第5章 信息编码和逻辑运算

计算机只能处理0和1的数字信号，因此必须对各种信息进行编码，将它们转换为计算机能够接收的形式。本章从“抽象、编码、转换”等计算思维概念出发，讨论数值和字符的编码方法，以及信息压缩编码、数据检错编码等方法，并且讨论数理逻辑的基本形式。

5.1 数值信息编码

5.1.1 二进制数的编码

1. 信息的二进制数表示

信息编码包括基本符号和组合规则两大要素。信息论创始人香农(Claude Elwood Shannon)指出：**通信的基本信息单元是符号，而最基本的信息符号是二值符号**。最典型的二值符号是二进制数，它以1或0代表两种状态。香农提出，信息的最小度量单位为比特(bit)。任何复杂信息都可以根据结构和内容，按一定编码规则，最终转换为一组由0、1构成的二进制数据，并能无损地保留信息的含义。

2. 二进制编码的优点

计算机采用十进制数做信息编码时，加法运算需要10个(0～9)运算符号，加法运算有100个运算规则(0+0=0，0+1=1，0+2=2，…，9+9=18)。如果采用二进制编码，则运算符号只需要两个(0和1)，加法运算只有4个运算规则(0+0=0，0+1=1，1+0=1，1+1=10)。用二进制做逻辑运算非常方便，可以用1表示逻辑命题值“真”(True)，用0表示逻辑命题值“假”(False)。计算机采用二进制逻辑设计电路时，可以将算术运算的电路设计转换为二进制逻辑门电路设计，这大大降低了计算机设计的复杂性。

也许可以指出，由于加法运算服从交换律，故此0+1与1+0具有相同的运算结果，这样十进制运算规则可以减少到50个，但是对计算机设计来说，结构还是过于复杂。

也许还能够指出，十进制“1+2”只需要做一位加法运算，转换为8位二进制数后，至少要做8位加法运算(如$[0000001]_2+[00000010]_2$)，可见二进制数大大增加了计算的工作量。但是目前普通的计算机(4核2.0GHz的CPU)每秒可以做80亿次以上的64位二进制加法运算，可见计算机善于做大量的、机械的、重复的高速计算工作。

3. 计算机中二进制编码的含义

当计算机接收到一连串二进制符号(0和1字符串流)时，它并不“理解”这些二进制符号的含义。**二进制符号的具体含义取决于程序对它的解释**。

【例5-1】 二进制数符号$[01000010]_2$在计算机中的含义是什么？这个问题无法给出

简单的回答，这个二进制数的意义要看它的编码规则是什么。如果这个二进制数是采用原码编码的数值，则表示为十进制数＋65；如果采用 BCD(Binary Coded Decimal，二-十进制编码)编码，则表示为十进制数 42；如果采用 ASCII(American Standard Code for Information Interchange，美国信息交换标准代码)编码，则表示字符 A；另外，它还可能是一个图形数据、一个视频数据、一条计算机指令的一部分，或者其他含义。

4. 任意进制数的表示方法

任何一种进位制都能用几个有限基本数字符号表示所有数。进位制的核心是基数，如十进制的基数为 10，二进制的基数为 2。对任意 R 进制数，基本数字符号为 R 个，任意进制的数 N 可以用式(5-1)进行位权展开表示：

$$N = A_{n-1} \times R^{n-1} + A_{n-2} \times R^{n-2} + \cdots + A_0 \times R^0 + A_{-1} \times R^{-1} + \cdots + A_{-m} \times R^{-m} \tag{5-1}$$

式中：A 为任意进制数字；R 为基数；n 为整数的位数和权；m 为小数的位数和权。

【例 5-2】 将二进制数 1011.0101 按位权展开表示。

$[1011.0101]_2 = 1\times2^3 + 1\times2^1 + 1\times2^0 + 1\times2^{-2} + 1\times2^{-4}$。

5. 二进制数运算规则

计算机内部采用二进制数进行存储、传输和计算。用户输入的各种信息，由计算机软件和硬件自动转换为二进制数，在数据处理完成后，再由计算机转换为用户熟悉的十进制数或其他信息。二进制数的基本符号为 0 和 1，二进制数的运算规则是“逢二进一，借一当二”。二进制数的运算规则基本与十进制相同，二进制数四则运算规则如下。

(1) 加法运算：0＋0＝0，0＋1＝1，1＋0＝1，1＋1＝10(有进位)。

(2) 减法运算：0－0＝0，1－0＝1，1－1＝0，0－1＝1(有借位)。

(3) 乘法运算：0×0＝0，1×0＝0，0×1＝0，1×1＝1。

(4) 除法运算：0÷1＝0，1÷1＝1(除数不能为 0)。

二进制数用下标 2 或在数字尾部加 B 表示，如$[1011]_2$ 或 1011B。

习惯上，十进制数不用下标或标志表示，如 100 就是十进制的数值一百。

6. 十六进制数编码

二进制表示一个大数时位数太多，计算领域专业人员辨认困难。早期计算机采用八进制数来简化二进制数，随后发现八进制编码太长，而且容易混淆，于是又采用十六进制数表示二进制数。十六进制的符号是 0、1、2、3、4、5、6、7、8、9、A、B、C、D、E、F。运算规则是“逢 16 进 1，借 1 当 16”。**计算机内部并不采用十六进制数进行存储和运算**，采用十六进制数是方便专业人员可以很简单地将十六进制数转换为二进制数。

十六进制数用下标 16 或在数字尾部加 H 表示。如$[18]_{16}$ 或 18H；更多时候，十六进制数采用前缀 0x 的形式表示，如 0x000012A5 表示十六进制数 12A5。

常用数制与编码之间的对应关系如表 5-1 所示。

表 5-1　常用数制与编码之间的对应关系

十 进 制 数	十六进制数	二 进 制 数	BCD 编码
0	0	0000	0000
1	1	0001	0001

续表

十 进 制 数	十六进制数	二 进 制 数	BCD 编码
2	2	0010	0010
3	3	0011	0011
4	4	0100	0100
5	5	0101	0101
6	6	0110	0110
7	7	0111	0111
8	8	1000	1000
9	9	1001	1001
10	A	1010	0001 0000
11	B	1011	0001 0001
12	C	1100	0001 0010
13	D	1101	0001 0011
14	E	1110	0001 0100
15	F	1111	0001 0101

5.1.2 不同数制的转换

1. 二进制数与十进制数的转换

在二进制数与十进制数的转换过程中，必须频繁地计算 2 的整数次幂。表 5-2 和表 5-3 给出了 2 的整数次幂和十进制数值的对应关系。

表 5-2 2 的整数次幂与十进制数值的对应关系

2^n	2^9	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
十进制数值	512	256	128	64	32	16	8	4	2	1

表 5-3 二进制分数与十进制小数的关系

2^{-n}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
十进制分数	1/2	1/4	1/8	1/16	1/32	1/64	1/125	1/256
十进制小数	0.5	0.25	0.125	0.0625	0.03125	0.015625	0.0078125	0.00390625

二进制数转换成十进制数时，可以采用按权相加的方法，这种方法按照十进制数的运算规则，将二进制数各位的数码乘以对应的权再累加起来。

【例 5-3】 将$[1101.101]_2$按位权展开转换成十进制数。

二进制数按位权展开转换成十进制数的运算过程如图 5-2 所示。

二进制数	1		1		0		1	.	1		0		1		
位权	2^3		2^2		2^1		2^0	.	2^{-1}		2^{-2}		2^{-3}		
十进制数	8	+	4	+	0	+	1	+	0.5	+	0	+	0.125		=13.625

图 5-1 二进制数按位权展开过程

【例 5-4】 将二进制整数$[11010101]_2$转换为十进制整数，Python 指令如下。

```
>>> int('11010101', 2)                    # 将二进制整数转换为十进制数
213                                       # 输出转换结果
```

2. 十进制数与二进制数的转换

十进制数转换为二进制数时，整数部分与小数部分必须分开转换。整数部分采用除 2 取余法，就是将十进制数的整数部分反复除 2，如果相除后余数为 1，则对应的二进制数位为 1；如果余数为 0，则相应位为 0。逐次相除，直到商小于 2 为止。

小数部分采用乘 2 取整法，即将十进制小数部分反复乘 2。每次乘 2 后，所得积的整数部分为 1，相应二进制数为 1，然后减去整数 1，余数部分继续相乘；如果积的整数部分为 0，则相应二进制数为 0，余数部分继续相乘，直到乘 2 后小数部分＝0 为止。如果乘积的小数部分一直不为 0，则根据数值的精度要求截取一定位数即可。

【例 5-5】 将十进制数 18.8125 转换为二进制数。

整数部分除 2 取余，余数作为二进制数，从低到高排列(见图 5-2(a))。小数部分乘 2 取整，积的整数部分作为二进制数，从高到低排列(见图 5-2(b))。

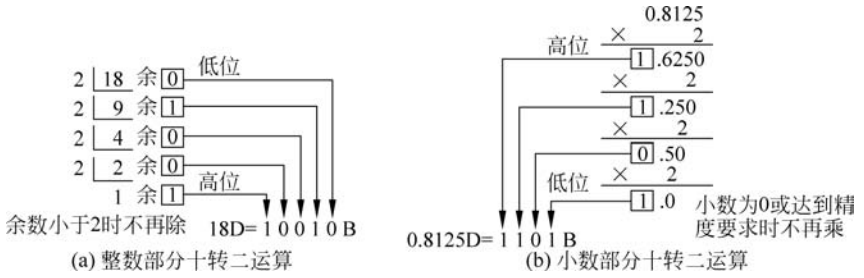


图 5-2 十进制数转换为二进制数的运算过程

运算结果为 18.8125＝$[10010.1101]_2$。

【例 5-6】 将十进制整数 234 转换为二进制数，Python 指令如下。

```
>>> bin(234)                              # 将十进制整数 234 转换为二进制数整数
'0b11101010'                              # 输出转换结果,前缀 0b 表示二进制数
```

说明：带小数的二转十、十转二案例，参见本书配套教学资源程序 F5-1.py。

3. 二进制数与十六进制数的转换

对于二进制整数，自右向左每 4 位分为一组，当整数部分不足 4 位时，在整数前面加 0 补足 4 位，每 4 位对应一位十六进制数；对二进制小数，自左向右每 4 位分为一组，当小数部分不足 4 位时，在小数后面(最右边)加 0 补足 4 位，然后每 4 位二进制数对应 1 位十六进制数，即可得到十六进制数。

【例 5-7】 将二进制数$[111101.010111]_2$转换为十六进制数。

$[111101.010111]_2$＝$[00111101.01011100]_2$＝$[3D.5C]_{16}$，转换过程如图 5-3 所示。

0011	1101	·	0101	1100
3	D	·	5	C

图 5-3 例 5-7 的转换过程

【例 5-8】 将二进制整数$[11010101]_2$转换为十六进制整数，Python 指令如下。

```
>>> hex(int('11010101', 2))          # 先转换为十进制整数,再转换为十六进制数
'0xd5'                               # 输出转换结果,前缀 0x 表示十六进制数
```

4. 十六进制数与二进制数的转换

将十六进制数转换成二进制数非常简单，只要以小数点为界，向左或向右每一位十六进制数用相应的四位二进制数表示，然后将其连在一起即可完成转换。

【例 5-9】 将十六进制数$[4B.61]_{16}$转换为二进制数。

$[4B.61]_{16}=[01001011.01100001]_2$，转换过程如图 5-4 所示。

十六进制数	4	B	·	6	1
二进制数	0100	1011	·	0110	0001

图 5-4 例 5-9 题图

【例 5-10】 将十六制整数$[4B]_{16}$转换为二进制整数，Python 指令如下。

```
>>> bin(int('4b', 16))               # 先转换为十进制数,再转换为二进制数
'0b1001011'                          # 输出转换结果,前缀 0b 表示二进制数
```

5. BCD 编码

计算机经常需要将十进制数转换为二进制数，利用以上转换方法存在两方面的问题：一是数制转换需要多次做乘法和除法运算，这大大增加了数制转换的复杂性；二是小数转换需要进行浮点运算，而浮点数的存储和运算较为复杂，运算效率低。

BCD 码用 4 位二进制数表示 1 位十进制数。BCD 有多种编码方式，8421 码是最常用的 BCD 编码，它各位的权值为 8、4、2、1，且与 4 位二进制数编码不同，它只选用了 4 位二进制数编码中前 10 组代码。BCD 码与十进制数的对应关系如表 5-1 所示。当数据有很多 I/O 操作时(如计算器，每次按键都是一个 I/O 操作)，通常采用 BCD 码，因为 BCD 码更容易将二进制数转换为十进制数。

二进制数使用$[0000\sim1111]_2$全部编码，而 BCD 码仅仅使用$[0000\sim1001]_2$十组编码，编码到$[1001]_2$后就产生进位，而二进制数编码到$[1111]_2$才产生进位。

【例 5-11】 将十进制数 10.89 转换为 BCD 码。

$10.89=[0001\ 0000.1000\ 1001]_{BCD}$，对应关系如图 5-6 所示。

十进制数	1	0	·	8	9
BCD 码	0001	0000	·	1000	1001

图 5-5 例 5-11 题图

【例 5-12】 将 BCD 码$[0111\ 0110.1000\ 0001]_{BCD}$转换为十进制数。

$[0111\ 0110.1000\ 0001]_{BCD}=76.81$，对应关系如图 5-6 所示。

BCD 码	0111	0110	·	1000	0001
十进制数	7	6	·	8	1

图 5-6 例 5-12 题图

【例 5-13】 将二进制数$[111101.101]_2$转换为 BCD 码。

如图 5-7 所示，二进制数不能直接转换为 BCD 码，因为编码方法不同，可能会出现非法码。可以将二进制数$[111101.101]_2$转换为十进制数 61.625 后，再转换为 BCD 码。

二进制数	0011	1101	·	1010
非法BCD码	~~0011~~	~~1101~~	·	~~1010~~
正确BCD码	0110	0001	·	0110 0010 0101

图 5-7 例 5-13 题图

常用数制之间的转换方法如图 5-8 所示。

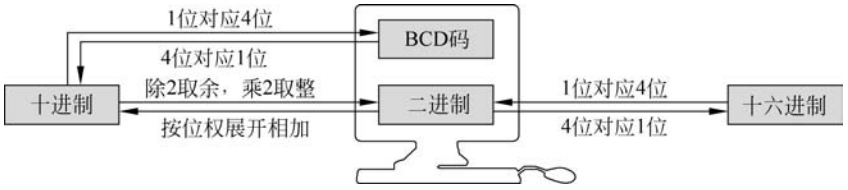


图 5-8 常用数制之间的转换方法

5.1.3 二进制整数编码

计算机以字节(Byte)组织各种信息，字节是计算机用于存储、传输、计算的基本计量单位，一个字节可以存储 8 位二进制数。

1. 无符号二进制整数编码形式

计算过程中，如果运算结果超出了数据表示范围称为溢出。如例 5-14 所示，8 位二进制无符号整数运算结果大于 255 时，就会产生溢出问题。

【例 5-14】 $[11001000]_2+[01000001]_2=1[00001001]_2$(8 位存储时，最高位溢出)。

解决数据溢出最简单的方法是增加数据的存储长度，数据存储字节越长，数值表示范围越大，越不容易产生溢出现象。如果小数字(小于 255 的无符号整数)采用 1 字节存储，大数字(大于 255 的无符号整数)采用多字节存储，则这种**变长存储会使存储和计算复杂化**，因为每个数据都需要增加 1 字节来表示数据长度，更麻烦的是计算机需要对每个数据进行长度判断。例如 Python 语言可以动态分配变量的数据长度，这也是导致它运行速度很慢的原因之一。解决数据不同存储长度的方法是建立不同的数据类型，静态编程语言(如 C 语言)在程序设计时要首先声明数据类型，对同一类型数据采用统一存储长度，如整型(int)数据的存储长度为 4 字节，长整型数据的存储长度为 8 字节。

【例 5-15】 无符号数$22=[10110]_2$在计算机中的存储形式如图 5-9 所示。

用1字节存储时：	00010110			
用2字节存储时：	00000000	00010110		
用4字节存储时：	00000000	00000000	00000000	00010110

图 5-9 数据的不同存储长度

由例 5-15 可见，数据的等长存储会浪费一些存储空间(4 字节存储时)，但是等长存储提高了运算速度，这是一种“以空间换时间”的计算思维模式。

2. 带符号二进制整数编码形式

数值有“正数”和“负数”之分，数学中用“+”表示正数(常被省略)，“−”表示负数。但是

计算机只有 0 和 1 两种状态，为了区分二进制数“＋”“－”符号，符号在计算机中也必须“数字化”。当用一个字节表示一个数值时，将该字节的最高位作为符号位，用 0 表示正数，用 1 表示负数，其余位表示数值大小。

“符号化”的二进制数称为机器数或原码，没有符号化的数称为真值。机器数有固定的长度（如 8、16、32、64 位等），当二进制数位数不够时，整数在左边（最高位前面）用 0 补足，小数在右边（最低位后面）用 0 补足。

【例 5-16】 ＋23＝[＋10111]$_2$＝[00010111]$_2$，如图 5-10 所示，最高位 0 表示正数。

真值	8 位机器数（原码）	16 位机器数（原码）
＋10111	**0**0010111	**0**0000000 00010111

图 5-10　例 5-16 题图

【例 5-17】 －23＝[－10111]$_2$＝[10010111]$_2$。二进制数[－10111]$_2$ 真值与机器数的区别如图 5-11 所示，最高位 1 表示负数。

真值	8 位机器数（原码）	16 位机器数（原码）
－10111	**1**0010111	**1**0000000 00010111

图 5-11　例 5-17 题图

5.1.4　二进制小数编码

1. 定点数编码方法

定点数是小数点位置固定不变的数。在计算机中，整数用定点数表示，小数用浮点数表示。当十进制整数很大时（十进制整数大于 18 位，或者存储大于 16 位），一般也用浮点数表示。

