能力有限，发送方能以高于接收方处理能力的速度发送
- 假设接收方的网络层及以上的处理能力高于数据链路层(不会在上层拥塞)

★ 最坏处理方法：调低数据链路层速度，使接收方的处理能力不拥塞

★ 算法：接收方发送确认帧，发送方收到后才能继续

## ★ 算法

```
typedef enum {  
    frame_arrival  
} event_type; //双方均只有一种事件产生
```

```
void sender2(void)  
{  
    frame s;  
    packet buffer;  
    event_type event;  
  
    while(true) {  
        from_network_layer(&buffer);  
        s.info = buffer;  
        to_physical_layer(&s);  
        //等待ACK, 因为假设不错, 不判断  
        wait_for_event(&event);  
    }  
}
```

```
void receiver2(void)  
{  
    frame r, s;  
    event_type event;  
  
    while(true) {  
        wait_for_event(&event);  
        from_physical_layer(&r);  
        to_network_layer(&r.info);  
        to_physical_layer(&s); //ACK  
    }  
}
```

## § 3. 数据链路层

### 3.3. 基本数据链路层协议

#### 3.3.3. 有错信道上的单工停-等式协议

##### ★ 前提

- 数据单向传输，且发送方必须收到确认帧才能发送下一帧
- 传输可能出错/丢失，且所有错误都能被检出
- 出错/丢失的既可能是数据帧，也可能是确认帧
- 错误需要重传，并可能重复多次，但必须保证接收方的网络层每个数据包只能接收一次，而且在网络层数据必须被按序接收

##### ★ 算法

- 为区分不同帧，发送/确认均需要加帧编号
- 为防止确认帧丢失，发送方要有超时重传定时器

## ★ 算法

```
typedef enum {  
    frame_arrival, //双方均有  
    cksum_err,     //双方均有  
    timeout        //仅发送方  
} event_type; //共三种事件会发生
```

另：本算法中，MAX\_SEQ = 1（只能同时1帧）

```
void sender3(void)  
{  
    seq_nr next_frame_to_send;  
    frame s;  
    packet buffer;  
    event_type event;  
  
    next_frame_to_send = 0; //初始帧号为0  
    from_network_layer(&buffer); //取首帧  
    while(true) {  
        s.info = buffer;  
        s.seq = next_frame_to_send;  
        to_physical_layer(&s);  
        start_timer(s.seq);  
        wait_for_event(&event); //等三个事件  
        if (event==frame_arrival) {  
            from_physical_layer(&s);  
            if (s.ack == next_frame_to_send) {  
                stop_timer(s.ack);  
                from_network_layer(&buffer);  
                inc(next_frame_to_send); //0-1变换  
            }  
            //ACK到，但s.ack不对  
        } //end of if  
        //如果event=cksum_err  
        //如果event=timeout  
    } //end of while  
}
```

回到while  
循环的开始，  
重发本帧

```
void receiver3(void)  
{  
    seq_nr frame_expected;  
    frame r, s;  
    event_type event;  
    frame_expectd = 0;  
    while(true) {  
        wait_for_event(&event); //等两个事件  
        if (event==frame_arrival) {  
            from_physical_layer(&r);  
            if (r.seq == frame_expected) { //序号匹配  
                to_network_layer(&r.info);  
                inc(frame_expected);  
            }  
            s.ack = 1-frame_expected; //无论是否匹配, 0-1变换  
            to_physical_layer(&s); //书上缺  
        }  
        //event = cksum_err, 放弃，直接等待下一个事件  
    }  
}
```

## § 3. 数据链路层

### 3.4. 滑动窗口协议

#### ★ 基本假设

- 数据在一个信道上双向传输(因为效率问题, 不考虑两个信道收发)
- 发送数据仍采用发送/确认方式
- 传输可能出错/丢失, 且所有错误都能被检出
- 出错/丢失的既可能是数据帧, 也可能是确认帧
- 错误需要重传, 并可能重复多次, 但必须保证接收方的网络层每个数据包只能接收一次, 而且在网络层数据必须被按序接收
- 考虑到效率, 可将ACK与数据帧合并发送(暂时延缓确认以便捎带确认), 但需有时间限制
- 发送方有超时重传机制, 接收方也有超时以便单独发送ACK包
- 为了提高效率, 考虑一次发送多个帧, 因此每个数据帧要带一个发送序号( $0 \sim 2^n-1$ ), 因此要设定一个窗口大小( $2^n$ ), 落在窗口内的帧才允许发送, 对应也有接收窗口
- 发送/接收窗口的上下届不必相同, 甚至可以不同大小
- 数据发送后, 仍然放在发送方的滑动窗口中, 直到数据被确认为止, 如果发送方的滑动窗口满, 则需要关闭网络层, 直到窗口有空闲为止

