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，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 二进制编码特征

1. 信息的二进制数表示

一切信息编码都包括**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，最终变换为一组0、1构成的二进制数据，并能无损地保留信息的含义。

信息可以用十进制或二进制的数字来表示。例如，当我们和别人谈话时，说的每个字都是字典中的一个。如果将字典中所有单词从1开始顺序编号，我们就可以精确地使用数字进行交谈，而不使用单词。当然，对话的两个人需要一本给每个单词编过号的字典以及足够的耐心，人与计算机之间的交谈也是这个道理。

2. 二进制编码的优点

如果计算机采用十进制数作信息编码，则加法运算需要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2个(0和1)，加法一共只有4个运算规则(0+0=0，0+1=1，1+0=1，1+1=10)。另外，用二进制做逻辑运算也非常方便，可以用1表示逻辑命题值“真”(True)，用0表示逻辑命题值“假”(False)。基于二进制的计算机并不意味着非黑即白，计算机采用二进制只是为了物理实现的简单化和逻辑推理的方便，降低计算机设计的复杂性。

也许我们可以指出，由于加法运算服从交换律，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。

也许我们还能指出，十进制1+2只需要做一位加法运算；而转换为8位二进制数后，

至少需要做 8 位加法运算(如 0000001+00000010),可见二进制数大大增加了计算工作量。但是目前普通的计算机(4 核 2.0GHz 的 CPU)每秒钟可以做 80 亿次以上的 64 位二进制加法运算,可见计算机最善于做大量的、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

当计算机接收到一系列二进制符号(0 和 1 字符串流)时,它并不能直接"理解"这些二进制符号的含义。**二进制数据的具体含义取决于程序对它的解释**。

【例 5-1】 简单地问二进制数 01000010 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 编码,则表示为十进制数 42;如果采用 ASCII 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有限几个基本数字符号表示所有数。进制称为**基数**,如十进制的基数为 10,二进制的基数为 2。任意 R 进制数,基本数字符号为 R 个,任意进制数可以用公式(5-1)表示:

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

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。

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

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,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)。

【例 5-3】 二进制数与十进制数四则运算的比较如图 5-1 所示。

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

6. 十六进制数编码

二进制表示一个大数时位数太多,计算机专业人员辨认困难。早期程序员采用八进制来简化二进制,以后又采用十六进制数来表示二进制数。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,引入十六进制数的原因是让计算机专业人员可以很方便地将十六

二进制计算	十进制验算	二进制计算	十进制验算
1001 + 10 = 1011	9 + 2 = 11	1110 − 1001 = 101	14 − 9 = 5
1001 + 10 1011	9 + 2 11	1110 − 1001 101	14 − 9 5
101×10 = 1010	5×2 = 10	1010 ÷ 10 = 101	10 ÷ 2 = 5
101 × 10 000 101 1010	5 × 2 10	101 10) 1010 10 10 10 0	5 2) 10 10 0

图 5-1　二进制数与十进制数四则运算比较

进制数转换为二进制数。

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

常用数制之间的基本特征和对应关系如表 5-1、表 5-2 所示。

表 5-1　计算机常用数制的基本特征

基本特征	十 进 制 数	二 进 制 数	十六进制数
运算规则	逢十进一，借一当十	逢二进一，借一当二	逢十六进一，借一当十六
基数	$R=10$	$R=2$	$R=16$
数符	0，1，2，…，9	0，1	0，1，2，…，9，A，B，C，D，E，F
权	10^n	2^n	16^n
数制标识符	D	B	H 或 0x

表 5-2　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
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之间的转换

在二进制数与十进制数的转换过程中,要频繁地计算 2 的整数次幂。表 5-3 和表 5-4 给出了 2 的整数次幂和十进制数值的对应关系。

表 5-3 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 5-4 二进制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	1/128	1/256
十进制小数	0.5	0.25	0.125	0.0625	0.031 25	0.015 625	0.007 812 5	0.003 906 25

二进制数转换成十进制数时,可以采用按权相加的方法,这种方法是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 5-4】 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}	
十进制数值	8	+	4	+	0	+	1	+	0.5	+	0	+	0.125	=13.625

图 5-2 二进制数按位权展开过程

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

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

2. 十进制数与二进制数转换

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。转换为整数时,第一次除法得到的余数为二进制数低位(第 K_0 位),最后一次余数为二进制数高位(第 K_n 位)。

小数部分采用乘 2 取整法。就是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 后小数部分等于 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 5-6】　将十进制 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。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。竖式运算过程如图 5-3 所示。

运算结果为$[18.8125]_{10}=[10010.1101]_2$

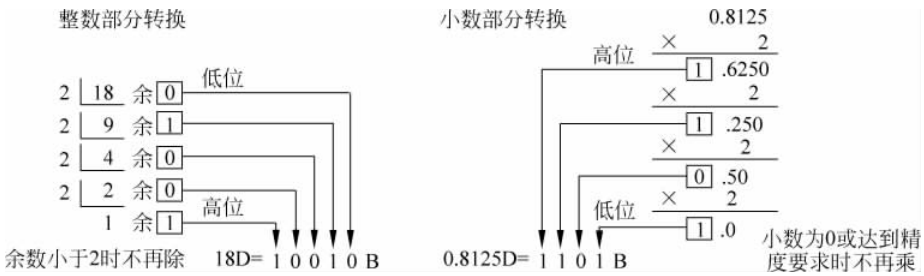


图 5-3　十进制数转换为二进制数的运算过程

【例 5-7】　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

3. 二进制数与十六进制数转换

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 5-8】　将二进制数 111101.010111 转换为十六进制数。

$[111101.010111]_2=[00111101.01011100]_2=[3D.5C]_{16}$，转换过程如图 5-4 所示。

0011	1101	0101	1100
3	D	5	C

图 5-4　例 5-8 题图

【例 5-9】　将二进制整数$[11010101]_2$转换为十六进制数整数，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 5-10】 将十六进制数 4B.61 转换为二进制数。

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

4	B	6	1
0100	1011	0110	0001

图 5-5 例 5-10 题图

【例 5-11】 将十六制整数 4B 转换为二进制数整数,Python 指令如下。

```
>>> bin(int('4b',16))                    # 先转换为十进制整数,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:一是数值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,而浮点数的存储和计算都较为复杂,运算效率低。

BCD 码是一种二-十进制编码,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,与 4 位二进制编码不同的是,它只选用了 4 位二进制编码中前 10 组代码。BCD 编码与十进制数的对应关系如表 5-2 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 编码,因为 BCD 编码更容易将二进制数转换为十进制数。

二进制数使用 0000～1111 全部编码,而 BCD 数仅仅使用 0000～1001 十组编码,编码到 1001 后就产生进位,而二进制编码到 1111 才产生进位。

【例 5-12】 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 5-6 例 5-12 题图

【例 5-13】 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 5-7 所示。

BCD码	0111	0110	1000	0001
十进制数	7	6	8	1

图 5-7 例 5-13 题图

【例 5-14】 将二进制数 111101.101 转换为 BCD 编码。

如图 5-8 所示,**二进制数不能直接转换为 BCD 码**,因为编码方法不同,可能会出现非法编码。可以将二进制数$[111101.101]_2$ 转换为十进制数$[61.625]_{10}$后,再转换为 BCD 码。

常用数制之间的转换方法如图 5-9 所示。

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

图 5-8　例 5-14 题图

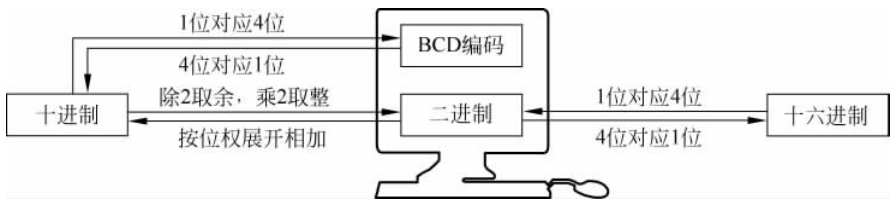


图 5-9　常用数制之间的转换方法

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(bit)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 5-15 所示，8 位无符号整数运算结果大于 255 时，就会产生“溢出”问题。

【例 5-15】　$[11001000]_2+[01000001]_2=$ $\boxed{1}$ 00001001(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，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一个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。解决数据不同存储长度的方法是建立不同的数据类型，程序设计时首先声明数据类型，计算机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 个字节，长整型数据的存储长度为 8 个字节。这样，小数字的数据虽然会浪费一些存储空间，但是等长存储提高了整体运算速度，这是一种“以空间换时间”的计算思维方式。

【例 5-16】　如图 5-10 所示，无符号数$[22]_{10}=[10110]_2$ 在计算机中的存储形式如下。

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 5-10　数据的不同存储长度

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 5-17】 $[+23]_{10}=[+10111]_2=[\mathbf{0}0010111]_2$,如图 5-11 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 5-11 例 5-17 题图

【例 5-18】 $[-23]_{10}=[-10111]_2=[10010111]_2$,如图 5-12 所示,最高位 1 表示负数。二进制数-10111 真值与机器数的区别如图 5-12 所示。

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

图 5-12 例 5-18 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。如图 5-13 所示,定点数假设小数点固定在最低有效位后面(隐含)。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(有效数大于 10 位),一般也用浮点数表示。

【例 5-19】 十进制数-73 的二进制数真值为-1001001,如果用 2 个字节存储,最高位(符号位)用 0 表示“+”,1 表示“-”,则二进制数原码的存储格式如图 5-13 所示。

图 5-13 16 位定点整数的存储格式

在 32 位计算机系统中,整型数(int)用 4 个字节表示,最高位用于表示数值的符号,其余 31 位表示数据。如果数据运算结果超出了 31 位,就会产生**溢出**问题。

2. 浮点数的表示

实数是最常见的自然数,实数中的小数在计算机中的存储和运算是一个非常复杂的事情。目前已有两位计算机科学家因为研究浮点数(小数)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为

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

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 5-20】 $[1001.011]_2=[0.1001011]_2\times 2^4$。

【例 5-21】 $[-0.0010101]_2=[-0.10101]_2\times 2^{-2}$。

3. 二进制小数的截断误差

1）浮点数存储空间不够引起的截断误差

如式(5-2)所示，假设用 1 个字节表示和存储浮点数 N，原始指数 E 的符号和数字本身需要 2 位，尾数 M 符号为 1 位，尾数 M 本身为 3 位。将二进制数 10.101 存储为浮点数时，尾数由于存储空间不够，最右边的 1 位数据(1)就会丢失(如例 5-22 所示)。这个现象称为**截断误差**(舍入误差)。由于尾数空间不够，导致部分数值丢失时，可以通过使用较长的尾数域来减少截断误差的发生。

【例 5-22】

0	10	0	1010	1

(存储长度为 8b 时，最后一位产生截断误差)。

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，无论用多少位数字，总有一些数值不能精确地表示出来。二进制记数法与十进制记数法的区别在于，**二进制记数法中有无穷展开式的数值多于十进制**。

【例 5-23】 十进制小数 0.8，转换为二进制时为 0.11001100…，后面还有无数个 1100，这说明十进制的有限小数转换成二进制时，不能保证精确转换；二进制小数转换成十进制也遇到同样的问题。

【例 5-24】 将十进制数值 1/10 转换为二进制数时，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。例如 Python 运算的截断误差如下。