【例 5-18】 十进制数－73 的二进制真值为[－1001001]$_2$，如果用 2 字节存储，最高位（符号位）用 0 表示“＋”，1 表示“－”，二进制数原码的存储格式如图 5-12 所示。

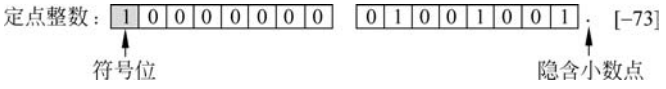


图 5-12　16 位定点整数的存储格式

在计算机系统中，整数（int）与浮点数（float）有截然不同的编码模式。如数值 12345 的整型编码为 0x00003039；而浮点数的编码为 0x4640E400。

2. 浮点数的表示

小数在计算机中的存储和运算是一个非常复杂的事情。目前有两位计算科学专家因研究浮点数的存储和运算而获得图灵奖。小数点位置浮动变化的数称为浮点数，浮点数采用指数表示，二进制浮点数的表示公式如下：

$$N = \pm M \times 2^{\pm E} \tag{5-2}$$

式中，N 为浮点数；M 为小数部分，称为“尾数”；E 为原始指数。**浮点数中，原始指数 E 的位数决定数值范围，尾数 M 的位数决定数值精度**。

【例 5-19】 [1001.011]$_2$＝[0.1001011]$_2$×2^4。

【例 5-20】 $[-0.0010101]_2=[-0.10101]_2\times2^{-2}$。

3. 二进制小数的截断误差

(1) 存储空间不足引起的截断误差。如果存储浮点数的长度不够,将会导致尾数最低位数据丢失,这种现象称为截断误差(舍入误差)。可以通过使用较长的尾数域来减少截断误差地发生。

【例 5-21】 十进制数 3.14159 转换为二进制数时为$[11.001001]_2$;将二进制数转换为科学计数法,则$[11.001001]_2=[0.11001001]_2\times2^2$;由此可知,尾数 M 符号位=$[0]_2$,尾数 M=$[11001001]_2$,指数 E 符号位=$[0]_2$,指数 E=2=$[10]_2$。如果按式(5-2)定义存储格式为:[指数 E 符号位][指数 E][尾数 M 符号位][尾数 M],则$[11.001001]_2=[0\ 10\ 0\ 11001001]_2$。这是一个 12 位二进制数,如果采用 8 位进行存储,将会导致尾数产生截断误差。

(2) 数值转换引起的截断误差。截断误差的另外一个来源是无穷展开式问题。例如,将十进制数 1/3 转换为小数时,总有一些数值不能精确地表示出来。二进制计数法与十进制计数法的区别在于:**二进制计数法中有无穷小数的情况多于十进制计数法**。

【例 5-22】 十进制小数 0.8 转换为二进制数为$[0.11001100...]_2$,后面还有无数个$[1100]_2$,这说明十进制小数转换成二进制数时,不能保证精确转换;二进制小数转换成十进制数也遇到同样的问题。

【例 5-23】 将十进制数 1/10 转换为二进制数也会遇到无穷展开式问题,总有一部分数不能精确地存储。编程语言在涉及浮点数运算时,会尽量计算精确一些,但是也会出现截断误差的现象。Python 运算的截断误差如下。

```
1  >>>(0.1 + 0.2) == 0.3                  # 注意," == "是等于符号
   False                                  # 比较结果为假
2  >>> 0.1 + 0.2
   0.30000000000000004                    # 在二-十进制转换中,会产生截断误差
3  >>> 4 - 3.6
   0.3999999999999999                     # 运算结果也会出现截断误差
```

说明:小数的十转二案例,参见本书配套教学资源程序 F5-2.py。

十进制小数转换成二进制小数时,如果遇到无穷展开式,计算过程会无限循环,这时可根据精度要求,取若干位二进制小数作为近似值,必要时采用“0 舍 1 入”的规则。

(3) 浮点数的运算误差。浮点数加法中,相加的顺序很重要,如果一个大数加上一个小数,那么小数就可能被截断。因此,多个数相加时,应当先相加小数字,将它们累计成一个大数字后,再与其他大数相加,避免截断误差。对大部分用户而言,商用软件提供的计算精度已经足够。一些特殊领域(如导航系统等),小误差会在运算中不断累加,最终产生严重的后果。如乘方运算中,当指数很大时,小误差将会呈指数级放大。

【例 5-24】 $1.01^{365}=37.8$; $1.02^{365}=1377.4$; $0.99^{365}=0.026$; $0.98^{365}=0.0006$。

4. IEEE 754 规格化浮点数

计算机中的实数采用浮点数存储和运算。浮点数并不完全按式(5-2)进行表示和存储。

如表 5-4 所示，计算机中的浮点数严格遵循 IEEE 754 标准。

表 5-4　IEEE 754 标准规定的浮点数规格

浮点数规格	码长/bit	符号 S/bit	阶码 e/bit	尾数 M/bit	十进制数有效位
单精度(float)	32	1	8	23	6～7
双精度(double)	64	1	11	52	15～16
扩展双精度数 1	80	1	15	64	20
扩展双精度数 2	128	1	15	112	34

说明：表中阶码 e 与式(5-2)中的原始指数 E 并不相同；尾数 M 与式(5-2)中的 M 也有所区别。

浮点数的表示方法多种多样，因此，**IEEE 754 标准对规格化浮点数进行了规定：小数点左侧整数必须为 1(如 1.xxxxxxxx)，指数采用阶码表示。**

【例 5-25】 $1.75=[1.11]_2$，用科学记数法表示时，小数点前一位为 0 还是 1 并不确定，IEEE 754 标准规定小数点前一位为 1，即规格化浮点数为 $1.75=[1.11]_2=[1.11]_2\times 2^0$。

浮点数规格化有两个目的：一是整数部分恒为 1 后，在存储尾数 M 时，就可以省略小数点和整数 1(与式(5-2)的区别)，从而可以用 23 位尾数域表达 24 位浮点数；二是整数位固定为 1 后，浮点数能以最大数的形式出现，尾数即使遭遇了极端截断操作(如尾数全部为 0)，浮点数仍然可以保持尽可能高的精度。

整数部分的 1 舍去后，会不会造成两个不同数的混淆呢？例如，$A=[1.010011]_2$ 中的整数部分 1 在存储时被舍去了，那么会不会造成 $A=[0.010011]_2$(整数 1 已舍去)与 $B=[0.010011]_2$ 两个数据的混淆呢？其实不会，因为数据 B 不是一个规格化浮点数，数据 B 可以改写成 $[1.0011]_2\times 2^{-2}$ 的规格化形式。所以省略小数点前的 1 不会造成任何两个浮点数的混淆。但是浮点数运算时，省略的整数 1 需要还原，并参与浮点数相关运算。

5. IEEE 754 浮点数编码方法

(1) 浮点数的阶码。原始指数 E 可能为正数或负数，但是 IEEE 754 标准没有定义指数 E 的符号位(见表 5-4)。这是因为二进制数规格化后，纯小数部分的指数必为负数，这给运算带来了复杂性。因此，IEEE 754 规定指数部分用阶码 e 表示，阶码 e 采用移码形式存储。阶码 e 的移码值等于原始指数 E 加上一个偏移值，32 位浮点数的偏移值为 127；64 位浮点数的偏移值为 1023。经过移码变换后，阶码 e 变成了正数，可以用无符号数存储。阶码 e 的表示范围是 1～254，阶码 0 和 255 有特殊用途。阶码为 0 时，表示浮点数为 0 值；阶码为 255 时，若尾数为全 0 则表示无穷大，否则表示无效数字。

(2) IEEE 754 浮点数的存储形式。IEEE 754 标准浮点数的存储格式如图 5-13 所示，编码方法是：省略整数 1、小数点、乘号、基数 2；从左到右采用：符号位 S(1 位，0 表示正数，1 表示负数)＋阶码位 e(余 127 码或余 1023 码)＋尾数位 M(规格化小数部分，长度不够时从最低位开始补 0)。

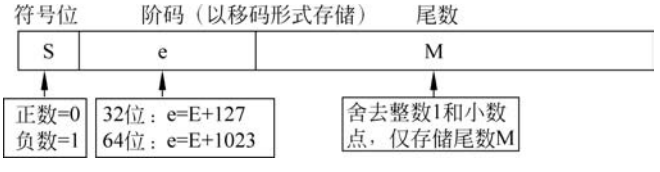


图 5-13　IEEE 754 规格化浮点数存储格式

6. 十进制实数转换为规格化二进制浮点数的案例

实数转换为 IEEE 754 标准浮点数的步骤是：将十进制数转换为二进制数→在 S 中存储符号值→将二进制数规格化→计算阶码 e 和尾数 M→连接$[S\text{-}e\text{-}M]_2$。

【例 5-26】 将十进制实数 26.0 转换为 32 位规格化二进制浮点数，如图 5-14 所示。

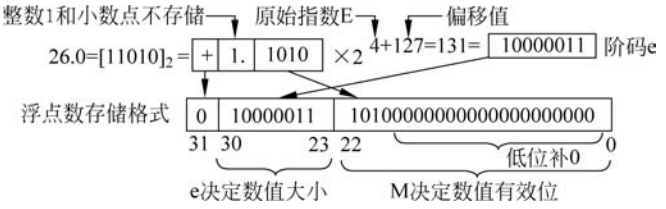


图 5-14 32 位规格化浮点数的转换方法和存储格式

(1) 将实数转换为二进制数，$26.0=[11010]_2$。

(2) 26.0 是正数，因此符号位 S=0。

(3) 将二进制数转换为规格化浮点数，$[11010]_2=[1.1010]_2\times2^4$。

(4) 计算浮点数阶码 e，E=4，偏移值=127，$e=4+127=131=[10000011]_2$。

(5) 浮点数尾数 M=$[1010]_2$，低位补 0 后 M=$[10100000000000000000000]_2$(注意，尾数省略了整数 1 和小数点，只取小数部分)。

(6) 连接符号位-阶码-尾数，N = $[01000001\ 11010000\ 00000000\ 00000000]_2$。

实现十进制实数转换为 32 位规格化二进制浮点数的 Python 程序如下。

```
1  import struct                                          # 导入标准函数 - 二进制数处理
2  def float_to_bits(f):                                  # 定义转换函数
3      s = struct.pack('>f', f)                           # 将浮点数转换为二进制数的字符串
4      return struct.unpack('>l', s)[0]                   # 将二进制数的字符串解包为元组
5  b = list(bin(float_to_bits(26.0)))                     # 调用函数,将元组转换为二进制数
6  b.insert(2, '0')                                       # 转换的实数为正数,在符号位插入 0
7  print('实数 26.0 的二进制浮点数为:', ''.join(b))        # 打印输出二进制数
```

```
>>>浮点数 26.0 的二进制数为:                             # 程序运行结果
0b10000011101000000000000000000000
```

其中，程序第 3 行的 struct.pack()函数将浮点数 f 转换为一个包装后的字符串。程序第 4 行的 struct.unpack()函数将元素解包为一个二进制数元组。

7. 二进制浮点数转换为十进制实数案例

【例 5-27】 将$[11000001\ 11001001\ 00000000\ 00000000]_2$ 转换成十进制数。

(1) 把浮点数分割成三部分：$[1]_2$、$[10000011]_2$、$[1001001\ 00000000\ 00000000]_2$，可得符号位 S=$[1]_2$，阶码 e=$[10000011]_2$，尾数 M=$[1001001\ 00000000\ 00000000]_2$。

(2) 还原原始指数 E：$E=e-127=[10000011]_2-[01111111]_2=[100]_2=4$。

(3) 还原尾数 M 为规格化形式：$M=[1.1001001]_2\times2^4$(“1.”从隐含位而来)。

(4) 还原为非规格化形式：$N=[S1.1001001]_2\times2^4=[S11001.001]_2$(S=符号位)。

(5) 还原为十进制数形式：$N=[S11001.001]_2=-25.125$(S=1，说明是负数)。

8. 浮点数能表示的最大十进制数

32 位浮点数(float)尾数 M 为 23 位,加上隐含的 1 个整数位,尾数部分共有 24 位,可以存储 6～7 位十进制有效数(见表 5-4)。由于阶码 e 为 8 位,IEEE 规定原始指数 E 的表示范围为－126～＋127,这样 32 位浮点数可表示的最大正数为$(2-2^{-23})\times 2^{127}=3.4\times 10^{38}$(有效数 6～7 位),可表示的最小正数为$2^{-126}=1.17\times 10^{-38}$(有效数 6～7 位)。

注意:最大/最小数涉及计算溢出问题;有效位涉及计算精度问题。

在 32 位浮点数中,原始指数 E 超过 127 时怎么处理? 0 的浮点数为 0.0 时怎么处理? 十进制有效数为什么是 6～7 位? 实数转换为浮点数的基本公式是什么? 浮点数如何进行四则运算? 总之,**小数的处理过程非常复杂**。浮点运算是对计算机性能的考验,世界 500 强计算机都是按浮点运算性能进行排名。

5.1.5 二进制补码运算

1. 原码在二进制数运算中存在的问题

用原码表示二进制数简单易懂,易于与真值进行转换。但二进制数原码进行加减运算时存在以下问题:一是做 $x+y$ 运算时,首先要判别两个数的符号,如果 x、y 同号,则相加;如果 x、y 异号,就要判别两数绝对值的大小,然后用绝对值大的数减去绝对值小的数。显然,这种运算方法不仅增加了运算时间,而且使计算机结构变得复杂。二是在原码中规定最高位是符号位,0 表示正数,1 表示负数,这会出现$[00000000]_2=[+0]_2$,$[10000000]_2=[-0]_2$ 的现象,而 0 有两种形式产生了“二义性”问题。三是两个带符号的二进制数原码运算时,在某些情况下,符号位会对运算结果产生影响,导致运算出错。

【例 5-28】 $[01000010]_2+[01000001]_2=[10000011]_2$(进位导致的符号位错误)。

【例 5-29】 $[00000010]_2+[10000001]_2=[10000011]_2$(符号位相加导致的错误)。

计算机需要一种可以带符号运算,而运算结果不会产生错误的编码形式,而“补码”具有这种特性。因此,**计算机中整数普遍采用二进制补码进行存储和计算**。

2. 二进制数的反码编码方法

二进制正数的反码与原码相同,负数的反码是该数原码除符号位外各位取反。

【例 5-30】 二进制数字长为 8 位时,$+5=[00000101]_{原}=[00000101]_{反}$。

【例 5-31】 二进制数字长为 8 位时,$-5=[10000101]_{原}=[11111010]_{反}$。

3. 补码运算的概念

两个数相加,计算结果的有效位(即不包含进位)为 0 时,称这两个数互补。如 10 以内的补码对有 1－9、2－8、3－7、4－6、5－5。十进制数中,**正数 x 的补码为正数本身$[x]_{补}$,负数的补码为$[y]_{补}=[模-|y|]_{补}$**。如图 5-15 所示,＋4 的补码为＋4,－1 的补码为＋9(10－|1|＝9)。

“模”是指数字计数范围,如时钟的计数范围是 0～12,模＝12。十进制数中,1 位数的模为 10,2 位数的模为 100,以此类推。下面是补码模运算案例。

【例 5-32】 $4+5\equiv[4]_{补}+[5]_{补}=[9]_{补} \bmod 10=9$(如图 5-15(a)所示)。

【例 5-33】 $6+7\equiv[6]_{补}+[7]_{补}=[13]_{补} \bmod 10=3$(如图 5-15(b)所示)。

【例 5-34】 $7-6\equiv[7]_{补}+[10-6]_{补}=[11]_{补} \bmod 10=1$(如图 5-15(c)所示)。

【例 5-35】 $6-7\equiv[6]_{补}+[10-7]_{补}=[9]_{补} \bmod 10=9$(如图 5-15(d)所示)。

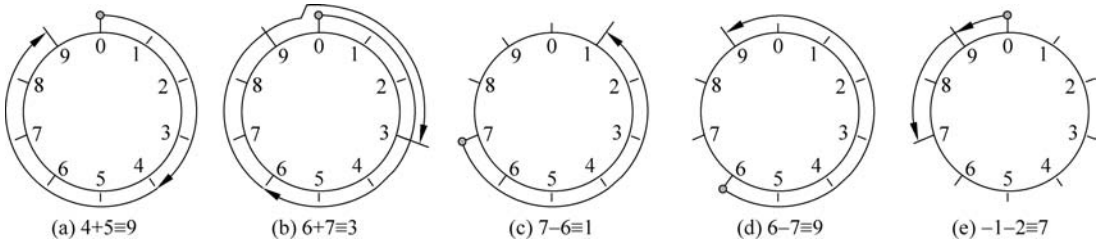


图 5-15　模＝10 的十进制数模运算示意图(顺时针方向为＋,逆时针方向为－)

【例 5-36】　$-1-2\equiv[10-1]_{补}+[10-2]_{补}=[17]_{补} \bmod 10=7$(如图 5-15(e)所示)。

【例 5-37】　补码运算的 Python 程序如下。

```
>>>(4 + 5) % 10                    # 输出为:9(正数的补码为正数本身,无须转换)
>>>(6 + 7) % 10                    # 输出为:3(% 为模运算符,Python 称为求余运算)
>>>(7 + (~6 + 1)) % 10             # 输出为:1(-6 的补码为绝对值 6 取反后 + 1)
>>>(6 + (~7 + 1)) % 10             # 输出为:9(~为取反运算符)
>>>((~1 + 1) + (~2 + 1)) % 10      # 输出为:7
```

