# 第二部分 数理逻辑基础（逻辑思维）

逻辑思维创新发展故事

进制的选取

1. 数的表示、存储与运算

### 计算机数据组织简介

计算机的应用领域及其广泛，但不论其应用在什么地方，信息在机器内部的形式都是一致的，即均位0和1组成的各种编码。这些二进制数字也被称为位(bit)，形成了第三次工业革命的基础。如今大家熟悉并使用了1000多年的十进制（以10为基数）起源于印度，在12世纪被阿拉伯数学家改进，并于13世纪被意大利数学家Leonardo Pisano（大约公元1170—1250，更为大家所熟知的名字是Fibonacci）带到西方。正常人类拥有10个手指头，所以使用十进制表示法是很自然的事情。但是当选择在机器上处理信息的数制时，使用二进制来工作的效果更好。二进制信号能够很容易地被表示、存储和传输，例如，在早期计算机上使用的穿孔卡片上用有洞和无洞来表示1和0、导线上的高电压或低电压，或者顺时针或逆时针的磁场。对二进制信号进行存储和执行计算的电子电路非常简单和可靠，并且经过集成电路的不断发展，制造商能够在一个单独的硅片上集成数百万甚至数十亿个这样的电路。

然而单个位不是非常有用，但是把这些位组合在一起，再人为赋予这些不同的可能位的组合以意义，我们就能够表示任何有限集合的元素。例如，使用一个二进制数字系统，我们可以用不同位的组合来对负数进行编码。我们还可通过使用标准定义的字符码来对文档中的字母和其他符号进行编码。在本章中，我们将讨论这两种编码，以及带符号数和其他音视频的编码等。

我们研究了三种最重要的数字表示。无符号（unsigned）编码基于传统的二进制表示法，能够表示大于或者等于0的数字。补码（two’s-complement）编码是表示有符号整数的最常见方式，有符号整数就是正数或者负数。浮点数（floating-point）编码是表示实数的以2为基数来表示的科学计数法版本。计算机用这些不同的数的表示方法来实现算术运算，例如加法和乘法，类似于对应的整数和实数来运算。

计算机的表示法是用有限数量的位来对一个数字编码，因此，当结果太大以至于不能表示时，某些运算就会产生溢出(overflow)。溢出会造成错误的结果。例如，在今天的大多数计算机上使用32位来表示数据类型int，计算表达式200\*300\*400\*500会得出结果为-884 901888。这显然违背了整数运算的特性，一些正数的乘积，结果不会得到负数。

另一方面，计算机在运算整数时满足我们熟知的真正整数运算的许多性质。例如，利用乘法的结合律和交换律，计算下面任何一个C表达式，都会得出结果-884 901888:

( 500 \* 400 ) \* ( 300 \* 200 )

( ( 500 \* 400 ) \* 300 ) \* 200

( ( 200 \* 500) \* 300 ) \* 400

400 \* ( 200 \* ( 300 \* 500 ) )

虽然上述表达式得到的结果都是错的，但是它们的结果至少是一致的！

浮点运算有完全不同的数学属性。虽然在溢出时会产生特殊的值，但是一组正数的乘积总是正的。由于表示的精度有限，浮点运算是不可结合的。比如在大多数机器上，表达式(3.14+1e20)-1e20求得的值会是0.0，而 3.14+(le20-le20)求得的值会是3.14。整数运算和浮点数运算会有不同的数学属性是因为它们处理数字表示有限性的方式不同——整数的表示虽然只能编码一个相对较小的数值范围，但是这种表示是精确的；而浮点数虽然可以编码一个较大的数值范围，但是这种表示只是近似的。

通过研究数字的实际表示情况，我们能够了解可以表示的值的范围和不同算术运算的属性。为了使编写的程序能在全部数值范围内正确工作，而且具有跨平台的可移植性，了解这种属性是非常重要的。