```
>>> 0.1 + 0.1 + 0.1                          #3 个 0.1 相加
0.30000000000000004                          #输出(出现截断误差)
>>> .1 + .1 + .1 + .1 + .1 + .1 + .1 + .1    #8 个 0.1 相加
0.7999999999999999                           #输出(出现截断误差)
```

在十进制小数转换成二进制小数时，整个计算过程可能会无限制地进行下去，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，大多数商用软件提供的计算精度已经足够了。

一些特殊应用领域(如导航系统等)，很小的误差可能在运算中不断累加，最终产生严重的后果。如浮点数乘方运算中，当指数很大时，很小的误差将会呈指数级放大。

【例 5-25】 $1.01^{365}=37.8$；$1.02^{365}=1377.4$；$0.99^{365}=0.026$；$0.98^{365}=0.0006$。

4. 规格化浮点数的表示与存储

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 5-5 所示,计算机中的浮点数严格遵循 IEEE 754 标准。

表 5-5 IEEE 754 标准规定的浮点数规格

浮点数规格	码长/b	S 符号/b	e 阶码/b	M 尾数/b	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 与公式(5-2)中的原始指数 E 并不相同;尾数 M 与公式(5-2)中的 M 也有所区别。

1) IEEE 规格化浮点数

浮点数的表示方法多种多样,因此 IEEE 对浮点数的表示进行了严格规定。IEEE 规格化浮点数规定:小数点左侧整数必须为 1(如 1.xxxxxxxx),指数采用阶码表示。

【例 5-26】 1.75D=1.11B,用科学计数法表示时,小数点前一位为 0 还是 1 并不确定,IEEE 规格化浮点数规定小数点前一位为 1,即规格化浮点数为 1.75D=1.11B $=1.11\times2^{0}$。

浮点数规格化的目的有两个:一是整数部分恒为 1,这样在存储尾数 M 时,就可以省略小数点和整数 1(与公式(5-2)的区别),从而用 23 位尾数域表达了 24 位尾数;二是尾数域最高有效位固定为 1 后,尾数能以最大数的形式出现,即使遭遇类似截断的操作,仍然可以保持尽可能高的精度。

2) 数据混淆问题

整数部分的 1 舍去后,会不会造成两个不同数据的混淆呢?例如,A=1.010 011 中的整数部分 1 在存储时被舍去了,那么会不会造成 A=0.010 011(整数 1 已舍去)与 B=0.010 011 两个数据的混淆呢?其实不会,仔细观察就会发现,数据 B 不是一个规格化浮点数,数据 B 可以改写成 1.0011×2^{-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。

3) 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(如表 5-5 所示)。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 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(float)的偏移值为 127;64 位浮点数(double)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 表示无穷大,否则表示无效数字。

4) IEEE 浮点数的存储形式

IEEE 标准浮点数的存储格式如图 5-14 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)。

实数转换为 IEEE 标准浮点数的步骤是:将数字转换为二进制数→在 S 中存储符号值→将二进制数规格化→计算出移码 e 和尾数 M→最后连接[SeM]即可。

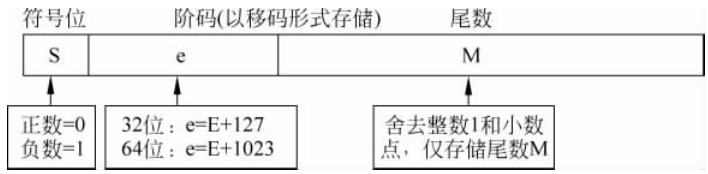


图 5-14　IEEE 754 规格化浮点数存储格式

【例 5-27】 将十进制实数 26.0 转换为 32 位 IEEE 规格化二进制浮点数。

实数 26.0＝11010B＝1.1010×2^4，规格化浮点数的转换方法如图 5-15 所示。

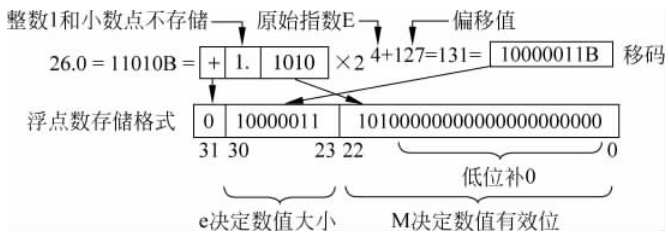


图 5-15　32 位规格化浮点数的转换方法和存储格式

【例 5-28】 将浮点数 11000001　11001001　00000000　00000000 B 转换成十进制数。

步骤 1：把 32 位浮点数分割成三部分：1　10000011　1001001 00000000 00000000 B，可得符号位 S＝1B；阶码 e＝10000011B；尾数 M＝1001001 00000000 00000000B。

步骤 2：还原原始指数 E：E＝e－127＝10000011B－01111111B＝100B＝4。

步骤 3：还原尾数 M 为规格化形式：M＝1.1001001B×2^4(1.从隐含位而来)。

步骤 4：还原为非规格化形式为 N＝S1.1001001B×2^4＝S11001.001B(S＝符号位)。

步骤 5：还原为十进制数形式为 N＝S11001.001B ＝－25.125(S＝1，说明是负数)。

5) 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(如表 5-5 所示)。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})×2^{127}＝3.4×10^{38}(有效数 6～7 位)，可表示的最小正数为 2^{-126}＝1.17×10^{-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 128 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？自然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？这些问题将在更深入的课程中讨论。总之，小数的处理过程非常复杂。浮点运算通常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序。

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，易于与真值的转换。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，然后将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了。二是在原码中，由于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 5-29】 $[01000010]_2+[01000001]_2=[10000011]_2$(进位导致的符号位错误)。

【例 5-30】 $[00000010]_2+[10000001]_2=[10000011]_2$(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，负数的反码是对该数的**原码除符号位外各位取反**。

【例 5-31】 二进制数字长为 8 位时，$[+5]_{10}=[00000101]_{原}=[00000101]_{反}$。

【例 5-32】 二进制数字长为 8 位时，$[-5]_{10}=[10000101]_{原}=[11111010]_{反}$。

3. 补码运算的概念

两个数相加时，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；100 以内的补码对有 1-99，2-98，…，50-50 等。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 5-16 所示，+4 的补码为+4，-1 的补码为+9(10-|1|=9)。

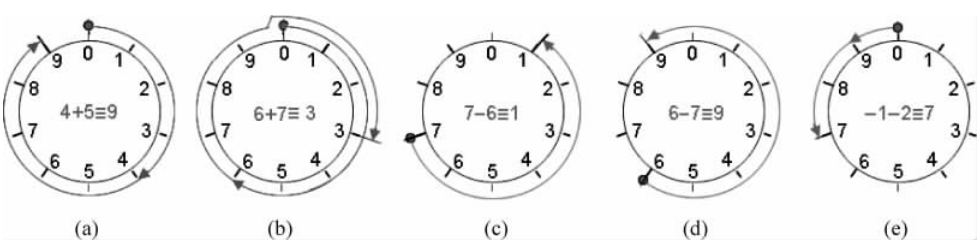


图 5-16 模=10 的十进制数模运算示意图(顺时针方向为+，逆时针方向为-)

“模”是指计量系统的计数范围，例如时钟的计量范围是 0～12，模=12。十进制数中，1 位数的模为 10，2 位数的模为 100，其余以此类推。模运算具有以下特征：任何有关模的计算，均可将减法转换为加法运算。下面利用模运算将减法简化为加法运算。

【例 5-33】 $4+5\equiv[4]_{补}+[5]_{补}=[9]_{补}\ \mathrm{mod}\ 10=9$(如图 5-16(a)所示)。

【例 5-34】 $6+7\equiv[6]_{补}+[7]_{补}=[13]_{补}\ \mathrm{mod}\ 10=3$(如图 5-16(b)所示)。

【例 5-35】 $7-6\equiv[7]_{补}+[10-6]_{补}=[11]_{补}\ \mathrm{mod}\ 10=1$(如图 5-16(c)所示)。

【例 5-36】 $6-7\equiv[6]_{补}+[10-7]_{补}=[9]_{补}\ \mathrm{mod}\ 10=9$(如图 5-16(d)所示)。

【例 5-37】 $-1-2\equiv[10-1]_{补}+[10-2]_{补}=[17]_{补}\ \mathrm{mod}\ 10=7$(如图 5-16(e)所示)。

【例 5-38】 $12-12\equiv[12]_{补}+[100-12]_{补}=[100]_{补}\ \mathrm{mod}\ 100=0$。

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 5-39】 $[10]_{10}$的二进制原码为$[10]_{10}=[00001010]_{原}$（最高位 0 表示正数）。

$[-10]_{10}$的二进制原码为$[-10]_{10}=[\mathbf{1}0001010]_{原}$（最高位 1 表示负数）。

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

$[-10]_{10}$的二进制反码为$=[\mathbf{1}1110101]_{反}$（最高位 1 表示负数）。

【例 5-41】 $[10]_{10}$的二进制补码为$[10]_{10}=[00001010]_{补}$（最高位 0 表示正数）。

$[-10]_{10}$的二进制补码为$[-10]_{10}=[\mathbf{1}1110110]_{补}$（最高位 1 表示负数）。

计算机中，二进制数各种编码的表示方法如表 5-6 所示。

表 5-6　8 位二进制数特殊值的编码方法

十进制数	二进制数真值	二进制数原码	二进制数反码	二进制数补码
0	0	00000000	00000000	00000000
0	0	10000000	11111111	00000000
+1	+1	00000001	00000001	00000001
−1	−1	10000001	11111110	11111111
−127	−1111111	11111111	10000000	10000001
−128	−10000000	—	—	10000000

5. 补码运算规则

补码运算的算法思想是：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形式，从而使加减法运算转换为单纯的加法运算。补码运算在逻辑电路设计中实现容易。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。不论对正数和负数，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、B 为正整数、负整数、0 均可。

【例 5-42】 $A=[-70]_{10}$，$B=[-55]_{10}$，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。

$[-70]_{10}=[-(64+4+2)]_{10}=[11000110]_{原}=[10111001]_{反}+[00000001]=[10111010]_{补}$

$[-55]_{10}=[-(32+16+4+2+1)]_{10}=[10110111]_{原}=[11001000]_{反}+[00000001]=[11001001]_{补}$

$$\begin{array}{rr} & 10111010 \\ + & 11001001 \\ \hline \boxed{1} & 10000011 \end{array}$$

没有溢出时,进位 1 自然丢失→

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?因为加法器设计中,本位值与进位由不同逻辑电路实现(参见图 5-48)。

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:

$$[10000011]_{补}=[11111100]_{反}+[00000001]_{2}=[11111101]_{原}=[-125]_{10}$$

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 为了提高计算效率,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 英文字符编码

计算机除了用于数值计算外,还要处理大量非数值信息,其中字符信息占有很大比重。字符信息包括西文字符(字母、数字、符号)和汉字字符等。它们需要进行二进制数编码后,才能存储在计算机中并进行处理,如果每个字符对应一个唯一的二进制数,这个二进制数就称为字符编码。西文字符与汉字字符由于形式不同,编码方式也不同。

1. BCDIC 编码

早期计算机的 6 位字符编码系统 BCDIC(二进制数与十进制数交换编码)从霍尔瑞斯(Herman Hollerith)卡片发展而来,后来逐步扩展为 8 位 EBCDIC 码,并一直是 IBM 大型计算机的编码标准,但没有在其他计算机中使用。

2. ASCII 编码

ASCII(读[阿斯克],美国信息交换标准码)制定于 1967 年。由于当时数据存储成本很高,专家们最终决定采用 7 位字符编码。ASCII 编码如表 5-7 所示。

表 5-7　ASCII 码表(部分字符)

字符	ASCII 码			字符	ASCII 码		
	二进制	十进制	十六进制		二进制	十进制	十六进制
0	0110000	48	30	A	1000001	65	41
1	0110001	49	31	B	1000010	66	42
2	0110010	50	32	C	1000011	67	43
3	0110011	51	33	⋮	⋮	⋮	⋮
4	0110100	52	34	Z	1011010	90	5A
5	0110101	53	35	⋮	⋮	⋮	⋮
6	0110110	54	36	a	1100001	97	61
7	0110111	55	37	b	1100010	98	62
8	0111000	56	38	⋮	⋮	⋮	⋮
9	0111001	57	39	z	1111010	122	7A

ASCII 编码用 7 位二进制数对 1 个字符进行编码。由于基本存储单位是字节(8b),计算机用 1 个字节存放 1 个 ASCII 字符编码。

【例 5-43】 Hello 的 ASCII 编码。

查 ASCII 表可知,“Hello”的 ASCII 码如图 5-17 所示。

H	e	l	l	o
1001000	1100101	1101100	1101100	1101111

图 5-17　例 5-43 题图

【例 5-44】 求字符 A 和 a 的 ASCII 编码,Python 指令如下。