由以上案例可见,**模运算可以将加法和减法运算都转换为补码的加法运算**。

4. 二进制数的补码编码方法

二进制正数的补码就是原码,负数的补码等于正数原码"取反加 1",即按位取反,末位加 1。负数的最高位(符号位)为 1,不管是原码、反码还是补码,符号位都不变。

【例 5-38】　10 的二进制原码为 $10=[00001010]_{原}$(最高位 0 表示正数)。

－10 的二进制原码为 $-10=[10001010]_{原}$(最高位 1 表示负数)。

【例 5-39】　10 的二进制反码为 $10=[00001010]_{反}$(最高位 0 表示正数)。

－10 的二进制反码为 $-10=[11110101]_{反}$(最高位 1 表示负数)。

【例 5-40】　10 的二进制补码为 $10=[00001010]_{补}$(最高位 0 表示正数)。

－10 的二进制补码为 $-10=[11110110]_{补}$(最高位 1 表示负数)。

5. 补码运算规则

补码运算的算法思想:把正数和负数都转换为补码形式,使减法变成加一个负数的补码形式,从而使加减法运算转换为单纯的补码加法运算。补码运算在逻辑电路设计中实现简单。当补码运算结果不超出表示范围(不溢出)时,可得出以下重要结论。

用补码表示的两数进行加法运算时,其结果仍为补码。补码的符号位可以与数值位一同参与运算。运算结果如有进位,则判断是否为"溢出",如果不是"溢出",就将进位舍去不要。不论对正数还是负数,补码都具有以下性质。

$$[A]_{补}+[B]_{补}=[A+B]_{补} \tag{5-3}$$

$$[[A]_{补}]_{补}=[A]_{原} \tag{5-4}$$

式中,A 和 B 为正整数、负整数、0 均可。

【例 5-41】　$A=-70$,$B=-55$,求 A 与 B 相加之和。

先将 A 和 B 转换为二进制数的补码,然后进行补码加法运算,最后将运算结果(补码)转换为原码即可。原码、反码、补码在转换中,要注意符号位不变的原则。

$-70=-(64+4+2)=[11000110]_{原}=[10111001]_{反}+[00000001]_2=[10111010]_{补}$

$-55=-(32+16+4+2+1)=[10110111]_{原}=[11001000]_{反}+[00000001]_2$

$=[11001001]_{补}$

$$\begin{array}{rrr} & & 10111010 \\ & + & 11001001 \\ \hline \text{没有溢出时,进位 1 自然丢失}\rightarrow & [1] & 10000011 \end{array}$$

相加后补码为$[10111010]_{补}+[11001001]_{补}=[10000011]_{补}$，进位[1]作为模丢失。为什么要丢弃进位呢？因为在加法器中，本位值与进位由不同逻辑电路实现(参见图 5-46)。

由补码运算结果再进行一次求补运算(取反加 1)就可以得到真值：

$$[10000011]_{补}=[11111100]_{反}+[00000001]_2=[11111101]_{原}=-125$$

通过以上案例可以看到，进行补码加法运算时，不用考虑数值的符号，直接进行补码加法即可。减法可以通过补码的加法运算实现。如果运算结果不产生溢出，且最高位(符号位)为 0，则表示结果为正数；如果最高位为 1，则结果为负数。

6. 补码运算的特征

补码设计的目的：一是使符号位能与有效值一起参加运算，从而简化运算规则；二是使减法运算转换为加法运算，进一步简化 CPU 中加法器的设计。

所有复杂计算(如线性方程组、矩阵、微积分等)都可以转换为四则运算，四则运算理论上都可以转换为补码的加法运算。**实际设计中，CPU 为了提高运算效率，乘法和除法采用了移位运算和加法运算。CPU 内部只有加法器，没有减法器，所有减法都采用补码加法进行。**程序编译时，编译器将数值进行了补码处理，并保存在计算机存储器中。补码运算完成后，计算机将运行结果转换为原码或十进制数据输出给用户。CPU 对补码完全不知情，它只按照编译器给出的机器指令进行运算，并对某些溢出标志位进行设置。

5.2 非数值信息编码

5.2.1 字符的早期编码

字符集是各种文字和符号的总称，它包括文字、符号、简单图符、数字等。字符集种类繁多，每个字符集包含的字符个数不同，编码方法不同。如 ASCII 字符集、GBK 字符集、Unicode 字符集等。计算机要处理各种字符集的文字，就需要对字符集中每个字符进行唯一性编码，以便计算机能够识别和存储各种文字。

1. 早期字符集编码标准

(1) ASCII 编码(1967 年)。ASCII 是英文字符编码规范。ASCII 用 7 位进行编码(00～7F)表示 128 个字符，其扩展编码使用 8 位表示(80～FF)，共可以表示 256 个字符，每个 ASCII 字符占 1 字节。ASCII 码是应用最广泛的编码。

(2) ANSI 编码。ANSI(American National Standards Institute，美国国家标准学会)编码与 ASCII 编码兼容，在 00～7F 的字符用 1 字节代表 1 字符(编码与 ASCII 相同)。对于汉字，ANSI 使用 80～FFFF 范围的 2 字节表示 1 个字符。在简体中文 Windows 系统下，

ANSI 代表 GBK 编码；在日文 Windows 系统下，ANSI 代表 JIS 编码；不同 ANSI 编码之间互不兼容。

(3) ISO 8859 编码(1998 年)。ISO 8859 是一系列欧洲字符集的编码标准(不包含汉字)，如 ISO 8859-1 标准收集了德法等西欧常用字符；ISO 8859-2 标准收集了中东欧字符，如克罗地亚语、捷克语、匈牙利语、波兰语、斯洛伐克语、斯洛文尼亚语等，英语也可用这个字符集显示。ISO 8859 标准采用 8 位编码，每个字符集最多可定义 95 个字符。ISO 8859 标准为了与 ASCII 码兼容，所有低位编码都与 ASCII 编码相同。

2. 中文字符集编码标准

(1) GB 2312 编码(1980 年)。GB 2312 是最早的国家标准中文字符集，它收录了 6763 个常用汉字和符号。GB 2312 采用定长 2 字节编码，其中 3755 个一级汉字按拼音序进行编码，3008 个二级汉字则按部首笔画序进行编码。

(2) GBK 编码(1995 年)。GBK 是 GB 2312 的扩展，它与 Unicode 编码兼容。GBK 加入了对繁体字的支持，收集了如計算機、囘、冊、円、冇等繁体字和生僻字。GBK 使用 2 字节定长编码，共收录 21 003 个汉字。

(3) GB 18030 编码(2000 年)。GB 18030 与 GB 2312 和 GBK 兼容。GB 18030 共收录 70 244 个汉字和符号，它采用变长多字节编码，每个汉字或符号由 1～4 字节组成。GB 18030 编码最多可定义 161 万个字符，支持国内少数民族文字、中日韩和繁体汉字等。Windows 7/8/10 默认支持 GB 18030 编码。

(4) 繁体中文 Big5 编码(2003 年)。Big5 是中国港澳台地区的繁体汉字编码。它对汉字采用 2 字节定长编码，共可表示 13 053 个中文繁体汉字。Big5 编码的汉字先按笔画再按部首进行排列。Big5 编码与 GB 类编码互不兼容。

【例 5-42】 如图 5-16 所示，中文字符在 UTF-8 编码中占 3 字节。

字符	GB 2312—80	GBK	BIG5	Unicode	UTF-16	UTF-8
中	D6 D0	D6 D0	A4 A4	4E 2D	4E 2D	E4 B8 AD
国	B9 FA	B9 FA	—	56 FD	56 FD	E5 9B BD

图 5-16　字符“中国”的各种编码形式(十六进制表示)

5.2.2　国际字符统一码 Unicode

1. Unicode 字符集和编码

(1) Unicode 字符集。Unicode(统一码)是一项国际字符集标准。Unicode 字符集的理论大小是 $2^{21}=2\ 097\ 152$ 个编码。Unicode 为全球每种语言的文字和符号都规定了一个唯一的二进制代码点和名称(简称码点)，如“汉”字的码点和名称是“U+6C49”。Unicode 码点通常用 2 字节表示一个字符，所以一个 Unicode 平面有 $2^{16}=65\ 535$ 个码点(理论值)。Unicode 目前规定了 17 种语言符号平面(110 万编码)。如 CJK(中日韩统一表意文字)平面收录了中文简体汉字、中文繁体汉字、日文假名、韩文谚文、越南喃字(大约 2 万个编码，不常用生僻字另有扩展字符集)。大多数字符集只有一种编码形式(如 ASCII、GB 2312 等)，Unicode 字符集有多种编码形式(如 UTF-8、UTF-16、UTF-32 等)。Unicode 字符集中汉字码点按《康熙字典》的偏旁部首和笔画数排列，编码排序与 GB(国标)字符集编码排序不同。

【例 5-43】 打印字符串的 Unicode 编码，Python 程序如下。

```
1  #E0543.py                                   # 【编码 - 打印 Unicode 码】
2  str = input('请输入一个字符串:')              # 输入字符串
3  a = [0] * len(str)                          # 计算字符串长度
4  i = 0                                       # 计数器初始化
5  for x in str:                               # 循环计算编码
6      a[i] = hex(ord(x))                      # 将字符 x 转换为 Unicode 编码
7      i = i+1                                 # 计数器累加
8  result = list(a)                            # 转换为列表
9  print("字符串的 Unicode 编码为:", result)    # 打印 Unicode 编码
   >>>请输入一个字符串:a 中国                   # 程序运行结果
   字符串的 Unicode 编码为: ['0x61', '0x4e2d', '0x56fd']   # 前缀 0x 表示十六进制编码
```

(2) UTF-8 编码。UTF-8 采用变长编码，128 个 ASCII 字符只需要 1 字节；带有变音符号的拉丁文、希腊文、西里尔字母、亚美尼亚语、希伯来文、阿拉伯文、叙利亚文等需要 2 字节；汉字为 3 字节；其他辅助平面符号为 4 字节。Python 3.x 程序采用 UTF-8 编码，因特网协议和浏览器等程序均采用 UTF-8 编码。

(3) UTF-16 编码。UTF-16 采用 2 或 4 字节变长编码。Windows 内核采用 UTF-16 编码，但支持 UTF-16 与 UTF-8 的自动转换。UTF-16 编码有 Big Endian(大端字节序)和 Little Endian(小端字节序)之分，UTF-8 编码没有字节序问题。

(4) UTF-32 编码。UTF-32 采用 4 字节编码，由于浪费存储空间，故很少使用。

2. UTF-8 编码方法

UTF-8 编码遵循了一个非常聪明的设计思想：**不要试图去修改那些没有坏或你认为不够好的东西，如果要修改，只去修改那些出问题的部分。**

UTF-8 采用变长编码，它用 1～4 字节表示一个字符。ASCII 码中每个字符的编码在 UTF-8 编码中保持完全一致，128 个 ASCII 字符只需要 1 字节编码(编码范围为 0000～007F)。带有附加符号的拉丁文、希腊文、西里尔字母、亚美尼亚语、希伯来文、阿拉伯文、叙利亚文等用 2 字节编码(编码范围为 0080～07FF)。汉字(包括中日韩三国使用的汉字)等字符使用 3 字节编码(编码范围为 4E00～9FFF)。其他极少使用的 Unicode 辅助字符使用 4 字节编码。

3. UTF-8 编码的优点

(1) UTF-8 编码中，128 个英文符号与 ASCII 编码完全一致，ASCII 编码无须任何改动。除 ASCII 编码外，其他字符都采用多字节编码，由于 ASCII 编码最高位为 0，多字节编码最高位为 1，所以不会产生编码冲突。

(2) UTF-8 从字符的首字节(编码第 1 个字节)编码就可以判断某个字符有几个字节。如果首字节以$[0]_2$开头，一定是单字节编码(即 ASCII 码)；如果首字节以$[110]_2$开头，一定是 2 字节编码(如一些欧洲文字)；如果首字节以$[1110]_2$开头，一定是 3 字节编码(如汉字)，其他以此类推。除单字节编码外，多字节 UTF-8 编码的后续字节均以$[10]_2$开头。UTF-8 变长编码的优点是节省存储空间，减少了数据传输时间。

【例 5-44】 UTF-8 汉字编码公式为$[\mathbf{1110}xxxx\ \mathbf{10}xxxxxx\ \mathbf{10}xxxxxx]_2$(x 表示 Unicode

码点中的 0 或 1)。如“中”字的 Unicode 码点为[01001110 00101101]$_2$(4E2D),将它填入 UTF-8 汉字编码公式,得到的 UTF-8 编码为[**1110**0100 **10**111000 **10**101101]$_2$(E4 B8 AD)。字符“中”的 UTF-8 编码方法如图 5-17 所示。

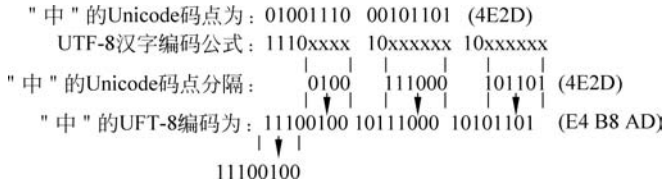


图 5-17　字符“中”的 UTF-8 编码方法

(3) UTF-16 采用定长 2 或 4 字节变长编码。UTF-16 的编码与 Unicode 中的码点相同(见图 5-16)。UTF-16 编码在存储时,2 个字节中哪个字节存高位?哪个字节存低位?这需要考虑字节序问题。UTF-8 多字节编码时,首字节高位与后续字节高位的编码不同,这本身就说明了字节序,所以不用再考虑字节序问题。

(4) UTF-8 具有编码错误快速恢复的优点。简单地说,程序可以从文件的任意部分开始读取数据。假设程序读到 1 个受损的字节,这只会使这个字符无法识别,并不会影响下一个字符的正确识别。因为 UTF-8 专门规定了字符首字节的编码格式,因此不会出现双字节编码(如 UTF-16、GBK 等)的字节错位问题,这有利于解决字符乱码问题。

5.2.3　音频数据编码

在计算机中,数值和字符都转换成二进制数来存储和处理。同样,声音、图形、视频、等信息也需要转换成二进制数后,计算机才能存储和处理。在计算机中,将模拟信号转换成二进制数的过程称为数字化处理。

1. 声音处理的数字化过程

声音是连续变化的模拟量。如对着话筒讲话时,如图 5-18(a)所示,话筒根据它周围空气压力的变化,输出连续变化的电压值。这种变化的电压值是对声音的模拟,称为模拟音频,如图 5-18(b)所示。要使计算机能存储和处理声音信号,就必须将模拟音频数字化。

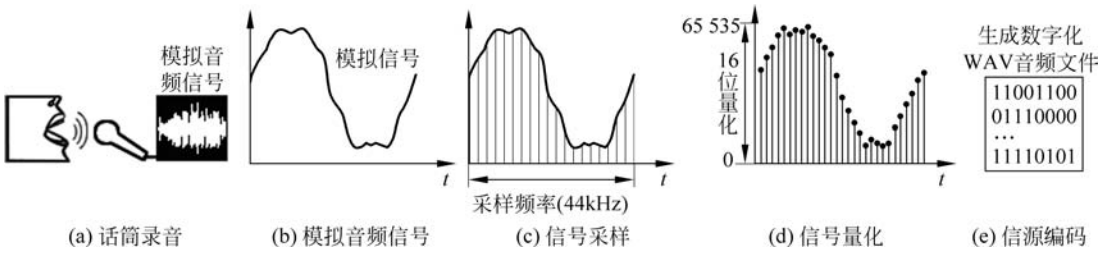


图 5-18　音频信号的数字化过程

(1) 采样。连续信号都可以表示成离散的数字序列。模拟信号转换成数字信号必须经过采样过程,采样是在固定的时间间隔内,对模拟信号截取一个振幅值,如图 5-18(c)所示,

并用定长的二进制数表示(如 16 位),将连续的模拟信号转换成离散的数字信号。**截取模拟信号振幅值的过程称为采样,所得到的振幅值为采样值**。单位时间内采样次数越多(采样频率越高),数字信号就越接近原声。奈奎斯特(Nyquist)采样定理指出:模拟信号离散化采样频率达到信号最高频率 2 倍时,可以无失真地恢复原信号。人耳听力范围在 20Hz～20kHz。声音采样频率达到 40kHz(每秒钟采集 4 万个数据)就可以满足信号采样要求,声卡采样频率一般为 44.1kHz 或更高。

(2) 量化。量化是将信号样本值截取为最接近原信号的整数值过程,例如,采样值是 16.2 就量化为 16,如果采样值是 16.7 就量化为 17。音频信号的量化精度(也称为采样位数)一般用二进制数位数衡量,如声卡量化位数为 16 位时,有 $2^{16}=65\ 535$ 种量化等级,如图 5-18(d)所示。目前声卡大多为 24 位或 32 位量化精度(采样位数)。音频信号采样量化时,一些系统的信号样本全部在正值区间,如图 5-18(d)所示,这时编码采用无符号数存储;另一些系统的样本有正值、0、负值(如正弦曲线),编码时用样本值最左边的位表示采样区间的正负符号,其余位表示样本绝对值。

(3) 编码。如果每秒钟采样速率为 S,量化精度为 B,它们的乘积为位率。例如,采样频率为 40kHz,量化精度为 16 位时,位率=40 000×16=640kbit/s。位率是信号采集的重要性能指标,如果位率过低,就会出现数据丢失的情况。数据采集后得到了一大批原始音频数据,对这些数据进行压缩编码(如 wav、mp3 等)后,再加上音频文件格式的头部,就得到了一个数字音频文件,如图 5-18(e)所示。这项工作由声卡和音频处理软件(如 Adobe Audition)共同完成。

2. 声音信号的输入与输出

数字音频信号可以通过网络、U 盘、数字话筒、电子琴乐器数字接口(MIDI)等设备输入计算机。模拟音频信号一般通过模拟信号话筒和音频输入接口(Line in)输入计算机,然后由声卡转换为数字音频信号,这一过程称为模/数(A/D)转换。需要将数字音频播放出来时,可以利用音频播放软件将数字音频文件解压缩,然后通过声卡或音频处理芯片,将离散数字量再转换成连续的模拟量信号(如电压或电流),这一过程称为数/模(D/A)转换。

5.2.4 点阵图像编码

1. 图像的数字化

数字图像(Image)可以由数码照相机等设备获取,设备对图像进行数字化处理,通过接口传输到计算机,并且以文件的形式存储在计算机中。当然,数字图像也可以直接在计算机中进行自动生成或人工设计,或由网络、U 盘等设备输入。