- 发送/接收窗口(●表示书上的阴影)

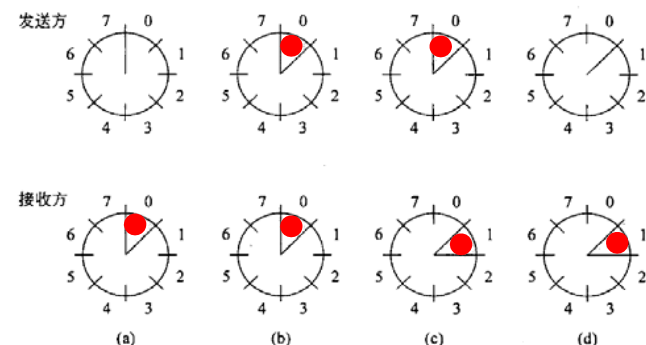


图 3-15 大小为 1 的滑动窗口, 序号 3 位  
(a) 初始化; (b) 第一帧发出之后; (c) 第一帧被接收之后; (d) 第一个确认被接收之后

## § 3. 数据链路层

### 3. 4. 滑动窗口协议

#### 3. 4. 1. 1位滑动窗口协议

##### ★ 前提

- 窗口尺寸为1

##### ★ 算法

## ★ 算法

```
typedef enum {  
    frame_arrival, //双方均有  
    cksum_err,     //双方均有  
    timeout        //双方均有  
} event_type; //共三种事件会发生
```

另：本算法中，MAX\_SEQ = 1（只能同时1帧）

```
void protocol4(void) //双方采用同一个协议  
{  
    seq_nr next_frame_to_send;  
    seq_nr frame_expected;  
    frame r, s;  
    packet buffer;  
    event_type event;  
  
    next_frame_to_send = 0; //初始帧号为0  
    frame_expect = 0;  
    from_network_layer(&buffer);  
    s.info = buffer;  
    s.seq = next_frame_to_send;  
    s.ack = 1 - frame_expected;  
    to_physical_layer(&s); //发送首帧  
    start_timer(s.seq);  
  
    while(true) {  
        wait_for_event(&event);  
        if (event==frame_arrival) { //到达的可能是数据帧/ACK帧  
            from_physical_layer(&r);  
            if (r.seq == frame_expected) { //序号正确则向上层送  
                to_network_layer(&r.info);  
                inc(frame_expected);  
            }  
            if (r.ack == next_frame_to_send) {  
                stop_timer(r.ack);  
                from_network_layer(&buffer); //buffer换新帧  
                inc(next_frame_to_send);  
            }  
        }  
        //收到对方帧(序号正确/错误)/收到cksum_err/timeout消息  
        s.info = buffer; //可能是新帧，也可能是旧帧  
        s.seq = next_frame_to_send;  
        s.ack = 1 - frame_expected;  
        to_physical_layer(&s);  
        start_timer(s.seq);  
    }  
}
```