计算机用几种不同的二进制表示形式来编码数值。在本章中，我们会介绍这些编码，并且教你如何推出这些数字的表示。通过直接操作的数字的位级表示，我们得到了几种算术运算的方式。理解这些技术对于理解编译器产生的机器码是很重要的，并且编译器会试图优化算术表达式求值的性能。

我们对这部分内容的处理是基于一些核心的数学原理。从编码的基本定义开始，然后得出一些属性，例如可表示的数字的形式以及算术运算的属性。我们相信从这样一个抽象的观点来分析这些内容，对你来说是很重要的，因为作为计算机专业学生需要对计算机运算与更为人熟悉的整数和实数运算之间的关系有清晰的理解。

### 数的表示及进制转换

### 3.2.1数的表示

1. 十进制数的表示方法

十进制计数法的特点如下：

1. 使用10个数字符号0，1，2，……，9的不同组合来表示一个十进制数。这些符号称为数码，数码的个数称为基数，十进制的基数是10。
2. 一个数中，每个数码表示的值不仅仅取决于数码本身，还取决于其所处的位置(对每一个数码赋以不同的“权重”)。十进制中，每个数码上的权是10的某次幂。从个位，十位，百位开始，权重依次为100，101，和102，例如678=6×103＋7×10'+8×10°。每个数位上的数字所表示的量是该位数字和该数位上的权的乘积。
3. 逢十进一。任何一个十进制数可以用以下公式来表示：

上式(3.1)中，m表示小数位的位数，n表示整数位的位数，a表示第i位上的数码(可以是0-9中的任意一个)

1. 二进制数

式(3.1)可以推广到任意进制数。设其基数为R，则任意数N为

而对于二进制，R=2，ai为0或1，逢二进一。

例如，-1101.01012 = -13.312510，对于计算机存储和处理，负号和小数点是不方便的，因为只能用二进制数字(0和1)来表示数。如果只使用非负整数，那么其表示是直截了当的。一个8位的二进制数能表示从0~255的数、例如：

00000000 = 0

00000001 = 1

00101001 = 41

10000000 = 128

11111111 = 255

1. 八进制数

对于八进制，R=8，ai为0~7中的任何一个，逢八进一。

1. 十六进制数表示法

对于十六进制，R=16，ai为0~9，A，B，C，D，E，F中的任何一个，逢十六进一。



一个字节由8位组成。在二进制表示法中，它的值域是000000002~111111112。如果看成十进制数，它的值域就是010~25510.两种符号表示法对于描述位模式都不是非常方便。二进制表示法太冗长，而十六进制表示法与位模式的互相转化很麻烦、替代的方法是，以16为基数，或者叫十六进制数(hexadecimal)数，来表示位模式。十六进制(简写为hex)使用数字0~9以及字符A，B，C，D，E，F来表示16个可能的值。下表3.1展示了16个十六进制数字对应的十进制值和二进制值。用十六进制书写，一个字节(8位二进制数)的值域为0016~FF16。

表3.1

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 十六进制 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 十进制值 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 二进制值 | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 |
| 十六进制 | 8 | 9 | A | B | C | D | E | F |
| 十进制值 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 二进制值 | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |

在C语言中，以0x或0X开头的数字常量被认为是十六进制的值。字符A~F可以是大写也可以是小写。例如可以将FA1D37B16写作0Xfa1437b，或者，0xfa1d37b，甚至是可以大小写混合，比如0xFa1D37b。

### 3.2.2进制转换

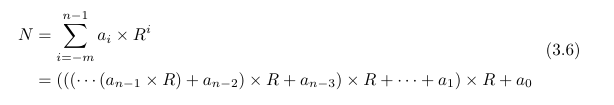
1. 任意进制数转化为十进制数

二进制，八进制和十六进制以至任意进制数转换为十进制数的方法很简单，可先将其按定义展开为多项式，再将系数及权均用十进制表示，按十进制进行乘法与加法运算，所得结果即为该数对应的十进制数。例如：