```
>>> ord('A')                    # 计算字符 A 的 ASCII 编码
65                              # 输出字符 A 的十进制编码
>>> chr(97)                     # 计算 ASCII 编码 = 97 的字符
'a'                             # 输出字符
```

3. 扩展 ASCII 码

ASCII 码(ANSI 码)最大的问题在于它是一个典型的美国标准,它不能很好地满足其他非英语国家的需要。例如,它无法表示英镑符号(£);英语中的单词很少需要重音符号(读音符号),但是在使用拉丁字母语言的许多欧洲国家中,在语言中使用重音符号很普遍(如 é);还有一些国家不使用拉丁字母语言,如希伯来语、阿拉伯语、俄语、汉语等。

由于 1 个字节有 8 位,而 ASCII 码只用了 7 位,还有 1 位多出来。于是很多人就想到“我们可以使用 128～255 的码字来表示其他东西”。这样麻烦来了,这么多人同时出现了这样的想法,而且将它付诸实践。1981 年 PC 推出时,显卡的 ROM 芯片中就固化了一个 256 字符的字符集,它包括一些欧洲语言中用到的重音字符,还有一些画图的符号等。所有计算机厂商都开始按照自己的方式使用高位的 128 个码字。例如,有些 PC 上编码 130 表示 é,

而在以色列的计算机中,它可能表示希伯来字母 λ。当 PC 在美国之外销售时,这些扩展的 ASCII 码字符集就完全乱套了。

最终 ANSI(美国国家标准学会)结束了这种混乱。ANSI 标准支持 1～4 个字节的编码,并规定每个字节低位的 128 个码字采用标准 ASCII 编码;高位的 128 个码字,根据用户所在地语言的不同采用"码页"处理方式。如最初的 IBM 字符集码页为 CP437,以色列使用的码页是 CP862,中国使用的码页是 CP936 等。

5.2.2 中文字符编码

1. 双字节字符集

亚洲国家常用文字符号有大约 2 万多个,如何容纳这些语言的文字而保持和 ASCII 码的兼容性呢? 8 位编码无论如何也满足不了需要,解决方案是采用双字节字符集(DBCS)编码,即用 2 个字节定义 1 个字符,理论上可以表示 $2^{16}=65\ 535$ 个字符。当编码值低于 128 时为 ASCII 码,编码值高于 128 时,为所在国家语言符号的编码。

【例 5-45】 早期双字节汉字编码中,1 个字节最高位为 0 时,表示一个标准的 ASCII 码;字节最高位为 1 时,用 2 个字节表示一个汉字,即有的字符用 1 个字节表示(如英文字母),有的字符用 2 个字节表示(如汉字),这样可以表示 $2^{16-2}=16\ 384$ 个汉字。

双字节字符集虽然缓解了亚洲语言码字不足的问题,但是也带来了新的问题。

(1) 在程序设计中处理字符串时,指针移动到下一个字符比较容易,但移动到上一个字符就非常危险了,于是程序设计中 s++或 s--之类的表达式不能使用了。

(2) 一个字符串的存储长度不能由它的字符数来决定,必须检查每个字符,确定它是双字节字符,还是单字节字符。

(3) 丢失 1 个双字节字符中的高位字节时,后续字符会产生"乱码"现象。

(4) 双字节字符在存储和传输中,高字节还是低字节在前面? 没有统一标准。

互联网的出现让字符串在计算机之间的传输变得非常普遍,于是所有的混乱都集中爆发了。非常幸运的是 Unicode(国际统一码)字符集适时而生。

2. 汉字编码

英文为拼音文字,所有英文单词均由 52 个英文大小写字母组合而成,加上数字及其他标点符号,常用字符仅 95 个,因此 7 位二进制数编码就够用了。汉字由于数量庞大,构造复杂,这给计算机处理带来了困难。汉字是象形文字,每个汉字都有自己的形状。所以,每个汉字在计算机中需要一个唯一的二进制编码。

1) GB 2312—80 字符集的汉字编码

1981 年,我国颁布了《GB 2312—80 信息交换用汉字编码字符集·基本集》(简称国标码)。GB 2312—80 标准规定:一个汉字用两个字节表示,每个字节只使用低 7 位,字节最高位为 0。GB 2312—80 标准共收录 6763 个简体汉字、682 个符号,其中一级汉字 3755 个,以拼音排序,二级汉字 3008 个,以偏旁排序。GB 2312—80 标准的编码方法如表 5-8 所示。

表 5-8　GB 2312—80 中国汉字编码标准表(部分编码)

位码 / 区码		第 2 字节编码							
		00100001	00100010	00100011	00100100	00100101	00100110	00100111	00101000
第 1 字节	区/位	位 01	位 02	位 03	位 04	位 05	位 06	位 07	位 08
00110000	16 区	啊	阿	埃	挨	哎	唉	哀	皑
00110001	17 区	薄	雹	保	堡	饱	宝	抱	报
00110010	18 区	病	并	玻	菠	播	拨	钵	波
00110011	19 区	场	尝	常	长	偿	肠	厂	敞
00110100	20 区	础	储	矗	搐	触	处	揣	川

【例 5-46】 “啊”字的国标码如图 5-18 所示。

2) 内码

国标码每个字节的最高位为“0”,这与国际通用的 ASCII 码无法区分。因此,在早期计算机内部,汉字编码全部采用内码(也称机内码)表示,早期内码是将国标码两个字节的最高位设定为“1”,这样解决了国标码与 ASCII 码的冲突,保持了中英文的良好兼容性。目前 Windows 系统内码为 Unicode 编码,字节高位“0”“1”兼有。

【例 5-47】 “啊”字的内码如图 5-19 所示。

00110000	00100001
30H	21H

图 5-18　“啊”字的国标码

10110000	10100001
B0H	A1H

图 5-19　“啊”字的内码

早期在 DOS 操作系统内部,字符采用 ASCII 码;目前操作系统内部基本采用 Unicode 字符集的 UTF 编码。为了利用英文键盘输入汉字,还需要对汉字编制一个键盘输入码,主要输入码有拼音码(如微软拼音)、字形码(如五笔字型)等。

3) 互联网汉字编码体系

目前互联网上使用的汉字编码体系主要有四种,一是中国大陆使用的 GBK 码;二是中国港台地区使用的 BIG5 码;三是新加坡、美国等海外华语地区使用的 HZ 码;四是国际统一码 Unicode。同一语言文字在信息交流中存在如此大的差异,这给信息处理带来了复杂性。

3. 点阵字体编码

ASCII 码和 GB 2312 汉字编码主要解决了字符信息的存储、传输、计算、处理(录入、检索、排序等)等问题,而字符信息在显示和打印输出时,需要另外对“字形”进行编码。通常将字体(字形)编码的集合称为**字库**,将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,再输出到外设(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,因此计算机中有几十种中、英文字库。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不能放大,一旦放大后字符边缘就

会出现锯齿现象。

【例 5-48】 图 5-20 是字符“啊”的点阵图,每行用 2 个字节表示,共用 16 行、32 个字节来表达一个 16×16 点阵的汉字字体信息。

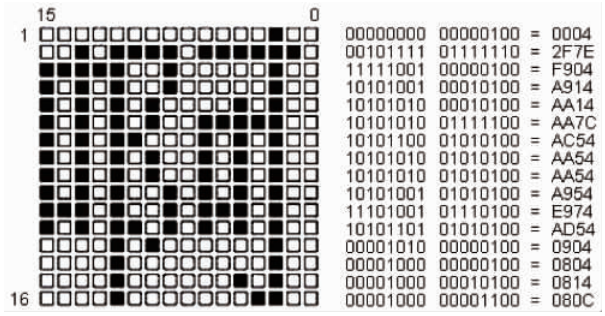


图 5-20 左:字符“啊”的点阵字体 左:字符“啊”的编码

4. 矢量字体编码

矢量字体保存的是每个字符的数学描述信息,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 5-21 所示,矢量字体重要的特征是轮廓(outline)和字体精调(hint)控制点。

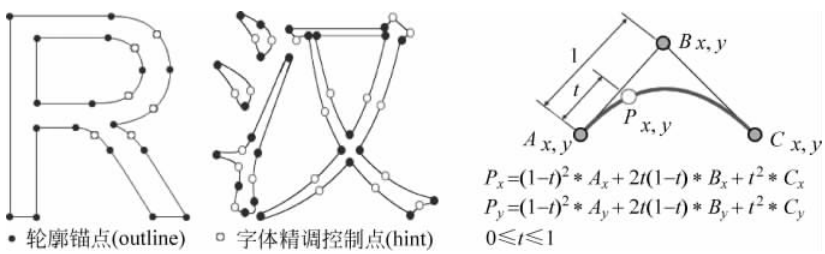


图 5-21 左-中:矢量字体轮廓和控制点 右:二次贝塞尔曲线计算公式示意图

轮廓是一组封闭的路径,它由线段或贝塞尔(Bézier)曲线(二次或三次贝塞尔曲线)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。

轮廓虽然精确描述了字体的外观形式,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,用来精密调整字体,让字体变得更美观,更清晰。

计算机大部分时候采用矢量字体显示。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,或者这样处理太麻烦了。因此,小字体一般采用点阵字体来弥补矢量字体的不足。

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。

5.2.3　国际字符编码

1. 国际通用字符集

国际统一码主要有 Unicode 联盟编制的 Unicode 字符集和 ISO(国际标准化组组织)编制的 UCS(通用字符集)字符集。Unicode 与 UCS 在 1991 年合并，目前两个组织独立公布各自的标准，但是都同意保持两者标准码表的兼容(编码相同)。

早期 Unicode 字符集采用 16 位编码，共有 65 536 个码字；UCS 字符集采用 32 位编码，共有 42 亿个码字；合并后的 Unicode 字符集共 1 112 064 码字。截止 Unicode 5.0 标准为止，大概使用了 25 万个码字，其中汉字为 7 万多个码字(汉字单字大约有 9 万多个)。Unicode 字符集的编码空间如图 5-22 所示。

图 5-22　Unicode 字符集的编码空间

UTF-16 是对 Unicode 字符集的编码，它用 2 个或 4 个字节表示一个字符，并且支持增补字符编码。UTF-32 是对 Unicode 字符集的编码，统一用 4 个字节表示一个字符。UTF-8 是对 Unicode 字符集的编码，用 1～6 个字节表示一个字符(变长编码)。

UCS-2 是对 UCS 字符集的编码，用 2 个字节表示 1 个字符，编码与 UTF-16 相同，不同之处是 UCS-2 不能表示增补字符。UCS-4 对所有字符编码都采用 4 个字节。

Unicode 对每个字符赋予了一个正式的名称，方法是在一个代码点值(十六进制数)前面加上 U+，如字符 A 的名称是 U+0041；字符“中”名称是 U+4E2D。

目前 Unicode 字符集获得了网络、操作系统、编程语言等领域的广泛的支持。当前所有主流操作系统都支持 Unicode 字符集，如 Windows 和 Linux 等。

2. Unicode 字符的存储和传输问题

1) 大端与小端字节序

英国讽刺小说家斯威夫特(Jonathan Swift)在《格列佛游记》中写道：小人国的内战源于吃鸡蛋时，究竟从大端(Big Endian)敲开还是从小端(Little Endian)敲开，由此曾发生过六次叛乱，其中一个皇帝送了命，另一个丢了王位。

斯威夫特虚构的故事却在计算机中真实地发生了。计算机字符编码的存储和传输过程中，同样遇到了大端与小端的问题。例如，“汉”字 Unicode 编码是 U+6C49，那么写入文件时，究竟是将 6C(高端字节)写在前面，还是将 49(低端字节)写在前面？将 6C 写在前面是大端字节序(BE，高位在前)；而将 49 写在前面是小端字节序(LE，低位在前)。x86 计算机的数据存储和传输采用小端字节序；Linux 和 TCP/IP 协议采用大端字节序。

【例 5-49】 字符串 Hello 在 Unicode 编码中,不同字节序的编码形式如下。

大端字节序编码:U+0048 U+0065 U+006C U+ 006C U+006F(用于 Linux);

小端字节序编码:U+4800 U+6500 U+6C00 U+6C00 U+6F00(用于 Windows)。

2) UBOM 字节序标识符

计算机在不清楚数据的字节序是大端还是小端的情况下,程序如何进行数据识别呢?解决方法是在每一个 Unicode 字符文本的最前面增加 2 个字节,用"FE FF"表示大端字节序(BE),用"FF FE"表示小端(LE)字节序,程序在读取到这些标识符后就知道字节序了。由于"FE FF"和"FF FE"不是 Unicode 规定的字符编码,正常情况下不会用到,所以不用担心出现与字符编码冲突的问题,这就是 UBOM(Unicode Byte Order Mark)字节序标识符。

3. UTF-16 编码

1) UTF-16 编码方法

早期 Unicode 字符集的 UTF-16 编码长度固定为 2 个字节,一共可以表示 $2^{16}=65\ 535$ 个字符。显然,它无法覆盖世界上的全部文字。Unicode 4.0 标准考虑到这种情况,定义了 45 960 个增补字符,增补字符用 4 个字节表示。目前的 UTF-16 编码就是 Unicode 字符集加上增补字符集。UTF-16 编码主要用于 Windows 操作系统。

UTF-16 编码有 2 个和 4 个字节的不同编码长度,这给字符的存储和计算带来了麻烦。因此 UTF-16 编码规定,要么用 2 个字节表示,要么是 4 个字节表示。

2) UTF-16 编码字节序

UTF 系列编码的字节序有 LE、BE、BOM、无 BOM 几种编码方法。

【例 5-50】 字符串"中国"的各个版本 UTF-16 字节序编码如表 5-9 所示。

表 5-9 UTF-16 编码的各种字节序案例

不同字节序的编码标准	"中国"字符的编码序列	说 明
UTF-16BE	4E 2D 56 FD	大端字节序编码
UTF-16LE	2D 4E FD 56	小端字节序编码
UTF-16(BOM,BE)	FE FF 4E 2D 56 FD	大端字节序标识符+大端编码
UTF-16(BOM,LE)	FF FE 2D 4E FD 56	小端字节序标识符+小端编码

3) UTF-16 编码在类 UNIX 操作系统下的问题

在类 UNIX 系统下使用 UTF-16 编码会导致非常严重的问题,因为 UTF-16 编码的头 256 个字符的第 1 个字节都是 00H,而在类 UNIX 系统中(如 C 语言),00H 有特殊意义,如 \0 和/在文件名和 C 语言库函数里有特别的含义;其次,大多数使用 ASCII 码文件的类 UNIX 下的软件,如果不进行重大修改,会无法读取 16 位字符。基于这些原因,UTF-16 不适合作为类 UNIX 的内码,而采用 UTF-8 编码就可以避免这些问题。