当计算机将图像数据输出到显示器、打印机等设备时,这些数据通过处理设备将离散化的数字整合为一幅自然图像。

2. 二值图像编码

只有黑、白两色的图像称为二值图像。图像信息是一个连续的变量,离散化的方法是设置合适的采样分辨率,然后对二值图像中的每个像素用 1 位二进制数表示,一般将黑色点编码为 1,白色点编码为 0(量化),这个过程称为数字化处理(见图 5-19)。

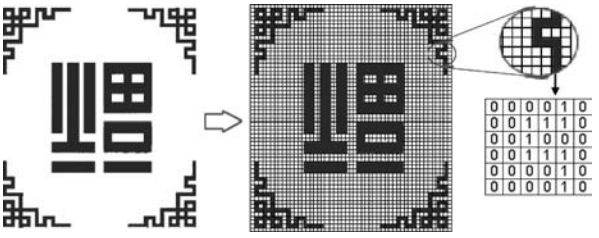


图 5-19　二值图像的数字化处理

图像分辨率是指单位长度内包含像素点的数量，分辨率单位有 dpi(点/英寸)等。图像分辨率为 1024×768 时，表示每条水平线上包含 1024 个像素点，垂直方向有 768 条线。分辨率不仅与图像的尺寸有关，还受到输出设备(如显示器点距等)等因素的影响。分辨率决定了图像细节的精细程度，图像分辨率越高，包含的像素就越多，图像越清晰。同时，太高的图像分辨率会增加文件处理时间和存储空间。

3. 灰度图像编码

灰度图像的数字化方法与二值图像相似，不同的是将白色与黑色之间的过渡灰色按对数关系分为若干亮度等级，然后对每个像素点按亮度等级进行量化。为了便于计算机存储和处理，一般将亮度分为 0～255 个等级(量化精度)，而人眼对图像亮度的识别小于 64 个等级，因此对 256 个亮度等级的图像，人眼难以识别出亮度的差别。灰度图像中每个像素点的亮度值用 8 位二进制数(1 字节)表示，如图 5-20 所示为灰度图像的编码方式。

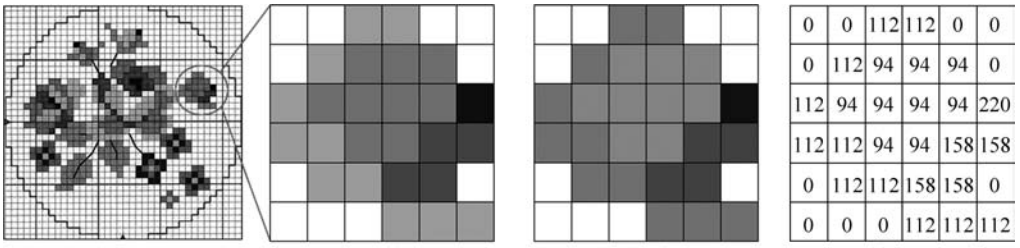


图 5-20　灰度图像的编码方式

4. 彩色图像编码

显示器上的任何色彩都可以用红绿蓝(RGB)三个基色按不同比例混合得到。RGB 的数值是指亮度，并用整数表示。如图 5-21 所示，红色用 1 字节表示，亮度范围为 0～255 个等级(如 R＝0～255)；绿色和蓝色也同样处理。

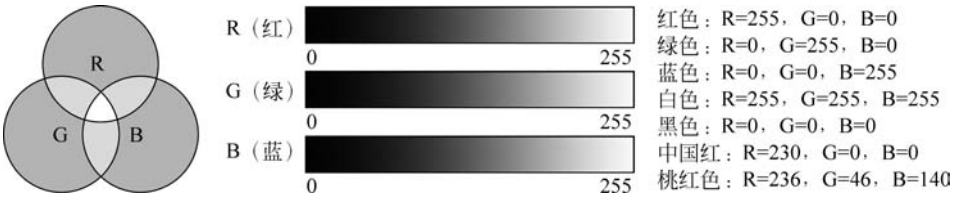


图 5-21　彩色图像的 RGB 编码方式

【例 5-45】 如图 5-21 所示，白色像素点的编码为 R＝255，G＝255，B＝255；黑色像素点的编码为 R＝0，G＝0，B＝0(亮度全部为 0)；红色像素点的编码为 R＝255，G＝0，B＝0；2008 年北京奥组委将中国红编码定义为 R＝230，G＝0，B＝0 等。

采用以上编码方式，一个像素点可以表达的色彩有 $2^{24}=1670$ 万种，这时人眼很难分辨出相邻两种颜色的区别。一个像素点用多少位二进制数表示，称为色彩深度(量化精度)，如上述案例中的色彩深度为 24 位。目前大部分显示器的色彩深度为 32 位，其中，8 位记录红色，8 位记录绿色，8 位记录蓝色，8 位记录透明度(Alpha)值，它们一起构成一个像素的显示效果。

【例 5-46】 对分辨率为 1024×768、色彩深度为 24 位的图片进行编码。

如图 5-22 所示，对图片中每个像素点进行色彩取值，其中某一个橙红色像素点的色彩值为 R＝233，G＝105，B＝66，如果不对图片进行压缩，则将以上色彩值进行二进制编码即可。形成图片文件时，还必须根据图片文件格式加上文件头部。

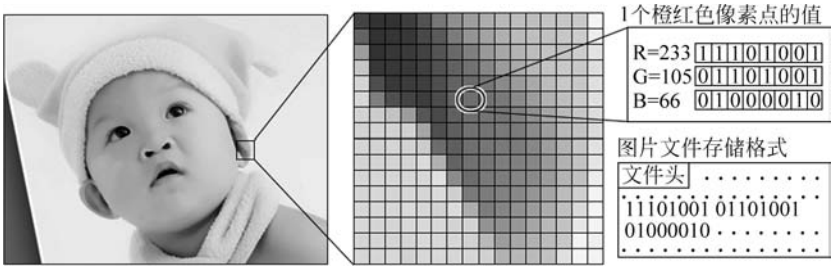


图 5-22　24 位色彩深度图像的编码方式(没有压缩时的编码)

5. 点阵图像的特点

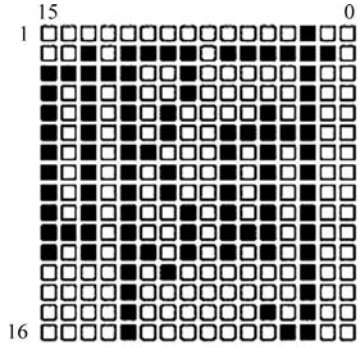
点阵图像由多个像素点组成，二值图、灰度图和彩色图都是点阵图像(也称为位图或光栅图)，简称为“图像”。图像放大时，可以看到构成整个图像的像素点，由于这些像素点非常小(取决于图像的分辨率)，因此图像的颜色和形状的连续性很好；一旦将图像放大观看，图像中的像素点会使线条和形状显得参差不齐。缩小图像尺寸时，也会使图像变形，因为缩小图像是通过减少像素点来使整个图像变小。

6. 点阵字体编码

ASCII 码和 Unicode 统一码主要解决了字符的存储、传输、计算、处理(录入、检索、排序等)等问题，而字符在显示和打印输出时，需要对“字形”进行再次编码。通常将字体(字形)编码的集合称为字库，字库以文件的形式存放在硬盘中，在字符输出(显示或打印)时，根据字符编码在字库中找到相应的字体编码，再输出到显示器或打印机中。汉字的风格有多种形式，如宋体、黑体、楷体等。由于字库没有统一的标准进行规定，同一字符在不同计算机中显示和打印时，可能字符形状会有所差异。

字体编码有点阵字体和矢量字体两种类型。点阵字体是将每个字符分成 16×16(或其他分辨率)的点阵图像，然后用图像点的有无(一般为黑白)表示字体的轮廓。点阵字体最大的缺点是放大后字符边缘会出现锯齿现象。

【例 5-47】 图 5-23 是字符“啊”的点阵图，每行用 2 字节表示，共用 16 行、32 字节来表达一个 16×16 点阵的汉字字体信息。



(a) 字符“啊”的点阵字体

00000000	00000100	=	0004
00101111	01111110	=	2F7E
11111001	00000100	=	F904
10101001	00010100	=	A914
10101010	00010100	=	AA14
10101010	01111100	=	AA7C
10101100	01010100	=	AC54
10101010	01010100	=	AA54
10101010	01010100	=	AA54
10101001	01010100	=	A954
11101001	01110100	=	E974
10101101	01010100	=	AD54
00001010	00000100	=	0904
00001000	00000100	=	0804
00001000	00010100	=	0814
00001000	00001100	=	080C

(b) 字符“啊”编码

图 5-23　点阵字体和编码

5.2.5　矢量图形编码

1. 矢量图形的编码

矢量图形(Graphic)使用直线或曲线来描述图形。**矢量图形采用特征点和计算公式对图形进行表示和存储**。矢量图形保存的是每个图形元件的描述信息,例如,一个图形元件的起始/终止坐标、特征点等。在显示或打印矢量图形时,要经过一系列的数学运算才能输出图形。矢量图形在理论上可以无限放大,图形轮廓仍然能保持圆滑。

如图 5-24 所示,矢量图形只记录生成图形的算法和图上的某些特征点参数。矢量图形中的曲线利用短直线插补逼近,通过图形处理软件,可以方便地将矢量图形放大、缩小、移动、旋转、变形等。矢量图形最大的优点是无论进行放大、缩小或旋转等操作,图形都不会失真、变色和模糊。由于构成矢量图形的各个图元相对独立,因而在矢量图形中可以只编辑修改其中某一个物体,而不会影响图形中的其他物体。

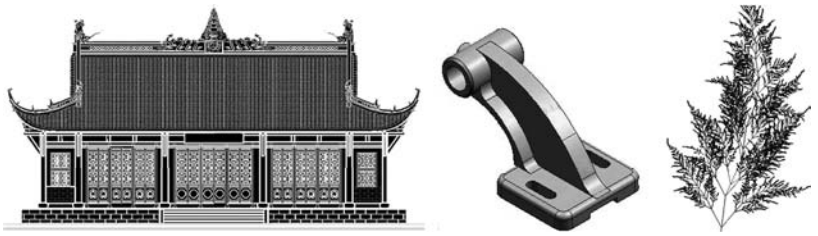


图 5-24　矢量图形

矢量图形只保存算法和特征点参数(如分形图),因此占用存储空间较小,打印输出和放大时图形质量较高。但是,矢量图形也存在以下缺点:一是难以表现色彩层次丰富的逼真图像效果;二是无法使用简单廉价的设备,将图形输入计算机中并矢量化;三是矢量图形目前没有统一的标准格式,大部分矢量图形格式存在不开放和知识产权问题,这造成了矢量图形在不同软件中进行交换的困难,也给图形设计带来了极大的不便。

矢量图形主要用于几何图形生成、工程制图、二维动画设计、三维物体造型、美术字体设计等。大多数绘图软件(如 Visio)、计算机辅助设计软件(如 AutoCAD)、三维造型软件(如 3ds Max)等,都采用矢量图形作为存储格式。矢量图形可以很好地转换为点阵图像,但是,

点阵图像转换为矢量图形时效果很差。

2. 矢量字体编码

矢量字体保存的是每个字体的数学描述信息，在显示和打印矢量字体时，要经过一系列的运算才能输出结果。矢量字体可以无限放大，笔画轮廓仍然保持圆滑。

字体绘制可以通过 FontConfig＋FreeType＋PanGo 三者协作来完成，其中 FontConfig 负责字体管理和配置；FreeType 负责单个字体的绘制；PanGo 则完成对文字的排版布局。

矢量字体有多种格式，其中 TrueType 字体的应用最为广泛。TrueType 是一种字体构造技术，要让字体在屏幕上显示，还需要字体驱动引擎，如 FreeType 就是一种高效的字体驱动引擎。FreeType 是一个字体函数库，它可以处理点阵字体和多种矢量字体。

如图 5-25 所示，矢量字体重要的特征是轮廓(outline)和字体精调(hint)控制点。轮廓是一组封闭的路径，它由线段或贝塞尔(Bézier)曲线(见图 5-26)组成。字形控制点有轮廓锚点和精调控制点，缩放这些点的坐标值将缩放整个字体轮廓。

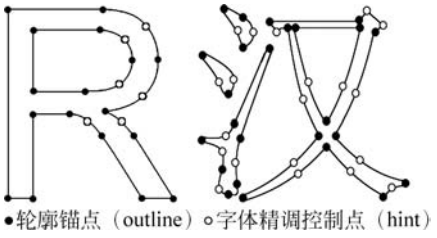


图 5-25　矢量字体轮廓和控制点

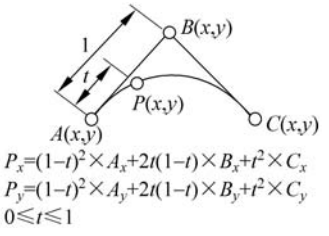


图 5-26　二次贝塞尔曲线示意图

轮廓虽然精确描述了字体的外观形状，但是数学上的正确对人眼来说并不见得合适。特别是字体缩小到较小的分辨率时，字体可能变得不好看，或者不清晰。字体精调就是采用一系列技术来精密调整字体，让字体变得更美观、更清晰。

矢量字体尽管可以任意缩放，但字体缩得太小时仍然存在问题。字体会变得不好看或者不清晰，即使采用字体精调技术，效果也不一定好，或者处理起来太麻烦。因此，显示大字体时多采用矢量字体，但是显示小字体(小于 8pt)时一般采用点阵字体。

矢量字体的显示大致需要经过以下步骤：加载字体→设置字体大小→加载字体数据→字体转换(旋转或缩放)→字体渲染(计算并绘制字体轮廓、填充色彩)等。可见在计算机显示一整屏文字时，计算工作量比我们想象的要大得多。

说明：绘制二次贝塞尔曲线案例，参见本书配套教学资源程序 F5-3.py。

5.3　压缩与纠错编码

5.3.1　信息熵的度量

1. 熵的概念和熵递增原理

1854 年，德国科学家克劳修斯(Rudolf Clausius)引入了熵(希腊语“变化”的意思)的概念。**熵是一个统计学意义上的概念**，熵可以衡量系统的混乱程度。例如，在一个充满气体的盒子里，可以测得气体的温度和压力，但是不知道盒子里每个气体粒子的位置和速度。盒子

里气体具有相同温度和压强的微观状态数越多，系统的熵就越大。

熵递增原理是自然界的基本规律，**熵递增原理指出：在一个孤立系统中（无外力作用时），当熵处于最小值时，系统处于最有序的状态；但熵会自发性地趋于增大，随着熵的增加，有序状态会逐步变为无序状态，并且不能自发地产生新的有序结构**。这就像懒人的房间，如果没有人替他收拾打扫（外力），房间只会杂乱下去，绝不会变得整洁。

热力学指出，在一个孤立的物理系统中，热熵不会随时间减小，只会增加。而信息熵则相反，在一个封闭系统中，信息熵只会减少，不会增加。信息熵表示不确定性，热熵反映混乱度，这点上两者有很大的一致性。

2. 信息的定义

信息是什么？维纳（Norbert Wiener）在《控制论》一书中指出：**信息就是信息，既非物质，也非能量**。这个定义对信息未作正面回答。香农绕过了这个难题，他基于概率统计，从计算的角度研究信息的量化。

3. 信息熵的计算

1948 年，香农发表了《通信的数学原理》论文，第一次将熵的概念引入信息论中。香农创造了 bit 这个单词，用于表示二进制的“位”。香农对信息的定义是：**信息是用来消除随机不确定性的东西**。香农提出，如果系统 S 内存在多个事件 $S=\{E_1, E_2, \cdots, E_n\}$，每个事件的概率分布为 $P=\{p_1, p_2, \cdots, p_n\}$，则事件的信息熵（Entropy）为

$$H(X)=-\sum_{i=1}^{n} p(x_i)\log_2 p(x_i) \tag{5-5}$$

式中，$H(X)$ 为信息熵，单位为 bit；$p(x_i)$ 是随机事件 x_i 的发生概率。对数底选择是任意的，使用 2 作为对数底只是遵循信息论的传统。公式前面的负号是为了确保信息熵是正数，或者说没有负信息熵。

4. 等概率事件对信息熵计算的简化

每个事件发生的概率相等（等概率事件）时，可以将信息熵计算过程简化。

【例 5-48】 向空中投掷硬币，硬币落地后有两个状态：一个是正面朝上，另一个是反面朝上，每个状态出现的概率为 1/2。如果是投掷正六面体的骰子，则可能出现的状态有 6 个，每个状态出现的概率均为 1/6。编程计算硬币的信息熵和骰子的信息熵，代码如下：

H（硬币）$=-(2\times 1/2)\times \log_2 p(1/2)=1$bit。

H（骰子）$=-(6\times 1/6)\times \log_2 p(1/6)\approx 2.6$bit。

```
>>> from math import log2                    # 导入标准模块 - 对数运算
>>> H1 = -(2*1/2) * log2(1/2)                # 计算硬币投掷的信息熵
>>> print('硬币投掷的信息熵 = ', H1)
硬币投掷的信息熵 = 1.0
>>> H2 = -(6*1/6) * log2(1/6)                # 计算骰子投掷的信息熵
>>> print('骰子投掷的信息熵 = ', H2)
骰子投掷的信息熵 = 2.584962500721156
```

【例 5-49】 π 是一个无限循环小数：3.1415926535…。如果希望 π 值的精度是十进制数的 30 位，编程计算最少需要多少 bit 二进制数才能满足要求？代码如下：

```
>>> from math import log2                    # 导入标准模块 - 对数运算
>>> H3 = -log2(10) / -log2(2)                # 计算十转二需要的信息熵 bit
>>> print('10 转 2 的信息熵 = ', H3)
10 转 2 的信息熵 = 3.321928094887362         # 1 位十进制数转为二进制数的信息熵
```