(101.01)2 = 1 × 22 + 0 × 21 + 1 × 20 + 0 ×2-1 + 1 × 1-2 = 5.25

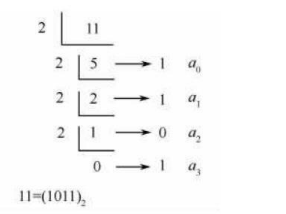
(AC7)16 = 10 × 162 + 12 × 161 + 7 × 160 = 2759

1. 十进制数转化为任意进制数

设N为任一十进制整数，如果要把它转换成n位R进制整数，则有

从式3.6可以看出，等式右边除了最后一项a0以外，其余各项都是包含基数R的因子，都能被R除尽。所以等式两边同除以基数R取其余数的方法得到ai。首先得到的是a0，如此一直进行下去，直到商等于0为止，就得到一系列余数a0，a1，…，an-1，他们正是要求的R进制数的各位。

十进制整数转换为任意进制整数的方法总结为：除以基数R取余数，先为低位后为高位。

例如将十进制数11转换为二进制数。

除了这种方法，如果对2的倍数比较熟悉的话，还可以直接看出，例如11 = 8+2+1，所以其二进制表示为1011。再比如592=512+64+16，其二进制数表示为100101000。

而十进制数转换为十六进制数需要使用乘法或者除法来处理一般情况。将一个十进制数字x转换为十六进制，可以反复地使用16除x，得到一个商q和一个余数r，也就是x = q × 16 + r。然后我们用十六进制数字表示的r作为最低为数字，并且对q反复进行这个过程得到剩下的数字。例如，考虑十进制314156的转换：

314156 = 19634 × 16 + 12 (C)

19634 = 1227 × 16 + 2 (2)

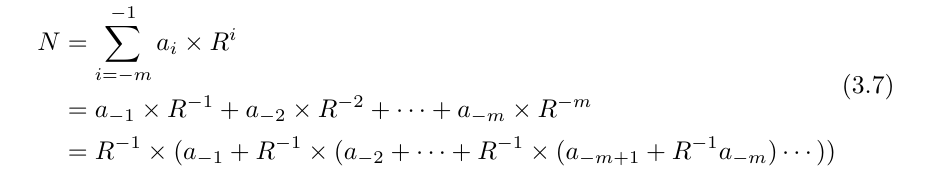
1227 = 76 × 16 + 11 (B)

76 = 4 × 16 + 12 (C)

4 = 0 × 16 + 4 (4)

通过上述步骤，我们能得出其十六进制表示为0x4CB2C。

对十进制小数转换成非十进制小数则使用如下方法。设N为任一十进制小数，若要把它转换为m位R进制小数，则有：



因此，可以将十进制小数不断乘以R，再取其乘积的整数作为ai，直到小数部分为零时停止。首先得到的是a-1，然后依次得到a-2，a-3，…，a-m。若乘积的小数部分始终不为0，说明相对应的R进制小数为不尽小数。这时可以乘到能满足计算机精度要求为止。综上所述，可以把十进制小数转换为相应R进制小数的方法总结为：乘以基数R取整数，现为高位后为低位。

例如，将0.625分别转换为二进制小数，其具体过程如下：

0.625 × 2 = 0.25 a-1 = 1

0.25 ×2 = 0.5 a-2 = 0

0.5×2 = 1 a-3 = 1

0.625 = (0.101)2

1. 二进制数与十六进制数之间的转换

因为24=16，即可用4位二进制数表示1位十六进制数，所以二进制和十六进制之间的转换比较简单直接。数字的转换可以参考上表3.1。一个简单的方法是：以小数点为界，向左(整数部分)每4位为一组，高位不足4位时补0；向右(小数部分)每4位为一组，低位不足4位时补0.然后分别用一个十六进制数表示每一组中的4位二进制数。

将十六进制数转换为二进制数的方法是：直接将每1位的十六进制数写成其对应的4位二进制数(参考表3.1)。

例如，假设一个十六进制数0x173A4C，可以通过展开每个十六进制数字，将它转换为二进制格式，如下所示：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 十六进制 | 1 | 7 | 3 | A | 4 | C |
| 二进制 | 0001 | 0111 | 0011 | 1010 | 0100 | 1100 |