因为ACK可能和Data一起送到，所以此处不能是else if，而是两个if并列



## § 3. 数据链路层

### 3.4. 滑动窗口协议

#### 3.4.2. 回退N协议

##### ★ 概念

- 如果每发送完一帧，都需要等待确认帧后才能发送下一帧，则对于带宽高，时延大的线路，利用率很低

例：线路带宽 - 50k bps，传输1bit到对方的时延 - 250ms（距离长），帧长 - 1000 bit  
t=0开始发送，使用一位滑动窗口协议

则：发送完成： $1000 / (50 \times 1000) = 0.02s = 20ms$

该帧第 1bit - 250ms后对方收到

该帧最后1bit -  $250 + 20 = 270ms$ 后对方收到

对方处理时间忽略不计，立即发ACK

则ack的第1bit -  $270 + 250 = 520ms$ 后收到

假设ack很短，520ms后ack帧接收完毕

=> 0.52s发送1000bit

=>  $1000 / 0.52 = 1923bps$

=> 线路带宽2kbps足够了

=>  $1923 / 50000 =$  线路利用率3.8%

=>  $1923 / 10M =$  线路利用率0.01%

- 解决方法：连续发送w帧

## § 3. 数据链路层

### 3.4. 滑动窗口协议

#### 3.4.2. 回退N协议

##### ★ 概念

- 如果每发送完一帧，都需要等待确认帧后才能发送下一帧，则对于带宽高，时延大的线路，利用率很低

- 解决方法：连续发送w帧

=> w是多少比较合适？

上例：20ms发送1000bit，等待500ms后再次发送

=> 等待的500ms时间可发送 $500/20=25$ 帧

=> 发送26帧后收到第1帧的ack

=> 发第27帧，收第2帧ack

=> 发第x帧，收x-25的ack

=> 线路上可持续发送

=> 0.52s发送  $26 * 1000 = 26000$  bit

=>  $26000/0.52 = 50000\text{bps} = 50\text{kbps}$

=> 线路利用率100%

=> w最大为26

## § 3. 数据链路层

### 3.4. 滑动窗口协议

#### 3.4.2. 回退N协议

##### ★ 概念

- 如果每发送完一帧，都需要等待确认帧后才能发送下一帧，则对于带宽高，时延大的线路，利用率很低

- 解决方法：连续发送w帧

- 带宽延时乘积 = 带宽\*延时

含义：代表信道上能容纳的bit的最大数量

上例：50kbps \* 250ms

=>  $50\text{kb/s} * 0.25\text{s} = 12.5\text{kb}$

=> 第1bit t=0时发送/t=250ms时收到

=> 整个信道最多容纳12500bit

=> 每帧xbit，则分为BD帧

=> 第1帧接收完成后，才发ACK

=> 此时信道上还有BD帧

=> 在等待ACK到达发送方的过程中，  
信道上又发送了BD帧

=> 发送2BD+1后，ACK到达

=>  $w=2BD+1$ 时线路利用率100%

=> w最大为2BD+1

=> 线路利用率  $\leq w/(2BD+1)$

## § 3. 数据链路层

### 3.4. 滑动窗口协议

#### 3.4.2. 回退N协议

##### ★ 概念

- 如果每发送完一帧，都需要等待确认帧后才能发送下一帧，则对于带宽高，时延大的线路，利用率很低
- 解决方法：连续发送w帧
- 带宽延时乘积 = 带宽\*延时  
含义：代表信道上能容纳的bit的最大数量
- 管道技术：在往返传播时间内，连续发送w帧，调整w而不使发送窗口溢出
- 在连续多帧的传输中出现了错误如何处理？
  - 方法1：回退n
  - 方法2：选择重传

## § 3. 数据链路层

### 3.4. 滑动窗口协议

#### 3.4.2. 回退N协议

##### ★ 前提

- n位发送窗口，1位接收窗口，即只接收正确序号的帧，并发确认包，丢弃错帧及序号不正确的帧及以后各帧
- 对MAX\_SEQ+1大小的窗口（0..MAX\_SEQ），最多允许发送MQX\_SEQ帧（窗口大小-1），否则无法判断重传帧

例：窗口大小8（0-7），发8帧，并保留在缓冲区中等待确认

=> 全部接受正确，对方发送7# ack

=> 发送方清空缓存，再发8帧(0-7)