4. UTF-8 编码

1) UTF-16 编码对存储空间的浪费

在 UTF-16 编码中,英文符号是在 ACSII 码的前面加上一个编码为 0 的字节。如 A 的 ASCII 码为 41,而它的 UTF-16 编码是 U+0041。这样,英文系统就会出现大量为 0 的字

节。而美国程序员无法忍受这种字符串所占空间的翻倍，而且以前大堆的文档使用的是ASCII字符集，谁去转换它们？于是程序员的选择是忽略Unicode字符集，继续走自己的老路，这显然会让事情变得更加糟糕。解决这个问题的方法是采用UTF-8编码。

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**。在UTF-8编码中，0～127之间的码字用1个字节存储，超过128的码字用2～4个字节存储。也就是说，UTF-8编码的长度是可变的。ASCII码中每个字符的编码在UTF-8编码中保持完全一致，都是1个字节长，这就解决了美国程序员的烦恼。一般来说，欧洲字符长度为1～2个字节，亚洲大部分字符则是3个字节，附加字符为4个字节。

【例5-51】 如图5-23所示，字符“中”在UTF-8编码中占3个字节。

字符	GB 2312—80	GBK	BIG5	UCS-2	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5-23　字符“中国”的各种编码形式（十六进制表示）

【例5-52】 字符串Hello的UTF-8编码为48 65 6C 6C 6F。它与ASCII编码标准完全相同。这有非常好的效果，因为英文文本使用UTF-8编码时，完全与ASCII码一致。而UTF-16编码对字符串Hello的编码为0048 0065 006C 006C 006F（大端字节序），可见存储空间会增加大量冗余的00编码。

【例5-53】 以I am Chinese为例，用ANSI存储12字节；用UTF-8存储12字节；用UFT-16存储＝24＋2字节（字节序）；用UTF-32存储48＋4字节（字节序）。

【例5-54】 以“我是中国人”为例，用ANSI存储10字节；用UTF-8存储15字节；用UTF-16存储10＋2字节（字节序）；用UTF-32存储20＋4字节（字节序）。

3）UTF-8编码的应用

类UNIX系统普遍采用UTF-8编码，如Linux系统的内码是UTF-8。TCP/IP网络协议、HTML网页，大多数浏览器软件都采用UTF-8编码。而Windows操作系统和Java语言的内码是UTF-16编码，但是应用软件采用其他编码系统（如GBK、UTF-8等）时，操作系统会自动识别编码系统。例如，Python语言源代码一般采用UTF-8编码，Windows系统会识别并显示UTF-8编码的文档。但是，应用软件一般不具有这种编码识别功能。因此编写Python3程序时，程序源代码必须保存为UTF-8格式（可在IDE中设置），并且在源代码首行声明编码格式（＃ -＊- coding：utf-8 -＊-），否则容易产生中文乱码问题。

注意：在Windows下，文本文件（如.txt、.log、.py、.java、.cpp等）有4种编码方式：ANSI（简体中文Windows系统中，ANSI代表GBK编码），Unicode（小端UTF-16编码），Unicode big endian（大端UTF-16编码），UTF-8编码。由于这4种编码都与ASCII编码兼容，所以有些人错误地认为文本文件就是采用ASCII编码的文件。

5.2.4　声音的数字化

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。例如对着话筒讲话时(如图 5-24(a)所示),话筒根据它周围空气压力的不同变化,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 5-24(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。

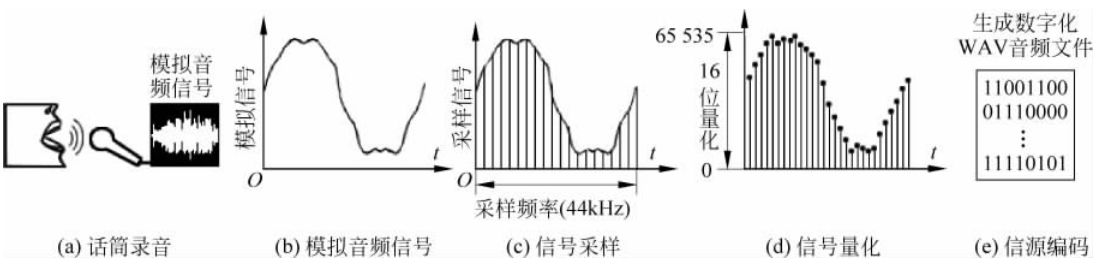


图 5-24 音频信号的数字化过程

1) 采样

任何连续信号都可以表示成离散值的符号序列,存储在数字系统中。因此,模拟信号转换成数字信号必须经过采样过程。采样过程是在固定的时间间隔内,对模拟音频信号截取一个振幅值(如图 5-24(c)所示),并用定长的二进制数表示(如 16b),将连续的模拟音频信号转换成离散的数字音频信号。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 之间。声音采样频率达到 40kHz(每秒钟采集 4 万个数据)就可以满足要求,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 5-24(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 5-24(b)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,采样速率为 40kHz,量化精度为 16 位时,位率=40 000×16=640kb/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 5-24(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、光盘、数字话筒、电子琴 MIDI 接口等设备输入计算机。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/数转换（A/D）。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，将离散的数字量再转换成为连续的模拟量信号（如电压），这一过程称为数/模转换（D/A）。

3. 编解码器

编解码器是对信号或者数据流进行变换的设备或者程序。这里的变换既包括对信号进行模/数或数/模转换，也包括对数据流进行压缩或解压缩操作。编解码器经常用在音频或视频等应用中，大多数编解码器对数据流进行了有损压缩，目的是为了得到更小的文件。

多媒体数据流往往同时包含了音频数据、视频数据，以及用于音频和视频数据同步的数据。这三种数据流可能会被不同的程序或者硬件处理，但是它们在传输或者存储时，这三种数据通常被封装在一起，这种封装通过文件格式来实现。例如，常见的音频格式有 wav、mp3、ac3 等；视频格式有 avi、mov、mp4、rmvb、3gp 等。这些格式中，有些只能使用特定的编解码器，而更多的格式能够以容器的方式使用各种编解码器。要播放某种格式的音频或视频文件，就需要支持该格式的解码器。台式计算机一般采用软件解码器解出音频或视频数据；而智能手机、数字电视、视频录像机等设备往往采用硬件编解码器。

5.2.5 图像的数字化

1. 图像的数字化

数字图像（Image）可以由数码照相机、数码摄像机、扫描仪、手写笔等设备获取，这些图形处理设备按照计算机能够接受的格式，对自然图像进行数字化处理，然后通过设备与计算机之间的接口传输到计算机，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。

当计算机将数字图像输出到显示器、打印机、电视机等设备时，又必须将离散化的数字图像合成为一幅图形处理设备能够接受的自然图像。

2. 图像的编码

1）二值图的编码

只有黑、白两色的图像称为二值图。图像信息是一个连续的变量，离散化的方法是设置合适的取样分辨率（采样），然后对二值图像中每一个像素用 1 位二进制数表示，就可以对二值图进行编码。一般将黑色点编码为 1，白色点编码为 0（量化），如图 5-25 所示。

图像分辨率（采样精度）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，表示每一条水平线上包含 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,图像就越清晰,图像输出质量也越好。同时,太高的图像分辨率会增加文件占用的存储空间。

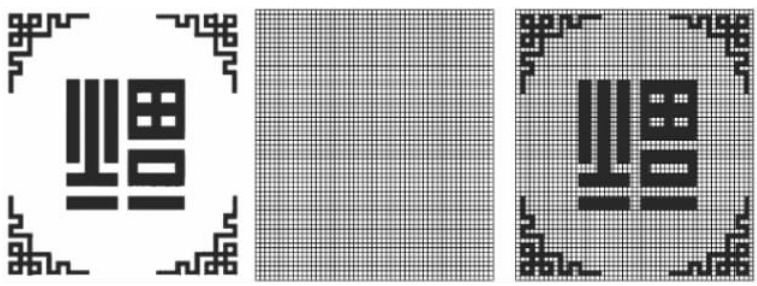


图 5-25 左:二值图 中:确定分辨率 右:二值图的数字化处理

2) 灰度图像的编码

灰度图的数字化方法与二值图相似,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差。图像中每个像素点的亮度值用 8 位二进制数(1 个字节)表示,如图 5-26 所示。

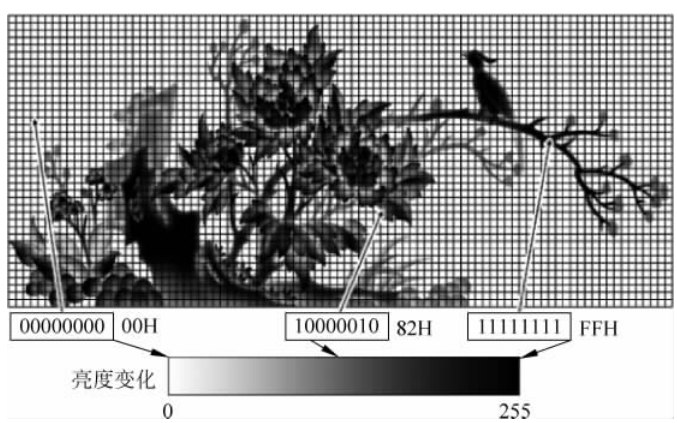


图 5-26 灰度图像的编码方式

3) 彩色图像的编码

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。因此,图像中每个像素点可以用 3 个字节进行编码。如图 5-27 所示,红色用 1 个字节表示,亮度范围为 0～255 个等级(R=0～255);绿色和蓝色也同样处理(G=0～255,B=0～255)。

【例 5-55】 如图 5-27 所示,一个白色像素点的编码为 R=255,G=255,B=255;一个黑色像素点的编码为 R=0,G=0,B=0;一个红色像素点的编码为 R=255,G=0,B=0;一个桃红色像素点的编码为 R=236,G=46,B=140 等。

采用以上编码方式,一个像素点可以表达的色彩范围为 2^{24}=1670 万种色彩,这时人眼

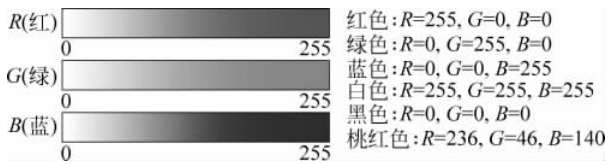


图 5-27　彩色图像的编码方式

已很难分辨出相邻两种颜色的区别了。一个像素点总计用多少位二进制数表示，称为色彩深度(量化精度)，例如，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 5-56】　对分辨率为 1024×768，色彩深度为 24 位的图片进行编码。

如图 5-28 所示，对图片中每一个像素点进行色彩取值(量化精度)，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就可以了。形成图片文件时，还必须根据图片文件格式加上文件头部。

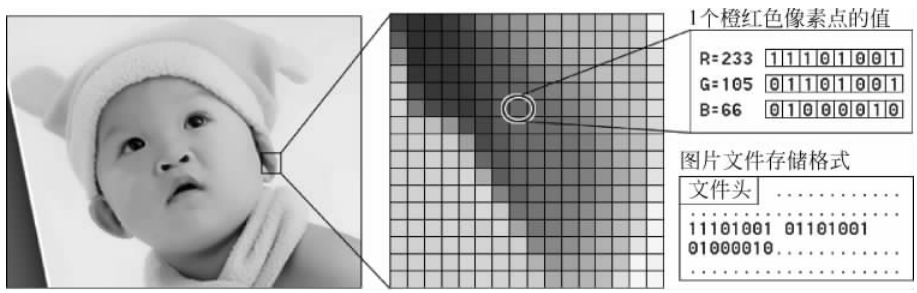


图 5-28　24 位色彩深度图像的编码方式(没有压缩时的编码)

3. 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显得是连续的；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的。

大部分情况下，点阵图像由数码相机、数码摄像机、扫描仪等设备获得，也可以利用图像处理软件(如 Photoshop 等)创作和编辑图像。

4. 矢量图形的编码

矢量图形(Graphic)使用直线或曲线来描述图形，矢量图以几何图形居多，它是一种面向对象的图形。矢量图形采用特征点和计算公式(如图 5-21 所示的二次贝塞尔曲线计算公式)对图形进行表示和存储。矢量图形保存的是每一个图形元件的描述信息，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 5-29 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。矢量图形中的曲线是由短的直线逼近的(插补),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部件(图元)是相对独立的,因而在矢量图形中可以只编辑修改其中的某一个物体,而不会影响图中其他物体。

图 5-29 左:AutoCAD 矢量图 中:3DMAX 矢量图 右:分形矢量图示例

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,将图形输入到计算机中并矢量化;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,也给多媒体应用带来了极大不便。

矢量图形主要用于线框形图片、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)、二维动画软件(如 Flash CS)、三维造型软件(如 3DMAX)等,都采用矢量图形作为基本图形存储格式。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。

5.3 压缩与纠错编码

5.3.1 信息量的度量

1. 熵的概念和熵递增原理

1854 年,德国科学家克劳修斯(Rudolf Clausius)首先引入熵(希腊语“变化”的意思)的概念,它是表示封闭体系中杂乱程度的一个量。

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,房间只会杂乱下去,决不会变得整齐。熵递增原理是自然界的基本规律,生物也离不开“熵递增原理”,生物需要从体外吸收负熵来抵消熵的增大。

热力学指出,对任何已知孤立的物理系统,热熵只会增加,不会减少。而信息熵则相反,信息熵只会减少,不会增加。热熵和信息熵互为负量,目前已证明,任何系统要获得信息必

须增加热熵来补偿。因此，获取信息必然伴随着能量的损失。

同样，一个大型系统软件随着功能越来越多，模块调用量的急剧增长，整个系统会逐渐碎片化，变得越来越无序，导致系统最终无法维护和扩展。因此，系统运行一段时间后，需要对系统进行有序化重构（版本升级），不断减少系统的“熵”，使系统不断进化。

【例 5-57】 网站设计中，一个 ASP 或 JSP 页面里，HTML 和脚本程序往往混合在一起，这时网站脚本代码越多，网站系统越混乱（熵增加），最终连开发者自己都无法理解。这时就需要对系统重新架构，可以引入新的开发模式（如 View Helper、视图助手），分离 HTML 代码和脚本语言，HTML 分离为视图，脚本分离为帮助类，然后整合在一起。通过重新分合，整个系统变得层次清晰，职责明确，系统的无序度大大降低。同时不同的开发人员（如前端设计员和后端程序员）可以负责不同部分，有效提高了软件开发效率。