这样就得到了该十六进制数的二进制表示：000101110011101001001100

反过来，如果给定了一个二进制数字1111001010110110110011，可以先把分为4位一组，每一组再转换成对应的十六进制数，再拼接起来。但是最左边的一组可能不足4位，这就需要在其前面补0，以满足4位。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 二进制 | (00)11 | 1100 | 1010 | 1101 | 1011 | 0011 |
| 十六进制 | 3 | C | A | D | B | 3 |

括号中为位数不足4位是补的0。

当值x是2的非负整数n次幂时，也就是x=2n，我们可以很容易地将x写成十六进制形式，只要记住x的二进制表示就是1后面跟n个0。十六进制数字0代表4个二进制0。所以，当n表示成i＋4j的形式，其中0≤i≤3，我们可以把x写成开头的十六进制数字为1(i=0)、2(i=1)、4(i=2)或者8(i=3)，后面跟随着j个十六进制的0。比如，x=2048=211，我们有n=11=3＋4·2，从而得到十六进制表示0x800。

而对于二进制与八进制的转换方式与十六进制的转换方式类似，只是在分组时每组只有3位。例如(725)8的转换，

|  |  |  |  |
| --- | --- | --- | --- |
| 八进制 | 7 | 2 | 5 |
| 二进制 | 111 | 010 | 101 |

那么得到其二进制表示为111010101，反过来也是一样，这里不在举例。

对于八进制与十六进制的转换可以借助二进制这个中间变量来转换，先将某数转换为二进制数，再转换成其他进制数。

### 数的码制

码制即编码体制，在数字电路中主要是指用二进制数来表示非二进制数字以及字符的编码方法和规则。日常生活中遇到的数，除了上述的无符号数(上面各进制数我们都是以正数为例)，还有带符号的二进制数，在计算机中通常用二进制数的最高位来表示数的符号。对于一个字节型(8位)二进制数来说，其最高位(最左边的位)表示符号位，剩下位为数值位。在带符号数中，规定用0表示正，用1表示负，而数值为表示该数的数值大小。

把一个数及其符号位在机器中的一组二进制数表示形式，称为“机器数”。机器数所表示的值称为该机器数的“真值”。例如：

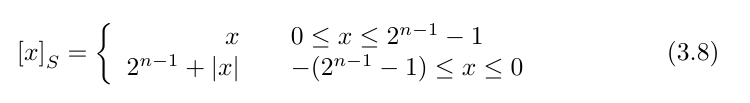
+32 = 0 0100000

-32 = 1 0100000

空格用来分割符号位与数值位。

### 3.3.1原码

原码(Sign-Magnitude)：最高有效位是符号位，用来确定剩下的位应该取负权还是正权。数x的原码记为[x]S，机器字长为n(二进制位数为n)，定义如下：



在原码的表示法中，最高位为符号位(正数为0，负数为1)，其余数字位表示数的绝对值，例如：

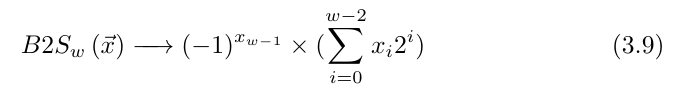
x1 = +1010101， 则[x1] S = 01010101，最高位的0表示它是正数

x2 = -1011101，则[x2] S = 11011101，最高位的1表示它是负数

可以看出，8位二进制原码表示数的范围为-127 ~ +127，16位二进制原码表示数的范围为-32767 ~ +32767。值得注意的是，0的原码表示有两种方式：