由以上计算可知，每位十进制数转换为二进制数时，信息熵大约需要 3.4bit(余量稍留大一点)。30 位十进制数转换成二进制数时，最少需要 30 位×3.4bit＝102bit。在实际应用中，由于存在编码方法、存储格式等要求，实际长度要大得多。

5. 事件发生概率不同的信息熵计算

在实际的情况中，每种可能情况出现的概率并不相同，所以必须用信息熵来衡量整个系统的平均信息量。因此，信息熵可以理解为信息的不确定性。

【例 5-50】 一个黑箱中有 10 个球，球有红、白两种颜色，但是不知道每种颜色的球各有多少个。黑箱中红、白球数量不同时，计算信息熵。Python 程序如下。

```
#E0550.py                                  #【编码 - 信息熵计算】
from math import log2                      # 导入标准模块 - 对数计算
for x in range(1, 10):                     # 循环计算信息熵
    p1 = x / 10                            # 计算红球概率
    p2 = 1 - p1                            # 计算白球概率
    n = [p1, p2]                           # 列表赋值
    H = -sum([p * log2(p) for p in n])     # 循环计算信息熵
    print(f'【{x}】红球概率:{p1:.2f};\      # 打印计算的信息熵
    白球概率:{p2:.2f};信息熵:{H:.2f}')      # 参数:.2f 为保留两位小数
```

```
>>>                                        # 程序运行结果
【1】红球概率:0.10; 白球概率:0.90;信息熵:0.47
【2】红球概率:0.20; 白球概率:0.80;信息熵:0.72
【3】红球概率:0.30; 白球概率:0.70;信息熵:0.88
【4】红球概率:0.40; 白球概率:0.60;信息熵:0.97
【5】红球概率:0.50; 白球概率:0.50;信息熵:1.00
【6】红球概率:0.60; 白球概率:0.40;信息熵:0.97
【7】红球概率:0.70; 白球概率:0.30;信息熵:0.88
【8】红球概率:0.80; 白球概率:0.20;信息熵:0.72
【9】红球概率:0.90; 白球概率:0.10;信息熵:0.47
```

通过程序计算可以发现，黑箱中红球越多，则抽到红球的概率越高，信息熵就越小，因为不确定性减少了。**越是确定的事件，不确定性越小，信息量也越少**。如果黑箱中全部是红球，则信息熵＝0，相当于没有任何信息。当黑箱中红白球都是 50%时，信息熵达到了最大值 1bit，这时难以确定会抽到哪个颜色的球。可见，**概率与信息熵成反比关系**，概率越高，信息熵越低；概率越低，信息熵越高。**信息熵与信息量成正比关系**。

6. 汉字的信息熵

1974 年，我国专家和科研部门对中文海量文本进行了统计，发现使用频率最高的 3755 个汉字对各类文本的覆盖率达到了 99.9%。因此在 GB 2312—1980 标准中，将这 3755 个汉字规定为一级汉字库。

【例 5-51】 假设 3755 个汉字在文本中出现的概率相同，计算汉字的平均信息熵，代码如下。

```
>>> from math import log2                                  # 导入标准模块 - 对数计算
>>> H3 = -(3755 * 1/3755) * log2(1/3755)                  # 计算汉字的平均信息熵
>>> print('一级汉字的信息熵 = ', H3)
一级汉字的信息熵 = 11.874597192401634
```

实际文本语料中，汉字的使用频率并不相同，如汉字出现频率最高的字是“的”，出现频率是 4.1%。**统计表明，常用语言单个字母的平均信息熵为汉语 9.65bit、英语 4.03bit、法语 3.98bit**。汉语的平均词长为 1.94 左右(词组)；英文单词的平均词长为 5 个字母左右，因此英语单词的信息熵要大于汉字(参见例 5-52)。

7. 信息熵与编码长度

香农信息论的特点是：**信息熵只考虑信息本身出现的概率，与信息内容无关。简单地说，多个信息熵的累加和称为信息量**。信息论中的信息量与日常语境中的信息量是不同的概念。日常语境的信息量往往是指信息的质量和信息表达的效率。或者说，**日常语境中的信息量与信息内容相关；而信息熵与信息内容无关，只与字符的编码长度有关**。

【例 5-52】 S_1 =“秋高气爽”(4)，S_2 =“秋天晴空万里，天气凉爽”(11)，S_3 = The autumn sky is clear and the air is crisp(44)。计算它们的信息量(信息熵之和)。

$H_1 = 9.65 \times 4 = 38.6$bit；$H_2 = 9.65 \times 11 = 106.15$bit；$H_3 = 4.03 \times 44 = 177.32$bit。

在日常语境中，S_1、S_2、S_3 三个短语表达了同一个语义，它们的信息量大致相同。但是，在信息熵计算中，$S2$ 的信息熵比 S_1 高 1 倍以上，S_3 的信息熵是 S_1 的 4 倍以上，也就是说，在同一语义下，S_2 和 S_3 的编码长度要大大高于 S_1。

5.3.2 无损压缩编码

可以通过数据压缩来消除多媒体信息中原始数据的冗余性。在保证信息质量的前提下，压缩比(压缩比＝压缩前数据的长度：压缩后数据的长度)越高越好。

1. 无损压缩的特征

无损压缩的基本原理是相同的信息只需要保存一次。例如，一幅蓝天白云的图像压缩时，首先会确定图像中哪些区域是相同的，哪些是不同的。蓝天中数据重复的图像就可以压缩，只需要记录同一颜色区域的起始点和终止点。从本质上看，无损压缩可以删除一些重复数据，大大减少图像的存储容量。

无损压缩的优点是可以完全恢复原始数据，而不引起任何数据失真。无损压缩算法一般可将文件压缩到原来的 1/2～1/4(压缩比为 2：1～4：1)。但是，无损压缩并不会减少数据占用的内存空间，因为读取压缩文件时，软件会对丢失的数据进行恢复填充。

2. 常用压缩编码算法

常用压缩编码算法如图 5-27 所示。

LZW(Lempel、Ziv 和 Welch 三人共同发明的算法)、LZ77(Lempel 和 Ziv 于 1977 年发明的算法)、LZSS(由 LZ77 改进的算法)编码属于字典模型的压缩算法，而 RLE(Run

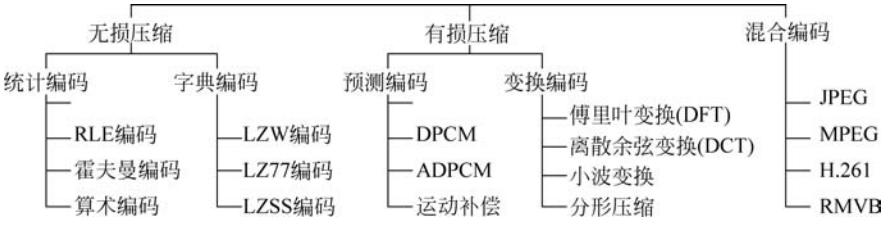


图 5-27　常用压缩编码算法

Length Encoding，行程长度编码）、霍夫曼编码和算术编码都属于统计模型压缩算法。字典模型压缩算法与原始数据的排列顺序有关，而与数据的出现频率无关；统计模型压缩算法正好相反，它与数据的排列顺序无关，而与数据的出现频率有关。这两类压缩算法各有所长，相互补充。许多压缩软件结合了这两类算法。

3. RLE 压缩编码算法原理

RLE（行程长度编码，也称为游程编码）是对重复的数据序列用重复次数和单个数据值来代替，重复的次数称为“游程”。RLE 是一种变长编码，RLE 用一个特殊标记字节指示重复字符开始，非重复节可以有任意长度而不被标记字节打断，标记字节是字符串中不出现的符号（或出现最少的符号，如@）。RLE 编码方法如图 5-29 所示。

标记字节	字符重复次数	字符

图 5-28　RLE 编码方法

【例 5-53】 对字符串 AAAAABACCCCBCCC 进行 RLE 编码。

编码为@5A@1B@1A@4C@1B@3C。标记字节@说明字符开始，数字表示字符重复次数，后面是被重复字符。Python 程序如下。

```
1  #E0553.py                                                    #【编码 - 游程编码】
2  line = input('请输入原始数据:')                               # 输入数据
3  count = 1                                                    # 计数器初始化
4  print('数据 REL 编码为:', end = "")                           # 打印提示信息
5  for i in range(1, len(line)):                                # 循环读取字符串
6      if line[i] == line[i - 1]:                               # 如果前后两个字符相同
7          count += 1                                           # 计数器 + 1
8      else:
9          print("(" + str(count) + "," + line[i - 1] + "),",end="")  # 打印重复次数和字符
10         count = 1                                            # 计数器初始化
11 print("(" + str(count) + "," + line[-1] + ")")               # 打印全部编码
```

```
>>>                                                             # 程序运行结果
请输入原始数据:AAAAABACCCCBCCC
数据 REL 编码为:(5,A),(1,B),(1,A),(4,C),(1,B),(3,C)
```

RLE 编码算法简单直观，编码和解码速度快，压缩效率低，适用于字符重复出现概率高的情况。许多图形、视频文件都会用到 RLE 编码算法。

4. 霍夫曼压缩编码算法原理

霍夫曼(David A. Huffman)编码算法的基本原理是：频繁使用的字符用短编码代替，较少使用的字符用长编码代替，每个字符的编码各不相同(变长编码)。例如，字母 e、t、a 的使用频率要大于字母 z、q、x，可以用短编码表示常用字母 e、t、a，用长编码表示不常用字母 z、q、x，这样每个字母都有唯一的编码。编码的长度可变，而不是像 ASCII 码那样都用 8 位表示。霍夫曼编码广泛应用于数据压缩、数据通信、多媒体技术等领域。

【例 5-54】 对字符串 I am a teacher 进行霍夫曼编码。

设信号源字符串为 $X=$｛空格，a，e，I，m，t，c，h，r｝(按字符出现概率的大小进行排列)。假设每个字符对应的概率为 $p=$｛0.22，0.22，0.14，0.07，0.07，0.07，0.07，0.07，0.07｝，霍夫曼编码过程如图 5-29 所示。霍夫曼编码算法的步骤如下。

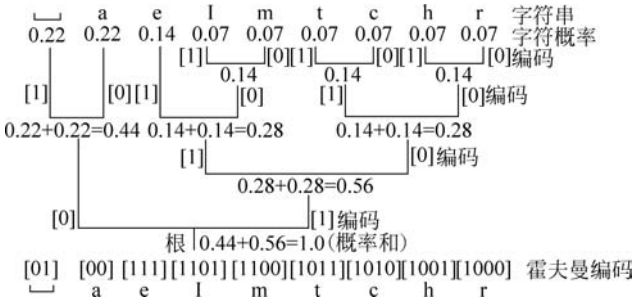


图 5-29 字符串 I am a teacher 的霍夫曼编码过程

(1) 如图 5-29 所示，按字符在文本中出现概率的大小从左到右顺序排列。

(2) 将概率相同的两个符号组成一个节点，如空格和 a、I 和 m、t 和 c 等。

(3) 将两个相邻概率组合相加，形成一个新概率(如 0.22＋0.22＝0.44)，将新出现的概率与未编码的字符一起重新排序(如 0.44 与 0.56)。

(4) 重复步骤(2)～步骤(3)，直到出现的概率和为 1(如根节点)，形成“霍夫曼树”。

(5) 编码分配。编码从根节点(树底部)开始向上分配(如空格编码＝01)。代码左边标 1 或 0 无关紧要，因为结果仅仅是编码不同，而编码平均长度相同。

字符串 I am a teacher 的霍夫曼编码如图 5-30 所示。

压缩文本	I		a	m		a		t	e	a	c	h	e	r
霍夫曼编码	1101	01	00	1100	01	00	01	1011	111	00	1010	1001	111	1000

图 5-30 字符串的霍夫曼编码(二进制)

说明：霍夫曼编码的案例，参见本书配套教学资源程序 F5-4.py。

5. 字典压缩编码算法

(1) 日常生活中的字典编码。人们日常生活中经常使用字典压缩方法。例如，奥运会、PC 等词汇，说者和听者都明白它们是指奥林匹克运动会、个人计算机，这就是信息压缩。人们之所以能使用这种压缩方式而不产生语义上的误解，是因为在说者和听者心目中都有一个事先定义好的缩略语字典，人们在对信息进行说(压缩)和听(解压缩)过程中，都对字典进行了查询操作。字典压缩编码就是基于这一计算思维设计的。

(2) 字典压缩编码算法思想。字典压缩编码就是把文本中的符号做成一个字典列表，并用特殊代码(如数字)来表示这个符号。字典压缩编码算法是创建一个短语字典，编码过程中，如果遇到字典中出现的短语时，就输出字典中短语的索引号，而不是短语本身。LZ77和LZW编码均采用这种算法思想。

(3) LZW压缩编码算法。LZW压缩编码算法由Lemple、Ziv、Welch三人共同创造，并用他们的名字的首字母命名。LZW压缩编码算法中，首先建立一个字典(字符串表)，把每个第一次出现的字符串放入字典中，并用一个数字标号来表示(一般在7位ASCII码上进行扩展)，压缩文件只存储数字标号，不存储字符串。这个数字标号与此字符串在字典中的位置有关，这个字符串再次出现时，可用字典中的数字标号来代替，并将这个数字标号存入压缩文件中，压缩完成后将字典丢弃。解压缩时，可根据压缩数据重新生成字典。

【例5-55】 对文本"good good study，day day up"进行LZW压缩编码。

对原始文本中的字符串进行扫描，生成的字典如图5-31所示。为易于理解，下面的数字标号没有采用扩展ASCII码，而采用顺序数字编码表示。

字符串	g	o	d		good	s	t	u	y	,	a	day	p	。	…
数字标号	1	2	3	4	5	6	7	8	9	10	11	12	13	14	…

图5-31 LZW压缩编码算法扫描文本后生成的字典

利用LZW压缩编码算法，对文本信息进行压缩的最终编码如图5-32所示。

压缩文本	good		good		study	,	day		day		up	。
压缩编码	1223	4	5	4	67839	10	3119	4	12	4	813	14

图5-32 利用LZW压缩编码算法对文本信息进行压缩的编码

说明：对字符串进行LZW压缩编码的案例，参见本书配套教学资源程序F5-5.py。

如图5-32所示，随着新字符串不断被发现，字典的数字标号也会不断地增长。如果字符数量过大，生成的数字标号也会越来越大，这时就会产生编码效率问题。如何避免这个问题呢？GIF格式图像压缩采用的方法是：当数字标号"足够大"时，就干脆从头开始进行数字标号，并且在字典这个位置插入一个清除标志(CLEAR)，表示从这里重新开始构造字典，以前所有数字标号作废，开始使用新数字标号。问题是"足够大"是多大？理论上数字标号集越大，压缩比越高，但系统开销也越高。LZW压缩编码算法的字典一般在7位ASCII字符集上进行扩展，扩展后的数字标号可用12位(4096个编码)甚至更多位表示。

LZW压缩编码算法适用情况是原始数据中有大量子字符串重复出现，重复越多，压缩效果越好，反之则越差。一般情况下LZW压缩编码算法可实现2∶1～3∶1的压缩比。

目前流行的GIF、TIF等格式图像文件采用了LZW压缩编码算法；WinRAR、WinZip等压缩软件也采用了LZW压缩编码算法，微软公司CAB压缩文件也采用了LZW编码机制。甚至许多硬件(如网络设备)中，也采用了LZW压缩编码算法。LZW压缩编码算法广泛用于文本数据和特殊应用领域的图像数据(如指纹图像、医学图像等)压缩。

【例 5-56】 将 d:\test\temp 子目录下所有文件进行压缩，Python 程序如下。

```
# E0556.py                                                    # 【编码 - 文件压缩】
import zipfile                                                # 导入标准模块 - 压缩
import os                                                     # 导入标准模块 - 系统
f = zipfile.ZipFile('my.zip', 'w', zipfile.ZIP_DEFLATED)      # my.zip 为压缩文件名,w 为写入
startdir = 'd:\\test\\temp'                                   # ZIP_DEFLATED 为压缩,f 为变量
for dirpath, dirnames, filenames in os.walk(startdir):        # 读取目录下所有文件
    for filename in filenames:                                # 将压缩文件循环写入临时变量
        f.write(os.path.join(dirpath, filename))              # 将临时变量循环写入压缩文件
f.close( )                                                    # 关闭文件
```

5.3.3 有损压缩技术

1. 有损压缩的特征

经过有损压缩的对象进行数据重构时，重构后的数据与原始数据不完全一致，是一种不可逆的压缩方式。如图像、视频、音频数据的压缩就可以采用有损压缩，因为对象中包含的数据往往多于人们视觉系统和听觉系统所能接收的信息，有损压缩是丢掉一些数据，并且不会对声音或者图像表达的意思产生误解，但可以大大提高压缩比。图像、视频、音频数据的压缩比高达 10∶1～50∶1，可以大大减少数据在内存和磁盘中的占用空间。因此多媒体信息编码技术主要侧重于有损压缩编码的研究。

有损压缩的算法思想是：通过有意丢弃一些对视听效果相对不太重要的细节数据进行信息压缩。这种压缩方法一般不会严重影响视听质量。

2. JPEG 图像压缩标准

国际标准化组织(International Organization for Standardization，ISO)和国际电信联盟(ITU)共同成立的联合图像专家组(Joint Photographic Experts Group，JPEG)，于 1991 年提出了“多灰度静止图像的数字压缩编码”(简称 JPEG 标准)。这个标准适用于彩色和灰度图像的压缩处理。JPEG(读[J-peg])标准包含两部分：第一部分是无损压缩，采用差分脉冲编码调制(Differential Pulse Code Modulation，DPCM)的预测编码；第二部分是有损压缩，采用离散余弦变换(DCT)和霍夫曼编码。

JPEG 算法的思想是：恢复图像时不重建原始画面，而是生成与原始画面类似的图像，丢掉那些没有被注意到的颜色。JPEG 压缩利用了人的心理和生理特征，因而非常适合真实图像的压缩。对于非真实图像(如线条图、卡通图像等)，JPEG 压缩效果并不理想。JPEG 对图像的压缩比为一般为 10∶1～25∶1。

3. JPEG 图像压缩编码原理

JPEG 图像压缩编码原理非常复杂，如图 5-33 所示，压缩编码分为以下四个步骤。

(1) 颜色模型转换及采样。JPEG 采用 YCbCr(Y 亮度，Cb 色度，Cr 饱和度)色彩模型，因此需要将 RGB(红绿蓝)色彩的图像数据转换为 YCbCr 色彩的数据。

(2) DCT(离散余弦变换)。将图像数据转换为频率系数(原理复杂，此处不详述)。

(3) 量化。将频率系数由浮点数转换为整数。

(4) 熵编码。其编码原理和过程更加复杂(此处不详述)。

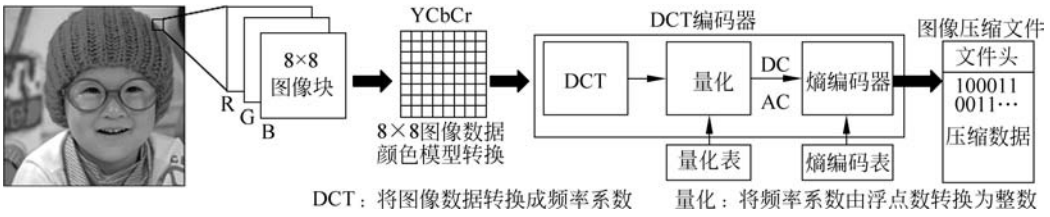


图 5-33　JPEG 图像压缩编码流程

4. MPEG 动态图像压缩标准

对计算技术而言视频是随空间(图像)和时间(一系列图像)变化的信息。视频图像由很多幅静止画面(称为帧)组成,如果采用 JPEG 压缩算法,对每帧图像画面进行压缩,然后将所有图像帧组合成一个视频图像,这种计算思维方法可行吗?