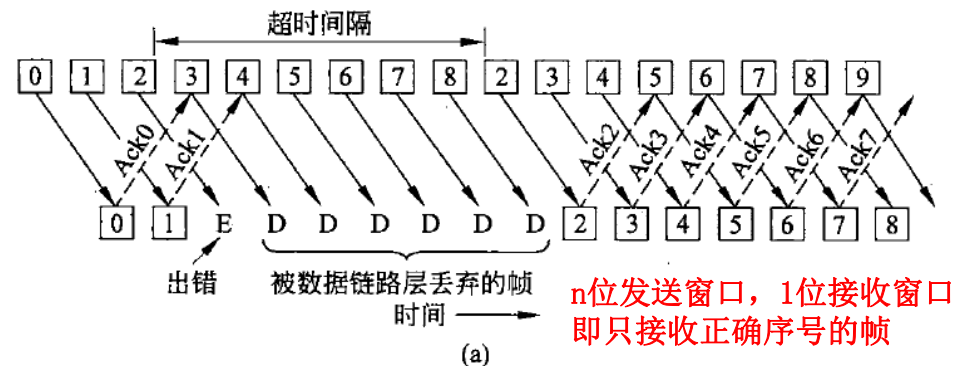
=> 对方发送7# ack （正确吗？）

=> 本次的第0帧出错，发送上次的7#ack

=> 本次全部正确，发送本次的7#ack

=> 无法分辨

- 发送方重传计时器超时后，重发所有待确认帧，而不仅仅是错帧



## ★ 算法

```
#define MAX_SEQ    7    //保持 $2^n-1$ 

typedef enum {
    frame_arrival,
    cksum_err,
    timeout,
    network_layer_ready
} event_type; //共四种事件会发生
```

```
static boolean between(seq_nr a, seq_nr b, seq_nr c)
{    //保证 $a \leq b < c$ , a为循环滑动窗口下界, c为上界(三种情况)
    if ( ((a<=b) && (b<c)) || ((c<a) && (a<=b)) || ((b<c) && (c<a)) )
        return (true);
    else
        return (false);
}

static void send_data(seq_nr frame_nr, seq_nr frame_expected, packet buffer[])
{
    frame s;
    s.info = buffer[frame_nr];    //0..MAX_SEQ中的一项
    s.seq = frame_nr;
    s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ+1);
    to_physical_layer(&s);
    start_timer(frame_nr);    //带帧号, 链表形式的定时器 (具体见前)
}
```

## ★ 算法

```
void protocol5(void)
{
    seq_nr next_frame_to_send;
    seq_nr ack_expected;
    seq_nr frame_expected;
    frame r;
    packet buffer[MAX_SEQ+1]; // 结构体数组[0..MAX_SEQ]
    seq_nr nbuffered;
    seq_nr i;
    event_type event;

    enable_network_layer(); // 允许上层发network_layer_ready
    ack_expected = 0;
    next_frame_to_send = 0; // 初始帧号为0
    frame_expect = 0;
    nbuffered = 0;

    while(true) {
        wait_for_event(&event);

        switch(event) { // 各事件分别处理
            case network_layer_ready:
            case frame_arrival:
            case cksum_err:
            case timeout:
            } // end of switch

        if (nbuffered < MAX_SEQ)
            enable_network_layer(); // 允许上层发数据
        else
            disable_network_layer(); // 不允许上层发数据
        } // end of while
    }
}
```

```
switch(event) { // 各事件分别处理
    case network_layer_ready:
        from_network_layer(&buffer[next_frame_to_send]);
        nbuffered = nbuffered + 1; // 填满发送窗口
        send_data(next_frame_to_send, frame_expected, buffer);
        inc(next_frame_to_send); // 发送窗上界增长
        break;

    case frame_arrival:
        from_physical_layer(&r);
        if (r.seq == frame_expected) { // 序号正确
            to_network_layer(&r.info);
            inc(frame_expected); // 接收序号增长
        }
        while(between(ack_expected, r.ack, next_frame_to_send)) {
            nbuffered = nbuffered - 1; // 确认后, -- 空出发送窗口
            stop_timer(ack_expected); // 链表中删
            inc(ack_expected); // 发送窗下界增长
        }
        break;

    case cksum_err:
        break;

    case timeout:
        next_frame_to_send = ack_expected; // 初始
        for (i=1; i<=nbuffered; i++) {
            send_data(next_frame_to_send,
                      frame_expected, buffer);
            inc(next_frame_to_send);
        }
        break;
} // end of switch
```