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（Claude Elwood Shannon）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。香农信息论的特点是：只考虑信息符号本身出现的概率，与信息内容无关。

信息量大小与它的不确定性有关。例如，要搞清楚一件非常不确定的事，或是人们一无所知的事情，就需要了解大量的信息。信息是个抽象的概念，人们常常说信息很多，或者信息较少，但很难说清楚信息到底有多少。例如一本 50 万字的书籍到底有多少信息量？直到 1948 年，香农提出了“信息熵”的概念，才解决了对信息的度量问题。

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到信息论中。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 $S=\{E1, E2, \cdots, En\}$，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

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

式中，$H(X)$ 是信息熵，单位是比特（bit）；P_i 是每个随机事件的发生概率。信息熵的单位与公式中对数的底有关。不同对数底之间的转换关系为 $\log_a X = \log_b X / \log_b a$。

1）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 5-58】 π 是一个无限循环小数，如果希望 π 值的精度是十进制数的 30 位，那么最少需要多少位二进制数（信息熵）表示才能满足精度要求。

假设 π 值中每个数字符号出现的概率相同（等概率事件），根据对数换底公式：

$$\log_2(0)\log_{10}(10)/\log_{10}(2) = 1/0.301\,03 = 3.3219;$$

$$\log_2(2) = \log_{10}(2)/\log_{10}(2) = 1;$$

一位十进制数转换为二进制数需要的二进制符号位（信息熵）为

$$k = -\log_2(10)/-\log_2(2) = 3.3219/1 = 3.3219 \approx 3.4\text{（余量稍留大一点）}$$

由以上计算可知,每位十进制数转换为二进制数时,信息熵大约需为 3.4b。30 位十进制数 π 值转换成二进制数时,最少需要 30×3.4b=102b。

2) 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,信息熵也可以理解为不确定性。

【例 5-59】 假设足球世界杯决赛的是巴西队和美国队,根据经验,巴西队夺冠的概率是 80%,美国队夺冠的概率是 20%,根据信息熵公式,猜测谁能获得冠军需要的信息量为

$$\begin{aligned} H(X) &= -(0.8 \times \log_2(0.8) + 0.2 \times \log_2(0.2)) \\ &= -(0.8 \times (-0.3219) + 0.2 \times (-2.3219)) \\ &= -((-0.257) + (-0.464)) = 0.721\text{b} \end{aligned}$$

通过计算可以发现,巴西队夺冠的概率越高,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。如果巴西队 100%夺冠,那么信息熵等于 0,相当于没有任何信息。当两队夺冠概率都是 50%时,信息熵达到了最大值 1b。

3) 信息熵与编码长度

信息熵的总量也称为信息量,但是它与我们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。统计数据表明,几种语言单个字母的平均信息熵为法语 3.98b; 英语 4.03b; 汉语 9.65b。

【例 5-60】 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)。计算它们的信息熵:

$H_1 = 9.65 \times 4 = 38.6\text{b}$, $H_2 = 9.65 \times 11 = 106.15\text{b}$, $H_3 = 4.03 \times 44 = 177.32\text{b}$

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,S_2 的信息熵比 S_1 高 1 倍以上,S_3 的信息熵比 S_1 高 3 倍以上,也就说,在同一语义下,S_2 和 S_3 的编码长度要大大高于 S_1。

4. 最大信息熵原理

1957 年,杰尼斯(E. T. Jaynes)提出了最大熵原理,它的主要思想是:根据不完整的信息进行推断时,符合已知条件的概率分布可能不止一个时,应该选取符合这些条件,并且信息熵最大的概率分布。

最大熵原理指出,当需要对一个随机事件的概率分布进行预测时,我们的预测应当满足全部已知条件,而且对未知事物不做任何主观假设,没有任何偏见。在这种情况下,概率分布最均匀,预测风险最小,因为这时概率分布的信息熵最大。我们常说,不要把所有的鸡蛋放在一个篮子里,其实就是最大信息熵原理的一个朴素说法。匈牙利数学家希萨(Csiszar)证明,对任何一组不自相矛盾的信息,最大熵模型不仅存在,而且是唯一的。

【例 5-61】 拼音转汉字的概率判断。假如输入的拼音是 wang-xiao-bo,利用语言模型,根据有限的上下文(如前 2 个词),我们能给出 2 个最常见的名字“王小波”和“王晓波”。至于要唯一确定是哪个名字就难了,即使利用较长的上下文也做不到。根据最大熵原理,我们可以判断:如果通篇文章是介绍文学作品,作家王小波的可能性(概率)较大;如果文章是讨论台海两岸关系时,台湾学者王晓波的可能性(概率)会较大。

5.3.2　无损压缩编码

对于多媒体信息，可以通过数据压缩来消除原始数据中的冗余性，将它们转换成较短的数据序列，达到减少数据存储空间的目的。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。但是蓝色可能还会有不同的深浅，天空有时也可能被树木、山峰或其他对象掩盖，这些部分数据需要另外记录。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 2∶1～4∶1）大小。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. 常用压缩编码方法

常用的压缩编码算法如图 5-30 所示。

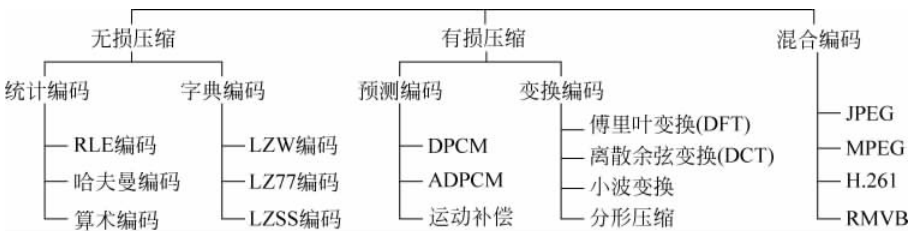


图 5-30　常用压缩编码算法

LZW、LZ77 编码属于字典模型的压缩算法，而 RLE、哈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次序有关，而与数据出现频率无关；统计模型压缩算法正好相反，它与数据的排列次序无关，而与数据出现的频率有关。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。如 WinRAR 就采用了字典压缩编码和哈夫曼压缩编码算法。

3. RLE 压缩编码原理

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 5-31 所示。

标记字节	字符重复次数	字符

图 5-31 RLE 编码方法

【例 5-62】 对文本字符串 AAAAABACCCCBCCC,采用 RLE 编码后为@5ABA@4CB@3C。标记字节@说明重复字符的开始,@后面的数字表示字符重复的次数,数字后面是被重复字符;没有重复的字符采用直接编码,不需要标记字节。

RLE 编码简单直观,编码和解码速度快,尽管 RLE 算法压缩效率很低,它还是得到了广泛应用。RLE 编码适用于信源字符重复出现概率很高的情况。许多图形和视频文件(如 BMP、AVI 等),以及 Winzip、WinRAR 等压缩软件,经常会用到 RLE 编码算法。

4. 哈夫曼压缩编码原理

哈夫曼(David A. Huffman)编码算法的基本原理是:频繁使用的字符用较短的编码代替,较少使用的字符用较长的编码代替,每个字符的编码各不相同,而且编码长度是可变的(变长编码)。例如在英文中,字母 e、t、a 的使用频率要大于字母 z、q、x,在对字母进行编码时,可以用较短的编码表示常用字母,用较长的编码表示不常用字母,这样每一个字母都有一个唯一的编码。编码的长度是可变的,而不是像 ASCII 码那样都用 8 位表示。哈夫曼编码现在广泛用于数据压缩、数据通信、多媒体技术等各个领域。

【例 5-63】 对字符串"I am a teacher"进行哈夫曼编码。

设信号源字符串为:X={空格,a,e,I,m,t,c,h,r}(按字符出现概率大小排列)。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 },哈夫曼编码过程如图 5-32 所示。

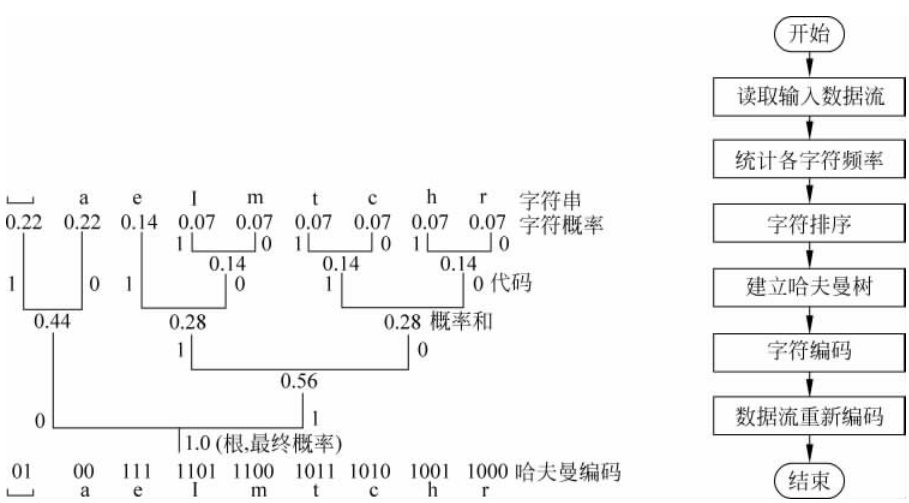


图 5-32 左:字符串"I am a teacher"的哈夫曼树编码过程 右:哈夫曼编码流程

哈夫曼编码算法步骤如下:

步骤 1:如图 5-32 所示,按符号在文本中出现的概率由大到小顺序排列。

步骤 2:将概率小的两个符号组成一个节点,如空格和 a,I 和 m,t 和 c 等。

步骤 3：将相邻两个概率组合相加，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）。

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

步骤 5：代码分配。代码从根节点（哈夫曼树底部）开始向上分配（如空格编码＝01）。代码左边标 1 或 0 无关紧要，因为结果仅仅是代码不同，而代码平均长度是相同的。

最终，每个字符的哈夫曼编码为：（空格）＝01，a＝00，e＝111，I＝1101，m＝1100，t＝1011，c＝1010，h＝1001，r＝1000。

字符串“I am a teacher”共 14 个字符，如果用 ASCII 码传送，每个字符 8 位，共需 112 位（14×8）。如果字符串 9 个不同符号采用 4 位二进制编码表示，传送该字符串要 56 位。如果采用哈夫曼编码，只需要 42 位，哈夫曼编码与 ASCII 码的比较如图 5-33 所示。

压缩文本	I		a	m		a		t	e	a	c	h	e	r
哈夫曼码	1101	01	00	1100	01	00	01	1011	111	00	1010	1001	111	1000
ASCII码	49	20	61	6D	20	61	20	74	65	61	63	68	65	72

图 5-33　哈夫曼编码（二进制）与 ASCII 码（十六进制）的比较

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”，这实际就是信息压缩。人们之所以能够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）字典编码算法思想

字典（严格说是词典）压缩编码是把信息中出现频率较高的数字组合做成一个对应的字典列表，并用特殊代码来表示这个数字。字典编码有以下两种实现方法。

第一种是查找正在压缩的字符序列是否在前面出现过，如果是则用“指针”指向出现过的字符串，替代重复出现的字符串。采用这个算法思想的有 LZ77 和 LZSS 算法。

第二种方法是对输入字符创建一个“短语字典”，在编码过程中，如果遇到在字典中出现的“短语”时，就输出字典中短语的“索引号”，而不是短语本身。LZ78 和 LZW 算法采用这种算法思想。

3）LZW 压缩编码算法

LZW 算法由 Lemple、Ziv、Welch 三人共同创造，用他们的名字命名。LZW 算法中，首先建立一个字典（字符串表），把每一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 5-64】 如 print 字符串,如果压缩时用数字标号 129(扩展 ASCII 码)表示,只要它再次出现,均用 129 表示,并将 print 字符串和数字标号 129 存入字典中。解码时遇到数字标号 129,即可从字典中查出 129 所代表的字符串是 print。

【例 5-65】 对文本"good good study,day day up"进行 LZW 算法压缩编码。

对原始文本中的字符串进行扫描,生成的字典如图 5-34 所示。为了容易理解起见,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 5-34 LZW 算法扫描文本后生成的字典

利用 LZW 算法,对文本信息进行压缩的最终编码如图 5-35 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

说明:第1个单词good不采用编码5涉及扫描窗口等问题,不详述。

图 5-35 利用 LZW 算法对文本信息进行压缩的编码

如图 5-34 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 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,则压缩比率越高,但系统开销也越高。LZW 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)甚至更多位来表示。

LZW 算法适用范围是原始数据中最好有大量的子串多次重复出现,重复越多,压缩效果越好,反之则越差。一般情况下 LZW 算法可实现 2∶1～3∶1 的压缩比。

目前流行的 GIF、TIF 等图像文件采用了 LZW 压缩算法;WinRAR,Winzip 等压缩软件也采用了 LZW 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 压缩算法。LZW 算法广泛用于文本数据、程序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 5-66】 将某一个子目录下所有文件进行压缩的 Python 程序如下。