[+0] S = 00000000 或 [-0] S = 10000000

另外，式(3.9)给出了一个二进制原码序列到十进制整数的映射公式。

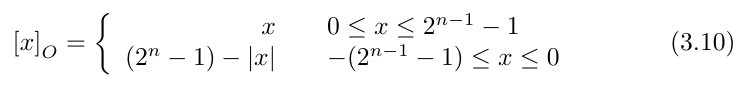


其中B2SW(Binary to Unsigned)是一个函数，下标w为二进制原码序列的位数，该函数描述了将一个长度为w的二进制序列映射到一个整数。

总的来说，原码表示法简单直观，且与真值的转换很方便，但不便于在计算机中进行加减运算。如进行两数相加，必须先判断两个数的符号是否相同。如果相同，则进行加法运算，否则进行减法运算。如进行两数相减﹐必须比较两数的绝对值大小，再由大数减小数﹐结果的符号要和绝对值大的数的符号一致。按上述运算方法设计的算术运算电路很复杂。为此引入了反码和补码表示法，它们可以使正、负数的加法和减法运算简化为单一相加运算。

### 3.3.2反码

反码(One’s Complement)通常用来作为由原码求补码或者由补码求原码的中间过度。数x的原码记为[x]O，机器字长为n(二进制位数为n)。反码的定义如下：



正数的反码与其原码相同。例如，当机器字长为8位时：

[+35]O = [+35]S = 00100011

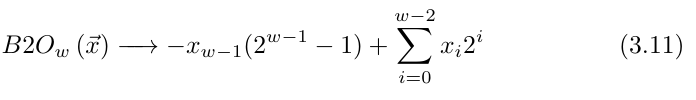
负数的反码是在原码基础上，最高位符号位不变(仍然为1)，数值位按位取反。例如当机器字长也为8位时：

[-0]O = (28-1)-0 = 1111 1111

[-35] O = (28-1)-35 = 1101 1100，也可由-35的原码为1010 0011按位取反得到。

由此可见由于最高位会占据一位符号位，所以对于8位二进制反码来说，其表示范围为-127 ~ +127。对于16位二进制反码，其表示范围为-32767 ~ +32767。值得注意的是，0的反码有两种表示方式，分别是：[+0] O = 0000 0000，[-0]O  = 1111 1111。

同样，式(3.11)给出了一个二进制反码序列到十进制整数的映射公式。



### 3.3.4补码

补码源于“补数”的概念。而在日常生活中，我们经常会遇到“补数”。例如，时钟指向6点，欲使它指向3点，既可以按照顺时针方向将分针转9圈，又可以按照逆时针方向将分针转3圈，而他们最终的结果是一致的。假设顺时针方向为正，逆时针方向为负，则有

|  |  |
| --- | --- |
| 6 | 6 |
| -3 | +9 |
| 3 | 15 |

由于时钟的针转一圈能指示12个小时，这“12”在时钟里是不被显示而自动丢失的，即15 – 12 = 3，故15点和3点均显示3点。这样-3和+9对时钟而言其作用是一致的。在数学上称12为模，写作mod 12，而+9是-3以12为模的补数，记作

-3 ≡ +9 (mod 12)

上式的符号代表“同余”，也即+9模12与-3模12的结果相同。

对于负数来说，例如-3，可以看作-3 = 12 × (-1) + 9。

或者说，对于模12而言，-3和+9是互为补数的。同理有

-4 ≡ +8 (mod 12)

-5 ≡ +7 (mod 12)

即对模12而言，+8和+7分别是-4和-5的补数。可见，只要确定了模，就可以找到一个与负数等价的正数(该正数即为负数的补数)来代替此负数，这样就可以把减法运算用加法运算所实现。例如：

设A = 9，B = 5，求A – B (mod 12)

解：

A - B = 9 - 5 = 4 (做减法)

对于模12而言，-5可以用其补数+7代替，即

-5 ≡ +7 (mod 12)

所以 A – B = 9 + 7 = 16 (做加法)

对模12而言，12会自动丢失，所以16等价于4，即有

+4 ≡ +16 (mod 12)