到达帧:  
单独的数据帧  
单独的确认帧  
数据捎带确认帧

因为ACK可能和数据一起送到,  
所以此处是if/while并列  
ack用while的原因是因为  
累计确认

重传的nbuffered不变

超时:  
重发发送缓冲区  
中全部未确认帧

## § 3. 数据链路层

### 3.4. 滑动窗口协议

#### 3.4.3. 选择重传协议

##### ★ 前提

- 接收方窗口大于1，可同时接受多帧
- 如果某帧出现错误，则向发送端发送nak，表示没收到该帧（主动请求重传，若本帧丢失，则由发送端的超时重传定时器来保证正确性），同时继续接收后续帧，直到接收窗口满
- 发送端收到nak后，重发被请求的帧（仅该帧），因此会出现乱序接收情况
- 采用累积确认方式，及收到某序号的确认帧，就认为之前所有帧均已确认

书上有错，应为nak2

接收方发现错误时，发送nak否定确认包，使发送方不必等到超时即可重传

累计确认：自动确认之前所有帧，而不必每帧都发



(b)

多位发送/接收窗口，  
可以缓存n帧



## § 3. 数据链路层

### 3.4. 滑动窗口协议

#### 3.4.3. 选择重传协议

##### ★ 前提

- 问题1: 对于选择重传方式,  $\text{MAX\_SEQ}+1$  的窗口, 最多可同时发送多少帧?  $\text{MAX\_SEQ}$  吗?

答: 应为  $(\text{MAX\_SEQ}+1)/2$ , 保证新老序号不重复

例:  $\text{MAX\_SEQ}=7$ , 已发0-6, 但尚未确认

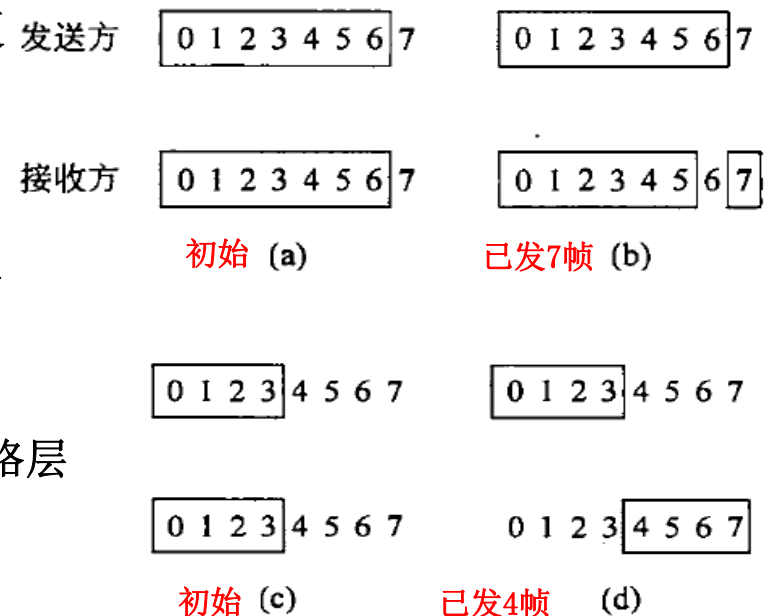
=> 对方正确接收, 发出确认帧, 并把0-6送到了网络层

=> 突发意外, 所有确认帧均丢失

=> 发送方重传计时器到, 重发0-6

=> 接收方认为是新一轮的0-6, 发出确认并送到网络层

=> 网络层重复接收, 出现错误



- 问题2: 对于选择重传方式,  $\text{MAX\_SEQ}+1$  的窗口, 接收方需要多大缓冲区?  $\text{MAX\_SEQ}+1$  吗?