```
import zipfile                                          #导入内置压缩包
import os                                               #导入内置系统包
f = zipfile.ZipFile('my.zip','w',zipfile.ZIP_DEFLATED)  #my.zip=压缩文件名,w=写入
startdir = 'd:/test/pic'                                #ZIP_DEFLATED=压缩,f=临时变量
for dirpath, dirnames, filenames in os.walk(startdir):  #压缩 pic 目录下所有子目录和文件
    for filename in filenames:                          #循环向压缩文件中添加文件和文件夹
        f.write(os.path.join(dirpath,filename))         #将临时变量写入压缩文件
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，因为其中包含的数据往往多于人们的视觉系统和听觉系统所能接收的信息，丢掉一些数据而不至于对声音或者图像所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占用的空间。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(ISO)和国际电信联盟(ITU)共同成立的联合图片专家组(JPEG)，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(DPCM)的预测编码；第二部分是有损压缩，采用离散余弦变换(DCT)和哈夫曼(Huffman)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 图像编码原理

JPEG 图像编码原理非常复杂，压缩编码分为以下 4 个步骤，如图 5-36 所示。

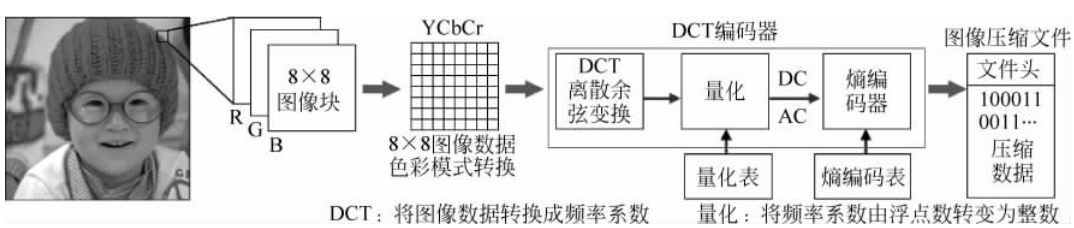


图 5-36　JPEG 图像压缩编码流程

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。

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

步骤 3：量化。将频率系数由小数转换为整数。

步骤 4：熵编码。其编码原理和过程更加复杂(不详述)。

4. MPEG 动态图像压缩标准

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 5-67】 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(640×480×24b)/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×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。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 30 帧/秒,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。据统计,在灰度视频图像中,大部分相邻图像帧之间,同一位置的像素,数据绝对值之差超过 3 的数据不大于 4%,可见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 5-68】 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 5-37 所示,视频图像中一个画面大致分为 3 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)。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。

图 5-37　视频画面的三个基本区域

6. MPEG 视频图像排列

MPEG 标准将图像分为 3 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)。

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 帧的采用类似 JPEG 的压缩算法，I 帧的压缩比较低。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/秒(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(1080P)视频中计算量很大，对计算机的要求很高。

3) 双向插值帧 B

B 帧是双向预测插值帧，它利用 I 帧和 P 帧做参考，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。

典型的 MPEG 视频图像帧序列如图 5-38 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检错和纠错编码。采用信道差错控制编码的目的是为了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**。

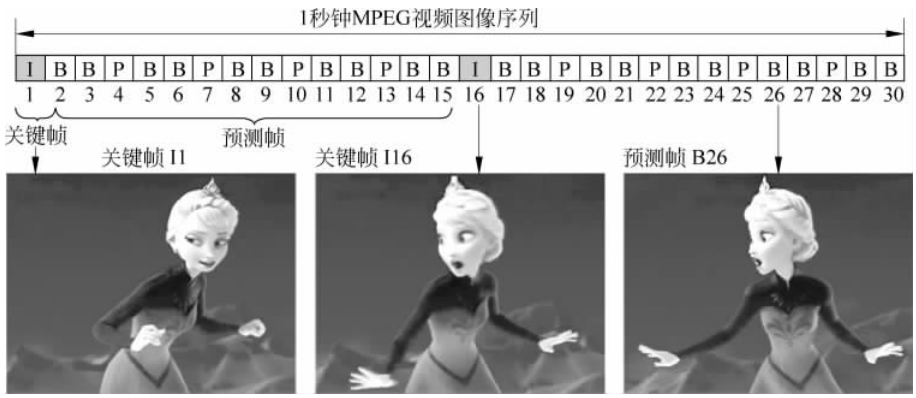


图 5-38 典型 MPEG 视频图像帧显示序列

差错控制编码有两类,一类是 ARQ(自动请求重发),另一类是 FEC(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,特别适合于网络通信。绝大多数情况下,计算机使用检错码,出现错误后请求对方重发(ARQ);只有在单工信道情况下,才会使用纠错编码。

2. 出错重传的差错控制方法

ARQ 采用出错重传的算法思想。如图 5-39 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 5-39 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,它分为奇校验或偶校验。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。奇偶校验的编码规则如表 5-10 所示。

表 5-10 奇偶校验规则表

奇 校 验		偶 校 验	
数据位中 1 的个数	校验值	数据位中 1 的个数	校验值
奇数个	0	偶数个	0
偶数个	1	奇数个	1

【例 5-69】 字符 A 的 ASCII 码为 01000001，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，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。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，这种方法就检测不出来了。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，美国火星探测器“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。

计算机领域常用的前向纠错编码是海明码(R. W. Hamming)，可以利用海明码进行检测并纠错。下面我们用一个简单的例子来说明纠错的基本原理，它虽然不是海明码，但是它们是算法思想相同，都是利用冗余编码来达到纠错的目的。

【例 5-70】 如图 5-40 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。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。

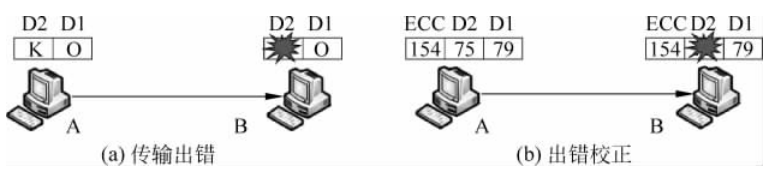


图 5-40　传输出错与出错校正示意图

如图 5-40(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 校验编码

CRC(循环冗余校验)编码是最常用的一种差错校验码,它的特点是检错能力极强,开销小,易于用编码器或检测电路实现。从检错能力来看,它的出错概率非常小。从性能上和开销上考虑,CRC 编码远远优于奇偶校验编码。因此,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 软件等采用了 CRC-32 编码;磁盘文件校验采用了 CRC-16 编码;GIF、TIFF 等图像文件也采用 CRC 编码作为检错手段。

1) CRC 校验原理

CRC 校验的算法思想是:先在待发送的数据帧后面附加一个数(校验码),生成一个新数据帧发送给接收端。当然,这个附加的校验码不是随意的,校验码要使生成的新数据帧能与发送端和接收端共同选定的某个特定数整除。数据帧到达接收端后,对接收的数据帧除以这个选定的除数(模 2 除法,即异或运算)。因为发送端已对附加的校验码做了"去余"处理,所以校验结果应该没有余数。如果有余数,则表明该数据帧在传输过程中出现了差错。