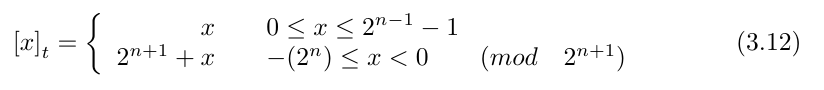
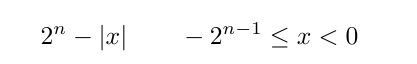
进一步分析发现我们可以得出，3点，15点，27点等等在时钟上看见的都是3点，即

+3 ≡ +15 ≡ +27 (mod 12)

这就说明正数相对于“模”的补数就是该正数本身。上述补数的概念可以用到任意“模”上。由此可以得到如下结论：

1. 一个负数可用它的正补数来代替，而这个正补数可以用模加上负数本身求得。
2. 一个正数和一个负数互为补数时，它们绝对值的和即为模数。
3. 正数的补数即该正数本身。

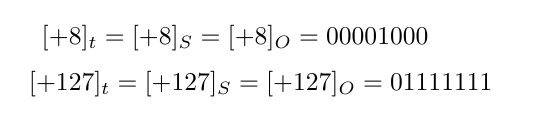
将补数的概念用到计算机中，便出现了补码(two’s-complement)这种机器数。补码的定义如下：



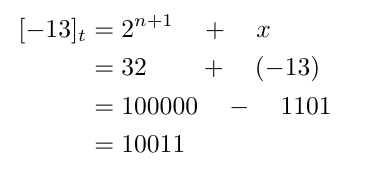
其中式(3.12)第二行也等价于上面的表达式。

上式(3.12)中，x为真值，n为整数位数。

正数的补码与其原码和反码相同，例如有

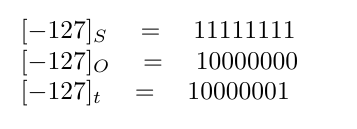


对于负数而言，按照上式定义计算即可，例如当x = -13时,



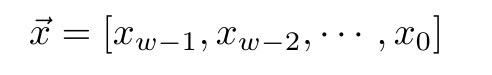
注意：这里位数只有五位，如果用8位二进制来表示，完整的是11110011。

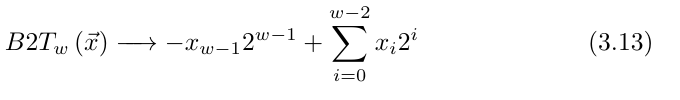
负数的补码除了用公式计算得到，还有一种简便方法。负数的补码是在原码的基础上，符号位不变(仍为1)，数值位按位取反，末位加1；或者在其反码的基础上加1。例如，当机器字长同样为8位时，有



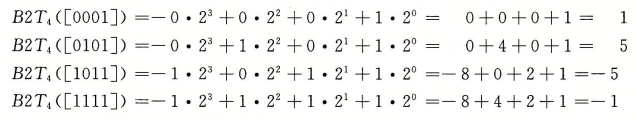
接下来让我们考虑以下当位数为w时，补码所能表示的数的范围。在此之前，我们同样补充补码对真值的映射公式如下：

对于一段二进制序列(例如1110)，我们用向量的形式来表示，对如下向量，



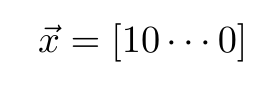


最高有效位xw-1也称为符号位，它的权重为-2w-1，是无符号表示中权重的负数。补码也同原码一样，符号位为1时代表负数，而当设置为0时代表正。这里我们来看一下示例，展示的是在下面几种情况下，B2T函数给出的从二进制向量到整数的映射。

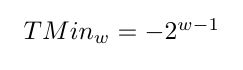


有了上面的公式(3.13)，我们正式开始探究在w位的补码长度内，其能表示的最小/最大值是多少。

首先明确，要找的最小值一定是负数。那么该向量的最高位一定是1，其次根据公式右边可以看出，是一个负数加上一个正数和，要想得到最小值，那么该正数和只能为0。所以它能表示的最小值的向量形式如下：