【例 5-57】 一幅分辨率为 640×480 的彩色静止图像,没有压缩时的理论大小为(640×480×24bit)/8=900KB,假设经过 JPEG 压缩后大小为 50KB(压缩比 18∶1)。按 JPEG 算法压缩一部 120min 的影片,则影片文件大小为 50KB×30fps(帧/秒)×60s×120min=10.3GB。显然,完全采用 JPEG 算法压缩视频图像时,还是存在文件太大的问题。

运动图像专家组开发了视频图像的数据编码和解码标准 MPEG(读[M-peg])。目前已经开发的 MPEG 标准有 MPEG-1、MPEG-2、MPEG-4 等。MPEG 算法除了对单幅视频图像进行编码压缩外(帧内压缩),还利用图像之间的相关特性,消除了视频画面之间的图像冗余,大大提高了数字视频图像的压缩比,MPEG-2 的压缩比可达到 20∶1～50∶1。

5. MPEG 压缩编码算法原理

MPEG 压缩编码基于运动补偿和离散余弦变换(DCT)算法。**MPEG 算法思想是:在一帧图像内(空间方向),数据压缩采用 JPEG 算法来消除冗余信息;在连续多帧图像之间(时间方向),数据压缩采用运动补偿算法来消除冗余信息。**

计算机视频的播放速率为 30fps,也就是 1/30s 显示一幅画面,在这么短的时间内,相邻两个画面之间的变化非常小,相邻帧之间存在极大的数据冗余。在视频图像中,可以利用前一帧图像来预测后一帧图像,以实现数据压缩。帧间预测编码技术广泛应用于 H.261、H.263、MPEG-1、MPEG-2 等视频压缩标准。

图像帧之间的预测编码有以下方法。

(1) 隔帧传输。对于静止图像或活动很慢的图像,可以少传输一些帧(如隔帧传输);没有传输的图像帧则利用帧缓存中前一图像帧的数据作为该图像帧的数据。

(2) 帧间预测。不直接传送当前图像帧的像素值,而是传送画面中像素 X_1 与前一画面(或后一画面)同一位置像素 X_2 之间的数据差值。

【例 5-58】 在一段太阳升起的视频中,第 n 帧图像中,太阳中 X_1 点的像素值为 $R=200$(橙红色);在 $n+1$ 帧图像中,该点 X_2 的像素值改变为 $R=205$(稍微深一点的橙红色)。这时可以只传输两个像素之间的差值 5,图像中没有变化的像素无须传输数据。

(3) 运动补偿预测。如图 5-34(a)所示,视频图像中一个画面大致分为三个区域:一是背景区(相邻两个画面的背景区基本相同);二是运动物体区(可以视为由前一个画面某一区域的像素移动而成);三是暴露区(物体移动后显露出来曾被遮盖的背景区域),如图 5-34(b)所示。运动补偿预测就是将前一个画面的背景区+平移后的运动物体区作为

后一个画面的预测值。

(a) 视频画面的初始帧（第1帧）　(b) 视频画面的结束帧（第30帧）

图 5-34　视频画面的三个基本区域

6. MPEG 视频图像排列

如图 5-35 所示，MPEG 标准将图像分为三种类型：帧内图像 I 帧(关键帧)、预测图像 P 帧(预测计算图像)和双向预测图像 B 帧(插值计算图像)。

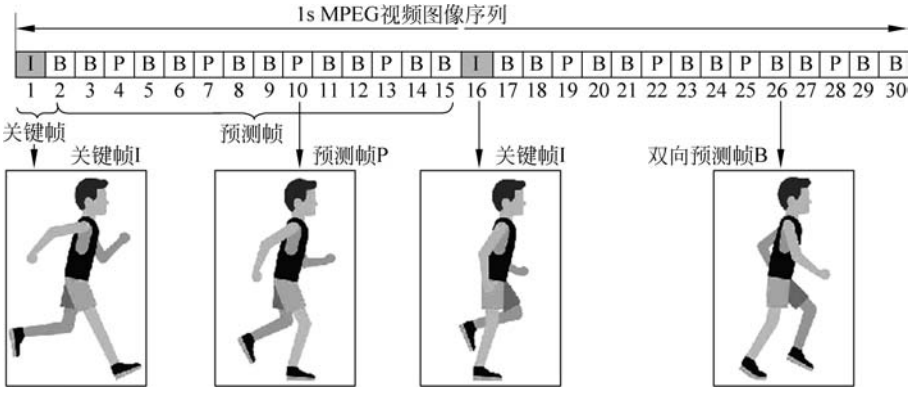


图 5-35　典型 MPEG 视频图像帧显示序列

(1) 关键帧 I。I 帧包含内容完整的图像，它用于其他图像帧的编码和解码参考，因此称为关键帧。I 帧采用类似 JPEG 的压缩算法，压缩比较低。I 帧图像可作为 B 帧和 P 帧图像的预测参考帧。I 帧周期性出现在视频图像序列中，出现频率可由编码器选择，一般为 2 次/s(2Hz)。对于高速运动的视频图像，I 帧的出现频率可以选择高一些，B 帧可以少一些；对于慢速运动的视频图像，I 帧出现的频率可以低一些，B 帧图像可以多一些。

(2) 预测帧 P。P 帧是利用相邻图像帧统计信息进行预测的图像帧。也就是说 P 帧和 B 帧采用相对编码，即不对整个画面帧进行编码，只对本帧与前一帧画面不同的地方编码。P 帧采用运动补偿算法，所以 P 帧图像的压缩比相对较高。由于 P 帧是预测计算的图像，因此计算量非常大，在高分辨率视频中计算量很大，对计算机显卡的要求高。

(3) 双向插值帧 B。B 帧是双向预测插值帧，它利用 I 帧和 P 帧作参考，进行运动补偿预测计算。B 帧不能用来作为对其他帧进行运动补偿预测的参考帧。

(4) 视频图像帧序列。典型的 MPEG 视频图像帧序列如图 5-35 所示，在 1s 时间里，30 帧画面有 28 帧需要进行运动补偿预测计算，可见视频图像的计算量非常大。

5.3.4 信号纠错编码

1. 信道差错控制编码

数据传输要通过各种物理信道，由于电磁干扰、设备故障等因素的影响，传送的信号可能发生失真，使信息遭到损坏，造成接收端信号误判。为了提高信号传输的准确性，除了提高信道抗噪声干扰外，还必须在通信中采用信道差错控制编码。采用信道差错控制编码的目的是提高信号传输的可靠性，改善通信系统的传输质量。

差错控制编码是数据在发送之前，按照某种关系附加一个校验码后再发送。接收端收到信号后，检查信息位与校验码之间的关系，确定传输过程中是否有差错发生。**差错控制编码提高了通信系统的可靠性，但它以降低通信系统的效率为代价**。

差错控制方法有两种：一种是 ARQ(Automatic Repeat-reQuest，自动重传请求)；另一种是 FEC(Forward Error Correction，前向纠错)。纠错码使用硬件实现时，速度比软件快几个数量级；纠错码使用软件实现不需要另外增加设备，特别适用于网络通信。绝大多数情况下，计算机使用检错码，出现错误后自动请求对方重发(ARQ)；只有在单工通信情况下，才会使用纠错编码。

2. 出错重传的差错控制方法

ARQ 采用出错重传的方法。如图 5-36 所示，在发送端对数据进行检错编码，通过信道传送到接收端，接收端经过译码处理后，只检测数据有无差错，并不自动纠正差错。如果接收端检测到接收的数据有错误时，则利用信道传送反馈信号，请求发送端重新发送有错误的数据，直到收到正确数据为止。ARQ 通信方式要求发送方设置一个数据缓冲区，用于存放已发送出去的数据，以便出现差错后，可以调出数据缓冲区的内容重新发送。在计算机通信中，大部分通信协议采用 ARQ 差错控制方式。

图 5-36　ARQ 差错控制方式

3. 奇偶校验方法

奇偶校验是一种最基本的检错码，它分为奇校验和偶校验。**奇偶校验可以发现数据传输错误，但是它不能纠正数据错误**。

【例 5-59】 字符 A 的 ASCII 码为$[01000001]_2$，其中有两位码元值为 1。如果采用奇校验编码，由于这个字符的编码中有偶数个 1，所以校验位的值为 1，其 8 位组合编码为$[10000011]_2$，前 7 位是信息位，最低位是奇校验码。同理，如果采用偶校验，可知校验位的值为 0，其 8 位组合编码为$[10000010]_2$。接收端对 8 位编码中 1 的个数进行检验时，如有不符，就可以判定传输中发生了差错。

如果通信前，通信双方约定采用奇校验码，接收端对传输数据进行校验时，如果接收到编码中 1 的个数为奇数时，则认为传输正确；否则就认为传输中出现了差错。在传输中有偶数个比特位(如 2 位)出现差错时，这种方法就不再适用。所以，奇偶校验只能检测出信息

中出现的奇数个错误，如果出错码元为偶数个，则奇偶校验不能奏效。

奇偶校验容易实现，而且一个字符(8 位)中 2 位同时发生错误的概率非常小，所以信道干扰不大时，奇偶校验的检错效果很好。计算机广泛采用奇偶校验进行检错。

4. 一个简单的前向纠错编码案例

在前向纠错(FEC)通信中，发送端在发送前对原始信息进行差错编码，然后发送。接收端对收到的编码进行译码后，检测有无差错。接收端不但能发现差错，而且能确定码元发生错误的位置，从而加以自动纠正。前向纠错不需要请求发送方重发信息，发送端也不需要存放以备重发的数据缓冲区。虽然前向纠错有以上优点，但是纠错码比检错码需要使用更多的冗余数据位。也就是说编码效率低，纠错电路复杂。因此，大多应用在单向传输或实时要求特别高的领域。例如，地球与火星之间距离太远，火星探测器“机遇号”的信号传输一个来回差不多要 20min，这使得信号的前向纠错非常重要。

计算机常用的前向纠错编码是汉明码(R. W. Hamming)，可以利用汉明码进行检测并纠错。下面用一个简单的例子来说明纠错的基本原理，它虽然不是汉明码，但是它们的算法思想相同，都是利用冗余编码来达到纠错的目的。

【例 5-60】 如图 5-37(a)所示，发送端 A 将字符 OK 传送给接收端 B，字符 OK 的 ASCII 码＝$[79,75]_{10}$。如果接收端 B 通过奇偶校验发现数据 D2 在传输过程中发生了错误，最简单的处理方法就是通知发送端重新传送出错数据 D2，但是这样会降低传输效率。

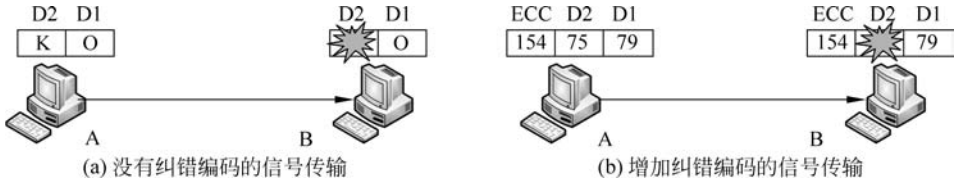


图 5-37 信号传输出错和出错校正编码示意图

如图 5-37(b)所示，如果发送端将两个原始数据相加，得出一个错误校验码 ECC(ECC＝D1＋D2＝79＋75＝154)，然后将原始数据 D1、D2 和校验码 ECC 一起传送到接收端。接收端通过奇偶校验检查没有发现错误，就丢弃校验码 ECC；如果接收端通过奇偶校验发现数据 D2 出错了，就可以利用校验码 ECC 减去另外一个正确的原始数据 D1，这样就可以得到正确的原始数据 D2(D2＝ECC－D1＝154－79＝75)，不需要发送端重传数据。

5. CRC 编码原理

(1) CRC 编码特征。信号在通信线路上串行传送时，通常会引发多位数据发生错误。这种情况下，奇偶校验和汉明码校验的作用就再明显，这时需要采用 CRC(Cyclic Redundancy Check，循环冗余校验)编码。CRC 编码是一种最常用的差错校验码，它的特点是检错能力极强(可检测多位出错)，开销小(开销小于奇偶校验编码)，易于用逻辑电路实现。在数据存储和数据通信领域，CRC 编码无处不在。例如，以太网、WinRAR 压缩软件等采用了 CRC-32 编码；磁盘文件采用了 CRC-16 编码；GIF 等格式图像文件也采用 CRC 编码作为检错手段。

(2) CRC 算法思想。根据选定的“生成多项式”生成 CRC 编码；在待发送的数据帧后面附加上 CRC 编码，生成一个新数据帧(源数据＋CRC 编码)发送给接收端。数据帧到达接收端后，接收方将数据帧与生成多项式进行模运算(mod)，如果结果不为 0，则说明数据

在传输过程中出现了差错。CRC 校验的工作原理如图 5-38 所示。

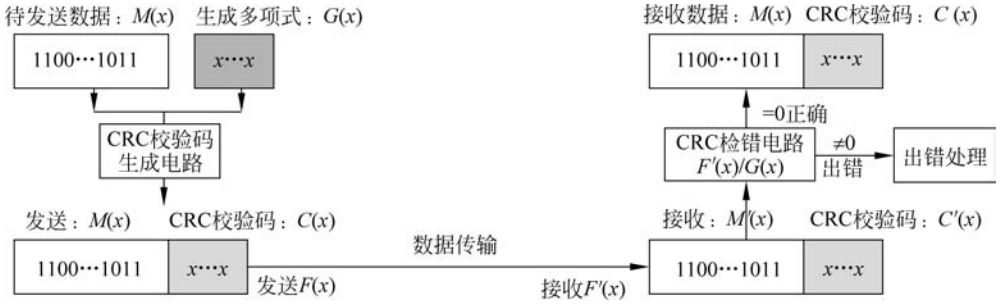


图 5-38　CRC 校验的工作原理

（3）发送方 CRC 编码过程。读取原始数据块→将该数据块左移 n 位后除以生成多项式 $G(x)$→将余数作为 CRC 码附在数据块最后面（原始数据块＋CRC 编码）→传送增加 CRC 码的数据块。

（4）数据接收方 CRC 校验过程。接收数据块→将接收数据块与生成多项式 $G(x)$进行模运算→如果余数为 0 则接收数据正确，余数不为 0 则说明数据传输错误。

6. 生成多项式

常用的 CRC 生成多项式国际标准如表 5-5 所示。生成多项式的最高位和最低位系数必须为 1，为了简化表示生成多项式，一般只列出二进制值为 1 的位，其他位为 0。生成多项式按二进制数最高阶数记为 CRC-m。如 CRC-8、CRC-16 等。

表 5-5　常用 CRC 生成多项式国际标准

标准名称	生成多项式	简记式*	生成多项式展开	应用案例
CRC-4-ITU	x^4+x+1	0x3	1 0011	ITU-G. 704
CRC-8-ITU	x^8+x^2+x+1	0x07	1 0000 0111	HEC
CRC-16	$x^{16}+x^{15}+x^2+1$	0x8005	1 10000000 00000101	美国标准
CRC-16-ITU	$x^{16}+x^{12}+x^5+1$	0x1021	1 00010000 00100001	欧洲标准
CRC-32	$x^{32}+x^{26}+x^{23}+\cdots+x^2+x+1$	0x04c11db7	1 0000…111	以太网、RAR

说明：* 生成多项式的最高幂次系数固定为 1，在简记式中，通常将最高位的 1 去掉。如 CRC-8-ITU 的简记式 0x07 实际上是 0x107，对应的二进制码为 0x07＝107H＝$[1\ 0000\ 0111]_2$。

【例 5-61】 码组$[1100101]_2$ 可以表示为 $1\times x^6+1\times x^5+0\times x^4+0\times x^3+1\times x^2+0\times x+1$，为了简化表达式，生成多项式省略了码组中为 0 的部分，即记为 $G(x)=x^6+x^5+x^2+1$。