2) 生成多项式

CRC 校验的除数可以随机选择,也可按国际标准选择(如表 5-10 所示),但最高位和最低位系数必须为 1。除数通常以多项式表示,称为"生成多项式"。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。

【例 5-71】 码组 1100101 可以表示为:$1 \cdot x^6 + 1 \cdot x^5 + 0 \cdot x^4 + 0 \cdot x^3 + 1 \cdot x^2 + 0 \cdot x + 1$,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:$G(x) = x^6 + x^5 + x^2 + 1$。

生成多项式按最高阶数 m,将 CRC 称为 CRC-m。如 CRC-8、CRC-16 等。生成多项式 $G(x)$一般按标准进行选择,常用的 CRC 生成多项式标准如表 5-11 所示。

表 5-11 常用 CRC 生成多项式国际标准

标准名称	生成多项式	十六进制简记式 *	应用案例
CRC-4-ITU	x^4+x+1	0x3	ITU-G.704
CRC-8-ITU	x^8+x^2+x+1	0x07	ATM,HEC,ISDN,HEC
CRC-16	$x^{16}+x^{15}+x^2+1$	0x8005	IBM SDLC
CRC-16-ITU	$x^{16}+x^{12}+x^5+1$	0x1021	ISO HDLC, ITU-X. 25, V. 34, PPP-FCS
CRC-32	$x^{32}+x^{26}+x^{23}+\cdots+x^2+x+1$	0x04C11DB7	IEEE 802.3,RAR,ZIP,IEEE 1394

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 去掉了。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 0111B。

3) CRC 校验码计算案例

【例 5-72】 为了说明简单起见,假设待发送的数据为字符 a,a 的 ACSII 码=0110 0001;假设收发双方选择的生成多项式为 CRC-16-ITU,计算字符 a 的 CRC 校验码。

步骤 1:将生成多项式转换成二进制数。

CRC-16-ITU $= G(x) = x^{16} + x^{12} + x^2 + 1 =$ 11021H = 1 0001 0000 0010 0001B。

步骤 2:CRC 校验码位数=生成多项式位数−1=17−1=16 位。在原始数据 0110 0001 后面加 16 个 0(16 位校验码),得到被除数为 0110 0001 0000 0000 0000 0000。

步骤 3：如图 5-41 所示，将步骤 2 得到的数作为被除数，生成多项式作为除数，进行模 2 除法(异或运算)，得到的余数 0111 1100 1000 0111 即为 CRC 校验码。

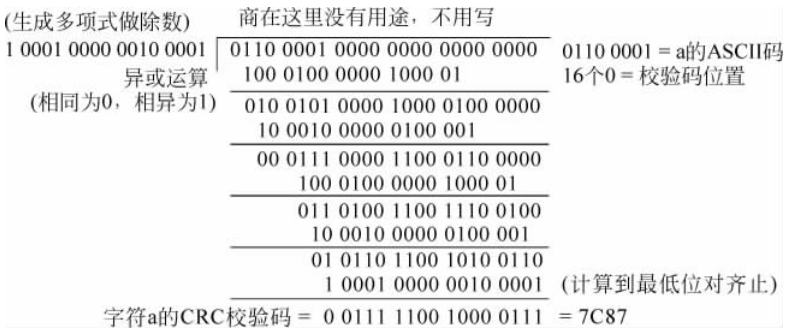


图 5-41　字符 a 的 CRC 校验码计算

步骤 4：将得到的 CRC 校验码附在原始数据帧 0110 0001 后面，得到的新数据帧为 0110 0001 **0111 1100 1000 0111**(阴影部分为 CRC 校验码)，然后把新数据帧发送到接收端。

步骤 5：新数据帧到达接收端后，接收端会把这个数据帧再用上面选定的除数 1 0001 0000 0010 0001(生成多项式)进行异或运算，验证余数是否为 0。如果余数＝0，则证明该数据帧在传输过程中没有出错；如果余数≠0，则说明传输过程中数据帧出现了差错。

从以上讨论来看，CRC 校验似乎非常麻烦，其实用逻辑电路实现非常简单。

【例 5-73】　将指定字符串生成 CRC 检验编码，Python 指令如下。

```
>>> import zlib                    # 导入内置压缩包
>>> s = b'hello,word!'             # 字符串赋值,前缀 b 说明字符串数据类型为字节(Byte)编码
>>> print(zlib.crc32(s))           # 进行 CRC32 校验编码
3035098857                         # 输出 CRC32 编码
```

4) CRC 校验码的特征

CRC 校验码能检错和纠错，但是纠错效率不高，或者说对计算资源要求过高。计算机网络大多采用 CRC 进行检错，发现错误后则采用出错重传(ARQ)技术。

采用 CRC 编码时，信息码长度可任意选定，校验码长度取决于选用的生成多项式。

采用 CRC 编码时，传送信号任一位发生错误时，生成多项式做除法后余数不为 0。

采用 CRC 编码时，传送信号不同位发生错误时，生成多项式做除法后余数不同。

采用 CRC 编码时，对生成多项式做除法后的余数继续做除法时，余数会循环出现。

5.4　逻辑运算与应用

5.4.1　基本逻辑运算

德国数学家莱布尼兹(Leibniz)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 Bool)1847 年创立了布尔代数；美国科学家香农(Shannon)将布尔代

数用于分析电话开关电路。布尔代数为解决工程实际问题提供了坚实的理论基础。

1. 逻辑运算的特点

在布尔代数中,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。例如,命题判断中的“真”与“假”,程序流程图中的“是”与“否”,工业控制系统中的“开”与“关”,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)。通过这三种基本逻辑运算,可组合出任何其他逻辑关系,在逻辑运算中,非运算的优先级最高,与运算次之,或运算最低。

一个逻辑关系往往有多种不同的逻辑表达式,可以利用布尔代数的基本规律和一些常用的逻辑恒等式,对逻辑表达式进行化简操作,以简化数字电路的设计。

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,逐位按规则运算即可。

(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。

逻辑运算由于具有以上特性,因此特别适合用于计算机的逻辑电路设计。

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(真真得真)。真值表是描述逻辑关系的直观表格。如果利用电子元件制造出符合与运算规则的电子器件,我们将这个器件为“与门”,与门的表示符号如图 5-42 所示(GB 为国家标准)。与运算常用于对某位二进制数进行复位(设置为 0)运算。

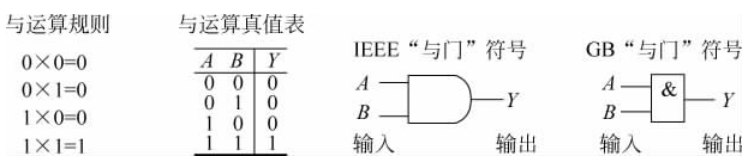


图 5-42 与运算规则和表示符号

【例 5-74】 10011100 AND 00111001=00011000。

```
    10011100    (输入 A)
AND 00111001    (输入 B)
    00011000    (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(假假得假)。如果利用电子元件制造出符合或运算规则的电子器件,我们将这个器件为“或门”,或门的表示符号如图 5-43 所示。或运算常用于对某位二进制数进行置位(设置置为 1)运算。

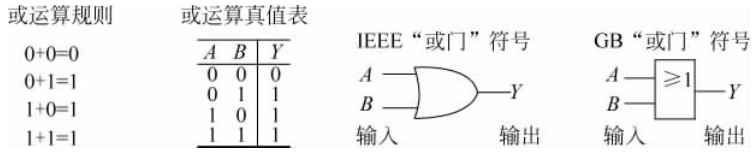


图 5-43　或运算规则和表示符号

【例 5-75】 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(也称为“反相”),逻辑表达式为：$Y=\overline{A}$,非运算也称为“取反”,规定在逻辑运算符上方加上画线“—”表示。非运算可以理解为逻辑值取反。利用电子元件制造出符合非运算规则的电子器件称为“非门”,非门表示符号如图 5-44 所示。在逻辑门符号中,一般用“小圆圈”表示逻辑值取反。非运算常用于对某部分(1～n 字节)二进制数进行全部反转(求全反)运算。

图 5-44　非运算规则和表示符号

【例 5-76】 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,异或运算通常用符号⊕表示,运算规则为：0⊕0＝0,0⊕1＝1,1⊕0＝1,1⊕1＝0。可以简单理解为：相同为 0,相异为 1(同假异真)。异或门符号和真值表如图 5-45 所示。异或运算的应用是对某位二进制数进行反转运算。

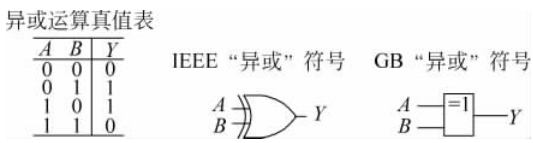


图 5-45　异或运算真值表和表示符号

【例 5-77】 10011100 XOR 00111001＝10100101。

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

5.4.2 命题逻辑演算

1. 数理逻辑概述

1) 数理逻辑的定义

逻辑是探索有效推理的学科,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”,可以将推理过程像数学一样利用公式来进行计算。

数理逻辑(又称为符号逻辑)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,所谓数学方法是指数学采用的一般方法,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。

2) 数理逻辑与计算机科学的关系

数理逻辑的研究分包括古典数理逻辑(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的理论,模型论主要研究形式语言与数学模型之间的关系,公理化集合论主要研究集合论中无矛盾性问题。数理逻辑和计算机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。许多计算机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。

数理逻辑和计算机的发展有着密切的联系,它为机器证明、自动程序设计、计算机辅助设计等计算机应用和理论研究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(对一个函数计算结果的断言可类比于一个逻辑定理,计算这个值的程序可类比于这个定理的证明)说明了“证明”和“程序”的等价性。柯里-霍华德同构经常被表述为“证明即程序”。

3) 计算的基本逻辑操作

现代数理逻辑已经证明,当逻辑状态多于两种时,其通用模型的基本逻辑有两个:一个是从一种状态变为另一种状态(逻辑开关);另外一个是在两种状态中,按照某种规则(如比较大小)有倾向性的选择其中一种状态(逻辑比较器)。依据这两种逻辑,可以表达任意多状态的任意逻辑关系,即任意多状态的逻辑是完备的。任何数学运算都可以用两个运算关系来联合表达:加法和比较大小(减法可以通过补码的加法来实现)。

2. 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,命题真值和假值通常用大写英文字母 T(True)和 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“连接词”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 5-78】 下列陈述句都是命题。