答: `#define NF_BUFS (MAX_SEQ+1)/2`

则只需要 `NF_BUFS` 个就可以了

旧序号 `[0..NF_BUFS-1]`

新序号 `[NF_BUFS .. MAX_SEQ+1]`

虽然 `0/NF_BUFS`、`1/NF_BUFS+1...` 会竞争同一缓冲区, 但序号不重复

## ★ 算法

```
#define MAX_SEQ    7    //保持 $2^n-1$ 
#define NR_BUFS    ((MAX_SEQ+1)/2)

boolean no_nak = true;
seq_nr oldest_frame = MAX_SEQ + 1; //初始非法，具体详见P. 189 3.5节前最后一段

typedef enum {
    frame_arrival,
    cksum_err,
    timeout,
    network_layer_ready,
    ack_timeout
} event_type; //共五种事件会发生
```

```
static boolean between(seq_nr a, seq_nr b, seq_nr c)
{
    return ( ((a<=b) && (b<c)) || ((c<a) && (a<=b)) || ((b<c) && (c<a)) )
}

static void send_data(frame_kind fk, seq_nr frame_nr, seq_nr frame_expected, packet buffer[])
{
    frame s;
    s.kind = fk; //帧类型，data/ack/nak三种
    if (fk==data)
        s.info = buffer[frame_nr % NR_BUFS];
    s.seq = frame_nr;
    s.ack = (frame_expected + MAX_SEQ) % (MAX_SEQ+1);
    if (fk == nak)
        no_nak = false; //全局变量，发完nak置false
    to_physical_layer(&s);
    if (fk==data)
        start_timer(frame_nr % NR_BUFS); //带帧号，链表
    stop_ack_timer(); //有任何类型数据发送则停止ack计时器
}
```

## ★ 算法

```
void protocol6(void)
{
    seq_nr ack_expected, seq_nr next_frame_to_send;
    seq_nr frame_expected, seq_nr too_far;
    int i;
    frame r;
    packet out_buf[NR_BUFS], packet in_buf[NR_BUFS];
    boolean arrived[NR_BUFS];
    seq_nr nbuffered;
    event_type event;

    enable_network_layer();
    ack_expected = 0;
    next_frame_to_send = 0; //初始帧号为0
    frame_expect = 0;
    too_far = NF_BUFS; //初始为非法(合法0..NF_BUFS-1)
    nbuffered = 0;
    for (i=0; i<NR_BUFS; i++)
        arrived[i] = false;

    while(true) {
        wait_for_event(&event);
        switch(event) {
            case network_layer_ready:
            case frame_arrival:
            case cksum_err:
            case timeout:
            case ack_timeout:
        } //end of switch

        if (nbuffered < NR_BUFS)
            enable_network_layer(); //允许上层发数据
        else
            disable_network_layer(); //不允许上层发数据
    } // end of while
}
```

```
switch(event) {
    case network_layer_ready:
        nbuffered = nbuffered + 1;
        from_network_layer(&out_buf[next_frame_to_send % NR_BUFS]);
        send_frame(data, next_frame_to_send, frame_expected, out_buf);
        inc(next_frame_to_send);
        break;
    case frame_arrival:
        from_physical_layer(&r);
        if (r.kind == data) {
            if ( (r.seq != frame_expected) && no_nak) {
                send_frame(nak, 0, frame_expected, out_buf);
            }
            else
                start_ack_timer();

            if (between(frame_expected, r.seq, too_far) && arrived[r.seq%NF_BUFS]==false) {
                arrived[r.seq%NF_BUFS] = true;
                in_buf[r.seq%NR_BUFS] = r.info; //放入接收窗中
                while( arrived[frame_expected % NR_BUFS]) {
                    to_network_layer(&inbuf[arrived[frame_expected % NR_BUFS]]);
                    no_nak = true; //全局量, 置true表示不发nak
                    arrived[frame_expected % NR_BUFS] = false; //清接收窗口
                    inc(frame_expected);
                    inc(too_far); //如果初始, 则为0, 以后正常
                    start_ack_timer();
                }
            } // end of if(r.kind==data)

            /* 如果发送nak, 则找出最后一个确认帧序号的下一个 */
            if ((r.kind==nak) && between(ack_expected, (r.ack+1)%(MAX_SEQ+1), next_frame_to_send))
                send_frame(data, (r.ack+1)%(MAX_SEQ+1), frame_expected, out_buf);

            /* 处理收到的ack (独立帧或数据帧捎带过来) */
            while(between(ack_expected, r.ack, next_frame_to_send) {
                nbuffered = nbuffered - 1;
                stop_timer(ack_expected % NR_BUFS); //链表中删除
                inc(ack_expected);
            }
            break;
        }
        case cksum_err:
            if (no_nak) //没发过nak则发nak
                send_frame(nak, 0, frame_expected, out_buf);
            break;
        case timeout: //数据包超时则重发数据包
            send_frame(data, oldest_frame, frame_expected, out_buf);
            break;
        case ack_timeout: //ack超时则单独发ack包(未被捎带的情况)
            send_frame(ack, 0, frame_expected, out_buf);
            break;
    } //end of switch
```

判断接收帧  
是否落在  
接收窗口内