7. 程序设计案例：生成 CRC 编码

【例 5-62】 生成字符串'hello，word!'的 CRC 检验编码，Python 程序如下。

```
1  >>> import zlib                                  # 导入标准模块 - 解包
2  >>> s = b'hello,word!'                           # 字符串赋值,b 说明字符串为 byte 编码
3  >>> print(hex(zlib.crc32(s)))                    # 生成 CRC - 32 编码,打印十六进制 CRC 编码
   0xb4e7eee9                                       # 前缀 0x 表示十六进制数
4  >>> print(bin(zlib.crc32(s)))                    # 生成 CRC - 32 编码,打印二进制 CRC 编码
   0b10110100111001111110111011101001               # 前缀 0b 表示二进制数
```

说明：生成 CRC-16 编码的案例，参见本书配套教学资源程序 F5-6.py。

8. CRC 编码的特征

CRC 编码只能检查错误，不能纠正错误，但是 CRC 能检查 3 位以上的错误，CRC 校验的好坏取决于选定的生成多项式。CRC 编码虽然可以用软件实现，但是大部分时候采用硬件电路实现（如网卡和主板中的 CRC 校验电路）。CRC 编码具有以下特征。

（1）源数据长度任意，校验码长度取决于选用的生成多项式。

（2）信号任一位发生错误时，生成多项式做模运算后余数不为 0。

（3）信号不同位发生错误时，生成多项式做模运算后余数不为 0。

（4）生成多项式做模运算后的余数不为 0 时，继续做模运算，余数会循环出现。

5.3.5 信道传输编码

信道传输编码就像商品的包装，商品包装的目的是使商品更适合运输，在运输过程中不受损。同样，信道编码的目的是使编码后的二进制数据更适合信道的传输。

1. 直流平衡

1s 对人类来说是一个短暂的时间，但是在 USB 3.0 总线中，信号传输频率为 5GHz，这意味在 1s 时间内可以传输 50 亿个（5Gbit/s）信号，1s 对计算机来说是一个漫长的时间。在数据传输过程中，如果长时间（如 100 个时钟周期）内没有信号出现（称为“直流不平衡”），会造成以下问题：一是不能确定传输的数据一直为 1 或为 0，还是数据传输系统出现了问题；二是长时间（如 100 个时钟周期）为低电平（为 0），偶尔出现一个高电平（为 1）时，系统不能确定这个信号是有效信号还是干扰信号；三是高速串行传输采用了同步传输技术，数据发送端将时钟信号与源数据组合成数据帧进行传输，接收端需要从接收的数据帧恢复时钟信号来保证同步，这需要线路中传输的二进制码流有足够多的跳变，即不能有过多连续的高电平，也不能有过多连续的低电平。

直流平衡是数据传输中 1 和 0 数量保持大致相同的一种技术。实现直流平衡的方法有 8B-10B 编码、64B-66B 编码、扰频等编码技术。

2. 8B-10B 编码技术

（1）8B-10B 编码的应用。8B-10B 信道编码技术广泛应用于高速串行接口，如 USB（Universal Serial Bus，通用串行总线）、SAS（Serial Attached SCSI，串行 SCSI）接口、SATA（Serial Advanced Technology Attachment，硬盘串行传输）接口、PCI-E 总线、光纤链路、GbE（吉比特以太网）、XAUI（10G 比特接口）、InfiniBand 总线、Serial Rapid IO 总线等。高速串行接口的 8B-10B 编码主要采用专用集成电路（ASIC）芯片实现，这些高速接口主要包括三个组成部分：电路部分（串行/解串行）、物理层部分（数据编码）、链路与协议部分（高层）。

（2）8B-10B 编码的原理。8B-10B 编码的原理是将 8 位二进制数字编码成 10 位，它包括 256 个数据编码和 12 个控制编码。8B 的含义是将 8 位数据分成两部分：低 5 位进行 5B-6B 编码，高 3 位则进行 3B-4B 编码，这两种映射关系可以从标准化表格中查到。人们习惯把 8B 数据表示成 Dx.y 的形式，其中 x＝5LSB（最低有效位），y＝3MSB（最高有效位）。

【例 5-63】 一个 8B 的源数据为$[10110101]_2$，其中 $x=[10101]_2$（低 5 位，十进制数＝21），$y=[101]_2$（高 3 位，十进制数＝5），一般将这个源数据写成 D21.5 的形式。

(3) 8B-10B 编码的技术特征。**8B-10B 编码的特性是保证直流平衡，它可以实现发送 0、1 的数量保持基本一致，连续的 1 或 0 不超过 5 位**，即每 5 个连续的 1 或 0 后，必然会插入一位 0 或 1。这些特性确保了 0 码元与 1 码元个数一致(直流平衡)；确保了字节同步易于实现(即在一个比特流中易于找到字节的起始位)；以及对误码率有足够的容错能力；降低硬件电路设计的复杂度等。

(4) 8B-10B 编码的缺点。8B-10B 编码的缺点只有一个，就是高达 25%的系统开销。人们提出过一些降低 8B-10B 编码技术系统开销的改进方法，如 64B-66B 编码技术(用于 10G 以太网，开销大约 3%)；10GBase-KR(用于 10GbE 背板连接)；CEI-P 编码技术；Interlaken PHY 编码技术等。这些技术的共同点都是以提高硬件设计复杂度(芯电晶体管数量)为代价，换取较低的编码开销。到目前为止，还没有哪种低开销的编码技术能够脱颖而出，成为继 8B-10B 编码之后广泛采用的首选信道编码技术。

5.4 数理逻辑与应用

5.4.1 数理逻辑概述

德国数学家莱布尼茨首先提出了用演算符号表示逻辑语言的思想；英国数学家乔治·布尔(George Boole)1847 年创立了布尔代数；美国科学家香农将布尔代数用于分析电话开关电路。布尔代数为计算机逻辑电路设计提供了理论基础。

逻辑学最早由古希腊学者亚里士多德创建。莱布尼茨曾经设想过创造一种“通用的科学语言”，将推理过程像数学一样利用公式进行计算。数理逻辑既是数学的一个分支，也是逻辑学的一个分支。数理逻辑是用数学方法研究形式逻辑的学科，所谓数学方法包括使用符号和公式，以及形式化、公理化等数学方法。通俗地说，**数理逻辑就是对逻辑概念进行符号化后(形式化)，对证明过程用数学方法进行演算**。

数理逻辑的研究包括古典数理逻辑(命题逻辑和谓词逻辑)和现代数理逻辑(递归论、模型论、公理集合论、证明论等)。递归论主要研究可计算性理论，模型论主要研究形式语言(如程序语言)与数学之间的关系，公理化集合论主要研究集合论中无矛盾性的问题。数理逻辑和计算科学有许多重合之处，两者都属于模拟人类认知机理的科学。许多计算科学的先驱者既是数学家，又是数理逻辑学家，如阿兰·图灵、布尔等。数理逻辑与计算技术的发展有着密切联系，它为机器证明、自动程序设计、计算机辅助设计等应用和理论提供必要的理论基础。例如，程序语言学、语义学的研究从模型论衍生而来，而程序验证则从模型论的模型检测衍生而来。柯里·霍华德(Haskell Brooks Curry-William Alvin Howard)同构理论说明了“证明”和“程序”的等价性。例如，LISP、Prolog 就是典型的逻辑推理程序语言，程序执行过程就是逻辑推理的证明过程(参见 2.3.4 节)。

现代数理逻辑证明，最基本的逻辑状态只有两个：一个是从一种状态变为另一种状态(如从 0 转变为 1 的逻辑开关)；另一个是在两种状态中，按照某种规则(如比较大小)选择其中的一种状态(如实现 if 语句功能的逻辑比较器)。根据这两种逻辑状态，可以构成多状态的任意逻辑关系。例如，在 CPU 内部，**算术运算都可以用“加法和比较大小”两个运算关系联合表达**，减法可以通过补码的加法来实现。

5.4.2 基本逻辑运算

1. 逻辑运算的特点

布尔代数中，逻辑变量和逻辑值只有 0 和 1，它们不表示数值的大小，只表示事物的性质或状态。如命题判断中的“真”与“假”，数字电路中的“低电平”与“高电平”等。

布尔代数有三种最基本的逻辑运算：与运算(AND)、或运算(OR)、非运算(NOT)。通过这三个基本逻辑运算可组合出任何其他逻辑关系。逻辑运算与算术运算的规则大致相同，但是逻辑运算与算术运算存在以下区别。

(1) 逻辑运算是一种位运算，按规则逐位运算即可。

(2) 逻辑运算中不同位之间没有任何关系，当然也就不存在进位或借位问题。

(3) 逻辑运算由于没有进位，因此不存在溢出问题。

(4) 逻辑运算没有符号问题，逻辑值在计算机中以原码形式表示和存储。

(5) 真值表是描述逻辑关系的直观表格。

逻辑运算的这些特性，特别适用于集成电路芯片中逻辑电路的设计。

2. 与运算

与运算(AND)相当于逻辑的乘法运算，它的逻辑表达式为 $Y=A \cdot B$。**与运算规则是：全 1 为 1，否则为 0(全真为真)**。用电子元件制造的与运算器件称为“与门”，与运算规则、真值表和与门表示符号如图 5-39 所示(GB 为国家标准)。与运算常用于存储单元的清零，如某个存储单元和一个各位为 0 的数相与时，存储单元结果为零。

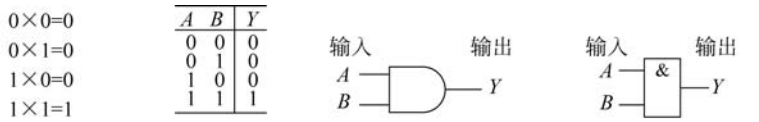


图 5-39 与运算规则和与门表示符号

【例 5-64】 10011100 AND 00000000 = 00000000。

	10011100	(输入 A)
AND	00000000	(输入 B)
	00000000	(输出 Y)

3. 或运算

或运算(OR)相当于逻辑的加法运算，它的逻辑表达式为 $Y=A+B$。**或运算规则是：全 0 为 0，否则为 1(全假为假)**。用电子元件制造的或运算器件称为“或门”，或运算规则、真值表和或门表示符号如图 5-40 所示。或运算常用于对某位存储单元置位(存储单元设置为全 1)运算。

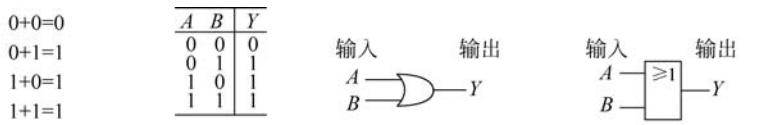


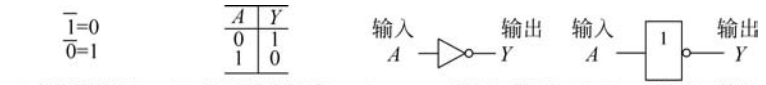
图 5-40 或运算规则和或门表示符号

【例 5-65】 10011100 OR 00111001 = 10111101。

	10011100	（输入 A）
OR	00111001	（输入 B）
	10111101	（输出 Y）

4. 非运算

非运算(NOT)是逻辑值取反运算，逻辑表达式为 $Y=\overline{A}$，非运算是对逻辑值“取反”，规定在逻辑运算符上方加上画线“—”表示。用电子元件制造的非运算器件称为“非门”，非运算规则、真值表和非门表示符号如图 5-41 所示。在逻辑门符号中，一般用“小圆圈”表示逻辑值取反。非运算常用于对某个二进制位进行取反运算；或者逻辑电路中的与非门、或非门电路设计。



(a) 非运算规则 (b) 非运算真值表 (c) IEEE “非门” 符号 (d) GB “非门” 符号

图 5-41 非运算规则和非门表示符号

【例 5-66】 NOT(10011100) = 01100011。

NOT	10011100	（输入 A）
	01100011	（输出 Y）

5. 异或运算

异或(XOR)是一种应用广泛的逻辑运算，异或运算用符号“⊕”表示，运算规则为 0⊕0=0，0⊕1=1，1⊕0=1，1⊕1=0。**异或运算可以理解为：相同为 0，相异为 1(同假异真)**。异或运算真值表和异或门符号如图 5-42 所示。异或也称为半加运算，运算法则相当于不带进位的二进制加法，所以异或运算常用于半加器逻辑电路设计(参见图 5-45)。

(a) 异或运算真值表 (b) IEEE “异或” 符号 (c) GB “异或” 符号

图 5-42 异或运算真值表和异或门表示符号

【例 5-67】 10011100 XOR 00111001 = 10100101。

	10011100	（输入 A）
XOR	00111001	（输入 B）
	10100101	（输出 Y）

5.4.3 命题逻辑演算

1. 数理逻辑中的命题

数理逻辑主要研究推理过程，而推理过程必须依靠命题来表达，**命题是能够判断真假的陈述句**。命题判断只有两种结论：命题结论为真或为假，命题真值通常用大写英文字母 T(True)或 F(False)表示。表达单一意义的命题称为“原子命题”，原子命题可通过“联结词”

构成“复合命题”。论述一个命题为真或为假的过程称为证明。

【例 5-68】 下列陈述句都是命题。

(1) 8 小于 10——命题真值为真。

(2) 8 大于 10——假命题也是命题,是真值为假的命题。

(3) 一个自然数不是合数就是素数——命题真值为假,1 不是合数和素数。

(4) 明年 10 月 1 日是晴天——真值目前不知道,但真值是确定的(真或假)。

(5) 公元 1100 年元旦下雨——可能无法查明真值,但真值是确定的(真或假)。

【例 5-69】 下列语句不是命题。

(1) 8 大于 10 吗? ——疑问句,非陈述性句,不是命题。

(2) 天空多漂亮! ——感叹句,非陈述性句,不是命题。

(3) 禁止喧哗——命令句,非陈述性句,不是命题。

(4) $x>10$——x 值不确定,因此命题真值不确定。

(5) $c^2=a^2+b^2$——方程不是命题。

(6) 这句话是谎言——悖论不是命题。

2. 逻辑联结词的含义

命题演算是研究命题如何通过一些逻辑联结词构成更复杂的命题。由简单命题组成复合命题的过程,可以看作逻辑运算的过程,也就是命题的演算。

数理逻辑运算也和代数运算一样,满足一定的运算规律。例如,满足数学的交换律、结合律、分配律; 同时也满足逻辑上的同一律、吸收律、双否定律、德摩根定律、三段论定律等。利用这些定律可以进行逻辑推理,简化复合命题。在数理逻辑中,与其说注重的是论证本身,不如说注重的是论证的形式。

将命题用合适的符号表示称为命题符号化(形式化)。数理逻辑规定了用逻辑联结词表示命题的推理规则,如表 5-6 所示。使用联结词可以将若干简单句组合成复合句。

表 5-6 逻辑联结词与含义

联结词符号	说 明	命题案例	命 题 读 法	命 题 含 义
¬	非(NOT)	¬P	非 P	P 的否定(逻辑取反)
∧	与(AND)	P∧Q	P 并且 Q	P 和 Q 的合取(逻辑乘)
∨	或(OR)	P∨Q	P 或者 Q	P 和 Q 的析取(逻辑加)
→	如果…则…(if-then)	P→Q	若 P 则 Q	P 蕴含 Q(单向条件)
↔	当且仅当	P↔Q	P 当且仅当 Q	P 等价 Q(双向条件)

命题的真值可以用图表来说明,这种表称为真值表,如表 5-7 所示。

表 5-7 逻辑运算真值表

命题前件	命题后件	不同逻辑命题运算的真值					
P	Q	P∧Q	P∨Q	¬P	¬Q	P→Q	P↔Q
0	0	0	0	1	1	1	1
0	1	0	1	1	0	1	0
1	0	0	1	0	1	0	0
1	1	1	1	0	0	1	1

3. 联结词的优先级

(1) 联结词的优先级按“否定→合取→析取→蕴含→等价”由高到低。

(2) 同级的联结词,优先级按出现的先后次序(从左到右)排列。

(3) 运算要求与优先次序不一致时,可使用括号,括号中的运算优先级最高。

联结词连接的是两个命题真值之间的联结,而不是命题内容之间的联结,因此命题的真值只取决于原子命题的真值,与它们的内容无关。

4. 逻辑联结词“蕴含”(→)的理解

(1) 前提与结论。日常生活中,命题的前提和结论之间包含有某种因果关系,但在数理逻辑中,允许前提和结论之间无因果关系,只要可以判断逻辑值的真假即可。

【例 5-70】 侯宝林先生相声中讲到:如果关羽叫阵,则秦琼迎战。

显然,这个命题在日常生活中是荒谬的,因为他们之间没有因果关系。但是在数理逻辑中,设 P=关羽叫阵,Q=秦琼迎战,命题:P→Q 成立。

(2) 善意推定。日常生活中,当前件为假时(P=0),P→Q 没有实际意义,因为整个语句的意义往往无法判断,故人们只考虑 P=1 的情形。但在数理逻辑中,当前件 P 为假时(P=0),无论后件 Q 为真或假(Q=0 或 Q=1),P→Q 总为真命题(即 P→Q=1),这有没有道理呢?