```
8 小于 10。                    #命题真值为真
8 大于 10。                    #假命题也是命题,是真值为假的命题
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
```

明年 10 月 1 日是晴天。　　#真值目前不知道，但真值是确定的
公元 1100 年元旦下雨。　　#可能无法查明真值，但真值是确定的

【例 5-79】 下列句子不是命题。

8 大于 10 吗？　　#疑问句，非陈述性语句，不是命题
天空多漂亮！　　#感叹句，非陈述性语句，不是命题
禁止喧哗。　　#命令句，非陈述性语句，不是命题
$x>10$　　#x 值不确定，因此命题真值不确定
$c^2=a^2+b^2$　　#方程不是命题
这句话是谎言。　　#悖论不是命题

3. 逻辑连接词

1）逻辑连接词的含义

命题演算是研究命题如何通过一些逻辑连接词，构成更复杂的命题。命题演算利用计算思维中**“抽象”**的方法，把命题看作运算对象，如同代数中的数字、字母或代数式，而把逻辑连接词看作运算符号，就像代数中的“加、减、乘、除”那样，那么由简单命题组成复合命题的过程，就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、狄摩根定律、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连接词表示命题的推理规则（如表 5-12 所示）。使用连接词可以将若干个简单句组合成复合句。

表 5-12　逻辑连接词与含义

连接词符号	说　明	命题案例	命题读法	命题含义
¬	非（NOT）	$\neg P$	非 P	P 的否定（逻辑取反）
∧	与（AND）	$P \wedge Q$	P 并且 Q	P 和 Q 的合取（逻辑乘）
∨	或（OR）	$P \vee Q$	P 或者 Q	P 和 Q 的析取（逻辑加）
→	如果……则……（if-then）	$P \rightarrow Q$	若 P 则 Q	P 蕴含 Q（单向条件）
↔	当且仅当	$P \leftrightarrow Q$	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 5-13 所示。

表 5-13　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	$P \wedge Q$	$P \vee Q$	$\neg P$	$\neg Q$	$P \rightarrow Q$	$P \leftrightarrow Q$
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

2) 连接词的优先级

连接词的优先级按“否定→合取→析取→蕴含→等价”由高到低。

同级的连接词,优先级按出现的先后次序(从左到右)排列。

如果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。

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

4. 逻辑连接词“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必含有某种因果关系;但在数理逻辑中,允许前提和结论之间无必然因果关系,只要可以判断逻辑值的真假即可。

【例 5-80】 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:$P \rightarrow Q$ 成立。

2) 善意推定

日常生活中,当前件为假时($P=0$),$P \rightarrow Q$ 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 $Q=1$),$P \rightarrow Q$ 总为真命题(即 $P \rightarrow Q=1$),这有没有道理呢?

【例 5-81】 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 $P \rightarrow Q$(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(如表 5-12 所示),命题 $P \rightarrow Q$ 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 5-82】 将下列命题用命题逻辑表示。

(1) 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:$P \vee Q$。

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

命题演算 2:如果 $P=0$(假),$Q=0$(假);则 $P \vee Q=0 \vee 0=0$(命题为假)。

(2) mm 既漂亮又会作。

令 P=mm 漂亮,Q=mm 会作,符号化命题为:$P \wedge Q$。

(3) 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:$\neg P$。

(4) 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:$P \rightarrow Q$。

(5) 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:$P \leftrightarrow Q$。

(6) 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：$P \wedge (\neg Q)$。

(7) 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：$\neg P \rightarrow Q$。

(8) 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：$(\neg P \rightarrow \neg Q) \wedge (P \rightarrow Q)$。

5.4.3　谓词逻辑演算

1. 命题函数

命题逻辑不能充分表达计算机科学中的许多陈述，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。

计算机程序中常见的语句如：“$x>5$”“$x=y+2$”“计算机 x 被攻击”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，它们就成了一个命题。

【例 5-83】 在语句 $x>5$ 中，包含了两个部分，第一部分变量 x 是语句的主语，第二部分是谓词“大于 5”。我们用 $p(x)$ 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 5-84】 令 $G(x,y)$表示“x 高于 y”，而 $G(x,y)$是一个二元谓词(2 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

原子命题是最简单的谓词逻辑命题，它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)。

(1) **个体是独立存在的物体**，它可以是抽象的，也可以是具体的。例如，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。

(2) **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 5-85】 “$x=y+2$”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，一般使用如下符号进行表示。

(1) 常量符号一般用 $a,b,c,\cdots$表示，它可以是集合中的某个元素。

(2) 变量符号一般用 $x,y,z,\cdots$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 $f,g,\cdots$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 $P,Q,R,\cdots$表示,n 元谓词表示为 $P(x_1,x_2,\cdots,x_n)$。

4. 量词

除了个体词和谓词外,组成命题的成分还有量词。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""有的水生动物是肺呼吸的"这两个命题中,"所有""有的"都是量词,其中"所有"是全称量词,"有的"是存在量词。

在汉语中,"所有""一切""凡"等表示全称量词;"有的""有""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"有的 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($\forall$、$\exists$)和命题连接词($\wedge$、$\vee$、$\rightarrow$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。谓词逻辑中的命题比命题逻辑更为复杂,数量也非常多,相应公式的数量是无穷的。

【例 5-86】 用谓词逻辑表示以下命题的公式。

命题"所有阔叶植物是落叶植物"的公式为$(\forall x)(F(x)\rightarrow G(x))$。

命题"有的水生动物是肺呼吸的"的公式为$(\exists x)(F(x)\wedge G(x))$。

命题"一切自然数有大于它的自然数"的公式为$(\forall x)(F(x)\rightarrow(\exists y)(F(y)\wedge G(x,y))$。

谓词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。

谓词演算只涉及符号、符号序列、符号序列的变换等,完全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、可证明性等概念,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 5-87】 用谓词逻辑表示:ZhangFei 是计算机专业的学生,他不喜欢编程。

定义:COMPUTER(x)表示 x 是计算机系的学生;

定义:LIKE(x, y)表示 x 喜欢 y;

谓词公式为:COMPUTER(ZhangFei)$\wedge\neg$LIKE(ZhangFei, Programing)。

【例 5-88】 设机器人(robot)在教室(room),A、B 为两张桌子(table),桌子 A 上放了一个盒子(box),让机器人把盒子从 A 放到 B 上,试用谓词逻辑描述状态。

定义谓词:TABLE(x)表示 x 是桌子;EMPTY(y)表示 y 手中是空的;AT(y, z)表示 y 在 z 旁边;HOLD(y, w)表示 y 手中拿着 w;ON(w, x)表示 w 放在 x 上。其中变量的取值范围是 x {A, B};z {A, B, alcove};w {box};y={robot}。谓词逻辑描述如下:

初始状态的谓词逻辑：		目标状态的谓词逻辑：	
AT(robot, room)	＃机器人在教室里	AT(robot, room)	＃机器人在教室里
EMPTY(robot)	＃机器人手中是空的	EMPTY(robot)	＃机器人手中是空的
ON(box, A)	＃盒子放在 A	ON(box, B)	＃盒子放在 B
TABLE(A)	＃A 是桌子	TABLE(A)	＃A 是桌子
TABLE(B)	＃B 是桌子	TABLE(B)	＃B 是桌子

6. 谓词逻辑的特点

1）谓词逻辑的优点

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；逻辑推理可以保证知识库中新旧知识在逻辑上的一致性和所得结论的正确性；逻辑推理方法不依赖于任何具体领域，具有较大的通用性。

2）谓词逻辑的缺点

难于表示过程和启发式的知识，不能表示具有不确定性的知识；由于缺乏组织原则，使得知识库难于管理；当事实的数量增大时，在证明过程中可能产生计算的“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。

5.4.4　逻辑运算应用

逻辑运算广泛用于计算机各个领域，如计算机硬件电路都采用逻辑器件进行设计；在程序设计中，逻辑运算得到了广泛应用；程序编译时，采用逻辑推理来分析程序语法。

1. 逻辑运算在数据库查询中应的用

逻辑运算中的“与”“或”“非”可以用来表示日常信息中的“并且”“或者”“除非”等思想。例如，我们需要在数据库中查询信息时，就会要用到逻辑运算语句。

【例 5-89】 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。

查询语句为：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

逻辑运算通常用于对图形做某些操作。如图 5-46 所示，将 2 个相交的图形进行或运算时，可以将 2 个图形合并成一个图形；将 2 个相交的图形进行异或运算时，可以提取 2 个图形中的一个子图；将 2 个相交的图形进行与运算时，可以提取其中相交部分的子图；将一个图形进行非运算时，就可以得到与这个图形色彩相反（反相）的图形。

3. 逻辑运算在集合中的应用

【例 5-90】 如图 5-47 所示，设集合 A 包含“全集”中所有偶数（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 $A \cap B$（在集合 A AND B 中所有的元素）

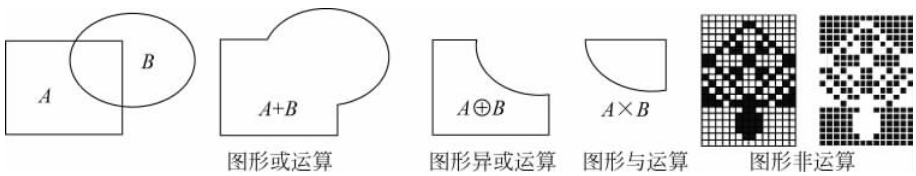


图 5-46　逻辑运算在图形处理中的应用

图 5-47　集合 A 和集合 B 之间的逻辑关系

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。加法运算是最基本的算术运算，因此加法器是 CPU 的核心单元。我们利用前述的逻辑门电路，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

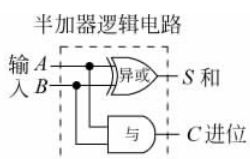
在半加器中，暂时不考虑低位进位的情况。半加器有两个输入端：加数 A、被加数 B；两个输出端：和 S、进位 C。半加器逻辑电路如图 5-48 所示，它由一个“异或门”和一个“与门”组成，考察真值表中 C 与 A、B 之间的关系是“与”的关系，即 $C=A\times B$；再看 S 与 A、B 之间的关系，也可看出这是“异或”关系，即 $S=A\oplus B$。因此，可以用“与门”和“异或门”来实现真值表的要求。

【例 5-91】 在图 5-48 半加器电路中，验证两个 1 位二进制数相加结果是否正确。

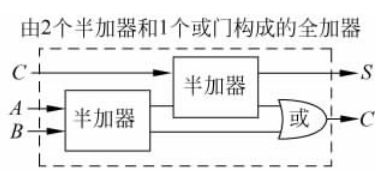
半加器真值表

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

(a) 1位半加器真值表



(b) 1位半加器逻辑电路



(c) 1位全加器逻辑电路

图 5-48　半加器真值表及逻辑电路设计

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。

2）全加器逻辑电路设计

全加器有多种逻辑电路形式。如图 5-48 所示，可用 2 个半加器和 1 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 语言设计。

【例 5-92】 用 VHDL 语言设计 1 个一位二进制全加器逻辑电路。

```
library IEEE;                                  -- 调用 IEEE 库函数( -- 为 VHDL 注释符)
use IEEE.STD_LOGIC_1164.all;                   -- 调用运算程序包
entity adder is port(a, b,ci : in bit;         -- 加法器实体定义,a、b、ci 为输入接口
     sum, co : out bit);                       -- sum 为输出接口
end adder;                                     -- 加法器实体定义结束

architecture all_adder of adder is             -- 加法器结构体
begin                                          -- 结构体开始
     sum <= a xor b xor ci;                    -- 和 sum 逻辑关系,<= 为赋值运算,xor 为异或逻辑
     co <= ((a or b) and ci) or ( a and b);    -- 进位 co 逻辑关系,or 为或逻辑,and 为与逻辑
end all_adder;                                 -- 结构体结束
```

3）4 位加法器电路设计

如图 5-49 所示，如果要搭建一个 4 位加法器，只需将 4 个一位全加法器连接即可。

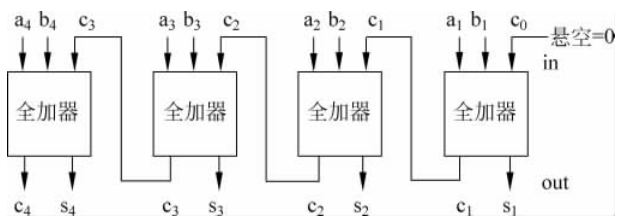


图 5-49　4 位加法器逻辑电路框图

将 n 个一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。

5. 谓词逻辑为数据库提供理论基础

关系代数中，可以用并运算和差运算表示插入和删除操作。在谓词逻辑中，并运算相当于连接词“或者”；差运算相当于连接词“并且”。如果设

$$R = \{(x_1, x_2, \cdots, x_n) \mid P(x_1, x_2, \cdots, x_n)\}$$

$$S = \{(x_1, x_2, \cdots, x_n) \mid Q(x_1, x_2, \cdots, x_n)\}$$

则有如表 5-14 所示关系。

表 5-14　对应关系

基本操作	关系代数	逻 辑 公 式
插入	$R \cup S$	$\{(x_1, x_2, \cdots, x_n) \mid P(x_1, x_2, \cdots, x_n) \vee Q(x_1, x_2, \cdots, x_n)\}$
删除	$R - S$	$\{(x_1, x_2, \cdots, x_n) \mid P(x_1, x_2, \cdots, x_n) \wedge \neg Q(x_1, x_2, \cdots, x_n)\}$

修改由删除和插入两个操作组合而成,因此删除、插入既然可由谓词描述,修改也就可以由谓词描述。同样,投影、选择、笛卡儿乘积都可以用谓词公式表示。

这样,我们把关系数据库中的数据子语言变成数理逻辑中的谓词公式,因而可以用谓词公式研究数据子语言,也可用谓词公式对数据子语言进行优化。由于关系代数、谓词逻辑、数据库三者之间建立了联系,使得关系数据库具有了坚实的理论基础。

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 个字节存储,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 请对中式英语 no zuo no die(不作死就不会死)进行 LZW 算法编码。

5-10 一个英文字符需要 7 位 ASCII 码,LZW 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 算法时,the 字符的压缩比是多少?