【例 5-71】 李逵对戴宗说:“我去酒肆一定帮你带壶酒回来。”

可以将这句话表述为命题 P→Q(P=李逵去酒肆,Q=带壶酒回来)。后来李逵因有事没有去酒肆(即 P=0),但是按数理逻辑规定(见表 5-7),命题 P→Q 为真(即李逵带壶酒回来了),这合理吗?我们应理解为李逵讲了真话,即:他要是去酒肆,我们相信他一定会带壶酒回来。这种理解在数理逻辑中称为“善意推定”,因为前件不成立时,很难区分前件与后件之间是否有因果关系,只能做善意推定。

5. 命题逻辑的演算

【例 5-72】 将下列命题用逻辑符号表示。

(1) 命题:今天会下雨或不上课。

令:P=今天下雨,Q=今天不上课,符号化命题为 P∨Q。

命题演算 1:如果 P=1(真),Q=0(假);则 P∨Q=1∨0=1(命题为真)。

命题演算 2:如果 P=1(真),Q=1(真);则 P∨Q=1∨1=1(命题为真)。

命题演算 3:如果 P=0(假),Q=0(假);则 P∨Q=0∨0=0(命题为假)。

命题演算 4:如果 P=0(假),Q=1(真);则 P∨Q=0∨0=1(命题为真)。

(2) 命题:mm 既漂亮又会做饭。

令:P=mm 漂亮,Q=mm 会做饭,符号化命题为 P∧Q。

(3) 命题:骑白马的不一定是王子。

令:P=骑白马的一定是王子,符号化命题为 ¬P。

(4) 命题:若 $f(x)$是可以微分的,则 $f(x)$是连续的。

令:P=$f(x)$是可以微分的,Q=$f(x)$是连续的,符号化命题为 P→Q。

(5) 命题:只有在老鼠灭绝时,猫才会哭老鼠。

令:P=猫哭老鼠,Q=老鼠灭绝,符号化命题为 P↔Q。

(6) 命题：铁和氧化合，但铁和氮不化合。

令：P＝铁和氧化合，Q＝铁和氮化合，符号化命题为 P∧(¬Q)。

(7) 命题：如果上午不下雨，我就去看电影，否则我就在家里看书。

令：P＝不下雨看电影，Q＝在家看书，符号化命题为¬P→Q。

(8) 命题：人不犯我，我不犯人；人若犯我，我必犯人。

令：P＝人犯我，Q＝我犯人，符号化命题为(¬P→ ¬Q)∧(P→Q)。

说明：求逻辑命题真值表的案例，参见本书配套教学资源程序 F5-7.py。

【例 5-73】 某地发生了一起谋杀案，警察通过排查确定凶手必为以下四个嫌疑犯之一。以下为嫌疑犯供词：A 说不是我；B 说是 C；C 说是 D；D 说 C 在胡说。假设其中有三人说了真话，一个人说了假话。请问谁是凶手？Python 程序如下。

```
#E0573.py                                                    #【编码 - 逻辑推理】
for m in['A', 'B', 'C', 'D']:                                #  循环对每个条件进行判断
    if (m != 'A') + (m == 'C') + (m == 'D') + (m != 'D') == 3:  #  根据条件,设置命题逻辑
        print(f'{m}是凶手')                                    #  满足以上条件的为凶手
>>> C 是凶手                                                 #  程序运行结果
```

5.4.4 谓词逻辑演算

1. 命题函数

命题逻辑不能充分表达计算科学中的许多陈述，如“n 是一个素数”就不能用命题逻辑来描述。因为它的真假取决于 n 的值，当 $n=5$ 时，命题为真；当 $n=6$ 时，命题为假。

程序中常见的语句如“x＞5”“x＝y＋2”等，当变量值未知时，这些语句既不为真，也不为假；但是当变量为确定值时，它们就成为了一个命题。

【例 5-74】 在语句“x＞5”中包含了两部分：第一部分变量 x 是语句的主语，第二部分是谓词“大于 5”。用“$p(x)$”表示语句“x＞5”，其中 p 表示谓词“＞5”，而 x 是变量。一旦将变量 x 赋值，$p(x)$就成为了一个命题函数。例如，当 $x=0$ 时，变量有了确定的值，因此命题 $p(0)$为假；当 $x=8$ 时，命题 $p(8)$为真。

【例 5-75】 令 $G(x, y)$表示“x 高于 y”，而 $G(x, y)$是一个二元谓词(两个变量)。如果将“张飞”代入 x，“李逵”代入 y，则 G(张飞，李逵)就是命题“张飞高于李逵”。可见在 x、y 中代入确定的个体后，$G(x,y)$命题函数就可以形成一个具体的命题。

2. 个体和谓词

原子命题是在结构上不能再分解出其他命题的命题(也称为简单命题)，如：今天是星期日等。原子命题中不能有与、或、非、如果、那么等联结词。

原子命题由个体词和谓词两部分组成。在谓词公式 $P(x)$中，P 称为谓词，x 称为变量(或个体变元)。

个体是独立存在的物体，它可以是抽象的，也可以是具体的。如人、学生、桌子、自然数等都可以作为个体，个体也可以是抽象的常量、变量、函数。在谓词演算中，个体通常在命题里表示思维对象。

表示个体之间关系的词称为谓词。表示一个个体的性质(一元关系)称为一元谓词；表

示 n 个个体之间的关系称为 n 元谓词。如“x 是素数”是一元谓词；“大于”是二元谓词；“在……之间”是三元谓词，如“x 大于 a，小于 b”就是三元谓词。

【例 5-76】 “$x=y+2$”表示了两个变量之间的关系（二元谓词），其中谓词是“( )=( )+2”。当 $x=7, y=5$ 时，命题函数 $p(x, y)$ 表示 $7=5+2$，命题 $p(7, 5)$ 为真；当 $x=2, y=3$ 时，命题函数 $p(x, y)$ 表示 $2=3+2$，命题 $p(2, 3)$ 为假。

3. 谓词逻辑的符号表示

在谓词逻辑的形式化中，一般使用以下符号进行表示。

(1) 常量符号一般用 a、b、c…表示，它可以是集合中的某个元素。

(2) 变量符号一般用 x、y、z…表示，集合中任意元素都可代入变量符号。

(3) 函数符号一般用 f、g…表示，n 元函数表示为 $f(x_1, x_2, \cdots, x_n)$。

(4) 谓词符号一般用 P、Q、R…表示，n 元谓词表示为 $P(x_1, x_2, \cdots, x_n)$。

4. 量词

量词是命题中表示数量的词，它分为全称量词和存在量词。例如，在“所有阔叶植物是落叶植物”“有些水生动物用肺呼吸”这两个命题中，“所有”“有些”都是量词，其中“所有”是全称量词，“有些”是存在量词。

汉语中，“所有”“一切”等表示全称量词；“有些”“至少有一个”等表示存在量词。全称量词一般用符号“$\forall x$”表示，读作“对任一 x”，或“所有 x”。存在量词一般用符号“$\exists x$”表示，读作“有些 x”，或“存在一个 x”。在一个公式前面加上量词，称为量化式，如 $(\forall x)F(x)$ 称为全称量化式，表示“对所有 x，x 就是 F”（即一切事物都是 F）。$(\exists x)F(x)$ 称为存在量化式，表示“有一个 x，x 就是 F”（即有一个事物是 F）。在谓词逻辑中，命题符号化必须明确个体域，无特别说明时，默认为全总个体域，一般使用全称量词。

5. 命题公式

用量词（$\forall$、$\exists$）和命题联结词（$\wedge$、$\vee$、$\rightarrow$等）可以构造出各种复杂的命题公式。例如，“5 是素数”“7 大于 3”这两个原子命题的公式为 $F(x)$ 和 $G(x, y)$。例如，陈述 n 个个体间有某关系的原子命题，可用公式 $F(x_1, x_2, \cdots, x_n)$ 表示。谓词逻辑中，命题的谓词逻辑公式有无穷多个。

【例 5-77】 用谓词逻辑公式表示以下命题。

(1) 命题：所有自然数乘 0 都等于 0（皮亚诺公理中乘法的定义）。

谓词逻辑公式：$\forall m, m\times 0=0$。

(2) 命题：有的水生动物是用肺呼吸的。

谓词逻辑公式：$(\exists x)(F(x)\wedge G(x))$。

(3) 命题：一切自然数都有大于它的自然数。

谓词逻辑公式：$(\forall x)(F(x)\rightarrow(\exists y)(F(y)\wedge G(x, y)))$。

谓词逻辑公式只是一个符号串，并没有具体的意义，但是，如果给这些符号串一个解释，使它具有真值，这就变成了一个命题。

谓词逻辑公式的演算只涉及符号、符号序列、符号序列的变换等，没有涉及符号和公式的意义。这种不涉及符号、公式意义的研究称为语法研究。例如，定理、可证明性等都是语法概念。而对符号和公式的解释，以及公式和它的意义等，都属于语义研究的范围。例如，个体域、解释、赋值、真假、普遍有效性、可满足性等，都是语义概念。

【例 5-78】 用谓词逻辑公式表示：张飞是计算专业的学生，他不喜欢编程。

定义：COMPUTER(x)表示 x 是计算专业的学生。

定义：LIKE(x，y)表示 x 喜欢 y。

谓词逻辑公式为 COMPUTER(张飞)∧¬LIKE(张飞，编程)。

6. 谓词逻辑的特点

谓词逻辑的优点是可以简单地说明和构造复杂的事物，并且分离了知识和处理知识的过程；谓词逻辑与关系数据库有密切的关系；一阶谓词逻辑具有完备的逻辑推理算法；逻辑推理方法不依赖于任何具体领域，具有较大的通用性。

谓词逻辑的缺点是难以表示过程和启发式的知识，不能表示具有不确定性的知识；当事实的数量增大时，在证明过程中可能产生计算量的“组合爆炸”问题；内容与推理过程的分离，使内容包含的大量信息被抛弃，使得处理过程效率低。

5.4.5 逻辑运算应用

逻辑运算广泛用于计算领域，如集成电路芯片都采用逻辑电路设计；如 Prolog 编程语言建立在一阶谓词逻辑的基础上；如程序编译时，采用逻辑推理分析程序语法。

1. 逻辑运算在数据库查询中的应用

逻辑运算中的“与”“或”“非”可以用来表示日常生活中的“并且”“或者”“除非”等判断。例如，在数据库中查询信息时，就会用到逻辑运算语句。

【例 5-79】 查询某企业中基本工资高于 5000 元，并且奖金高于 3000 元，或者应发工资高于 8000 元的职工。

查询语句：基本工资＞5000.00 AND 奖金＞3000.00 OR 应发工资＞8000.00。

2. 逻辑运算在图形处理中的应用

逻辑运算可以用于对图形进行某些剪辑操作。如图 5-43 所示，将两个相交的图形进行与运算时，可以剪裁出其中相交部分的子图；将两个相交的图形进行或运算时，可以将两个图形合并成一个图形，将两个图形进行非运算时，就可以减去其中一个图形中相交的部分；将两个相交的图形进行异或运算时，可以剪裁两个图形中的相交部分。

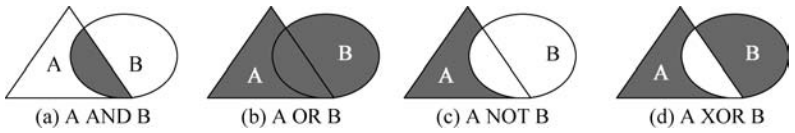


图 5-43 逻辑运算在图形处理中的应用

3. 逻辑运算在集合中的应用

【例 5-80】 如图 5-44 所示，设集合 A 包含“全集”中所有的偶数(2 的倍数)，集合 B 包含“全集”中所有 3 的倍数；集合 A 与集合 B 的交集 A∩B(在集合 A AND B 中所有的元素)将是“全集”中所有 6 的倍数。

4. 逻辑运算在加法器电路设计中的应用

计算机的基本运算包括：算术运算、移位运算、逻辑运算，它们由 CPU 内部的算术逻辑运算单元(ALU)进行处理。**加法运算是计算机最基本的运算**，如算术四则运算、程序中的条件判断语句(if)、循环语句(for)等，都可以转换为 CPU 内部的加法操作。利用前述的逻

图 5-44　集合 A 和集合 B 之间的逻辑关系

辑门电路，设计一个能实现两个 1 位二进制数做算术加法运算的电路，这个电路称为“半加器”，利用“半加器”就可以设计出“全加器”逻辑电路。

（1）半加器逻辑电路设计。在半加器中，暂时不考虑低位进位的情况。半加器有两个输入端：加数 A、被加数 B；两个输出端：和 S、进位 C。1 位半加器逻辑电路及真值表如图 5-46 所示，它由 1 个“异或门”和一个“与门”组成，考察真值表中 C 与 A、B 之间的关系是“与”的关系，即 $C=A\times B$；再看 S 与 A、B 之间的关系，也可看出这是“异或”关系，即 $S=A\oplus B$。因此，可以用“与门”和“异或门”来实现真值表的要求。

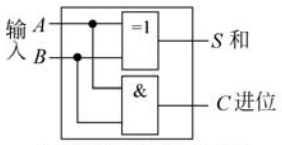
【例 5-81】 在图 5-45 所示半加器电路中，验证两个 1 位二进制数相加结果是否正确。

设 $A=0,B=0$；$S=A\oplus B=0\oplus 0=0$；$C=A\times B=0\times 0=0$；逻辑电路正确。

设 $A=1,B=0$；$S=A\oplus B=1\oplus 0=1$；$C=A\times B=1\times 0=0$；逻辑电路正确。

设 $A=0,B=1$；$S=A\oplus B=0\oplus 1=1$；$C=A\times B=0\times 1=0$；逻辑电路正确。

设 $A=1,B=1$；$S=A\oplus B=1\oplus 1=0$；$C=A\times B=1\times 1=1$；逻辑电路正确。



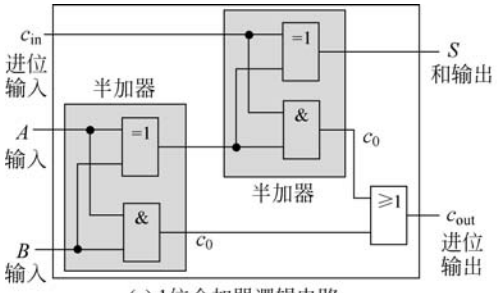
(a) 1位半加器逻辑电路

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

(b) 半加器真值表

图 5-45　1 位半加器逻辑电路和真值表

（2）全加器逻辑电路设计。全加器有多种逻辑电路设计形式。如图 5-46 所示，可用两个半加器和一个或门组成一个全加器，也可以利用其他逻辑电路组成全加器。逻辑电路可采用 VHDL 设计。



(a) 1位全加器逻辑电路

A	B	c_{in}	c_{out}	S
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

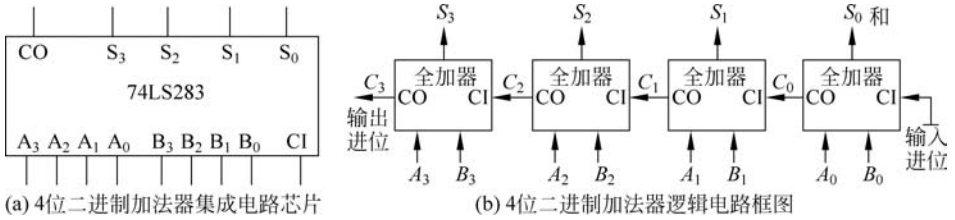
(b) 全加器真值表

图 5-46　1 位全加器逻辑电路和真值表

【例 5-82】 用 VHDL 设计一个一位二进制全加器逻辑电路，代码如下。

```
1  library IEEE;                                  -- 调用 IEEE 库函数(-- 为 VHDL 注释符)
2  use IEEE.STD_LOGIC_1164.all;                   -- 导入程序函数包
3  entity adder is port(a, b,ci : in bit;         -- 实体定义;输入:a、b、ci(输入进位)
4      sum, co : out bit);                        -- 输出:sum(和)、co(输出进位)
5  end adder;                                     -- 实体定义结束
6
7  architecture all_adder of adder is             -- 结构体定义
8  begin                                          -- 结构体开始
9      sum <= a xor b xor ci;                     -- 和的逻辑关系,<= 为赋值运算,xor 为异或逻辑
10     co <= ((a or b) and ci) or (a and b);      -- 进位的逻辑关系,or 为或逻辑,and 为与逻辑
11 end all_adder;                                 -- 结构体结束
```

(3) 4 位加法器电路设计。如图 5-47 所示，如果要搭建一个 4 位加法器，只需将 4 个 1 位全加法器连接即可。将 n 个 1 位加法器串行连接在一起，可搭建一个 n 位加法器，这种加法器称为行波进位加法器，加法器输出的稳定时间为$(2n+4)\Delta$。可以利用 4 位加法器逻辑电路，设计出 8 位、16 位、32 位、64 位等长度的二进制整数加法器。目前 CPU 中的整数加法器为 64 位，原理与以上相同，但结构要复杂得多。



(a) 4位二进制加法器集成电路芯片　　(b) 4位二进制加法器逻辑电路框图

图 5-47　由 4 个 1 位全加器组成的 4 位二进制加法器

习　题　5

5-1　计算机对同一类型的数据采用相同的数据长度进行存储，如 1 和 12345678 都采用 4 字节存储，为什么不对 1 采用 1 字节存储，12345678 采用 4 字节存储？

5-2　学生成绩评定等级有优秀、良好、中等、及格、不及格，需要几位二进制数表示？

5-3　为什么说整数四则运算理论上都可以转换为补码的加法运算？

5-4　为什么中文 Windows 系统内部采用 UTF-16(UCS-2)编码，而不使用 ASCII 码。

5-5　音频信号采样频率为 8kHz，样本用 256 级表示，采样位率多大时不会丢失数据？

5-6　简要说明有损压缩的算法思想。

5-7　视频图像帧之间的预测编码有哪些方法？

5-8　为什么 WinRAR 软件在压缩文本时压缩比很小，而在压缩图像文件时压缩比很高？

5-9　请对英语 So said，So done(说到做到)进行 LZW 算法编码。

5-10　一个英文字符需要 7 位 ASCII 码，LZW 压缩编码算法采用 12 位编码后会使压缩文件更大吗？如果一个文本中有 10 个 the 字符，采用 LZW 压缩编码算法时，the 字符的

压缩比是多少？

5-11　简要说明信息熵的特点。

5-12　实验：参考例 5-26 的程序案例，将十进制实数转换为 32 位规格化浮点数。

5-13　实验：参考例 5-43 的程序案例，打印字符串的 Unicode 编码。

5-14　实验：参考例 5-48 的程序案例，计算投掷硬币和投掷骰子的信息熵。

5-15　实验：参考例 5-53 的程序案例，对字符串进行 RLE 编码。

5-16　实验：参考例 5-56 的程序案例，将子目录下所有文件进行压缩。

5-17　实验：参考例 5-62 的程序案例，生成字符串的 CRC 检验编码。

5-18　实验：考试试卷中的单选题有 A、B、C、D 四个选项，答题必须从四个选项中选出一个正确选项，计算这个事件的信息熵。