也就是说设置这个位为负权，其他位全部置0，其对应的整数值如下：



T代表补码，Min代表最小值，下标w代表w位下的情况。这个结论也跟公式(3.13)相符合。

我们探讨了最小值，接下来我们来探讨最大值。同样的，最大值一定是正数，那么补码的最高位一定为0，根据公式(3.13)，要想得到最大值，说明除了最高位剩下的每一位都必须有数，也就是余下的每一位上都是1。所以，最大值的位向量表示如下：



清楚具有负权的位，而设置其他所有的位。同样，其对应的整数值如下：



我们以长度w为4举例，我们有

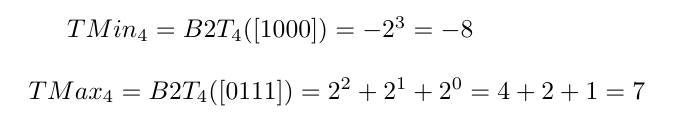


表3.2展示了针对不同字长，几个重要数字的二进制向量和对应的整数数值。前三个给出的是可表示的整数的范围，用UMax(U代表无符号unsigned) 、TMin(T代表补码)和TMax.来表示。在后面的讨论中，我们还会经常引用到这三个特殊的值。如果可以从上下文中推断出w，或者w不是讨论的主要内容时，我们会省略下标w，直接引用UMax 、TMin 和 TMax 。这里读者可以自行推导以下，加深对补码的理解。

表3.3

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 数 | 字长 | | | |
| 8 | 16 | 32 | 64 |
| UMax | 0xFF  255 | 0xFFFF  65 535 | 0xFFFFFFFF  4 294 967 295 | 0xFFFFFFFFFFFFFFFF  18 446 744 073 709 551 615 |
| TMin | 0x80  -128 | 0x8000  -32 768 | 0x80000000  -2 147 483 648 | 0x8000000000000000  -9 223 372 036 854 775 808 |
| TMax | 0x7F  127 | 0x7FFF  32 767 | 0x7FFFFFFF  2 147 483 647 | 0x7FFFFFFFFFFFFFFF  9 223 372 036 854 775 807 |
| -1  0 | 0xFF  0x00 | 0xFFFF  0x0000 | 0xFFFFFFFF  0x00000000 | 0xFFFFFFFFFFFFFFFF  0x000000000000000 |

关于这些数字，有几点值得注意。

第一，从表3.3可以看到，补码的范围是不对称的:|TMin| =| TMax|+1，也就是说，TMin没有与之对应的正数。这就导致了补码运算的某些特殊的属性，并且容易在程序中造成错误。之所以会有这样的不对称性，是因为一半的二进制向量(符号位为1的数)表示负数，而另一半(符号位设置为0的数)表示非负数。因为0是非负数，也就意味着能表示的整数比负数少一个。

第二，最大的无符号数值刚好比补码的最大值的两倍大一点: UMax =2TMax +1。补码表示中所有表示负数的位模式在无符号表示中都变成了正数。表3.3也给出了-1和0的表示。注意-1和UMax有同样的位表示(一个全1的串)。数值0在两种表示方式中都是全0的串。

C语言标准并没有要求要用补码形式来表示有符号整数，但是几乎所有的机器都是这么做的。C库中的文件<limits.h>定义了一组常量，来限定编译器运行的这台机器的不同整型数据类型的取值范围。比如，它定义了常量INT\_MAX、INT\_MIN和UINT\_MAx，它们描述了有符号和无符号整数的范围。对于一个补码的机器，数据类型int有w位，这些常量就对应于TMax、TMin.和UMax的值。

### 定点数和浮点数

### 3.4.1定点数

定点表示法，是指小数点在书中的位置是固定的，一般有两种格式。从原理上讲，小数点的位置固定在哪一位都是可以的,但通常将数据表示成纯小数或纯整数形式。如图3.1所示。

对于纯小数，规定小数点固定在最高数值位之前，机器中能表示的所有数都是小数。n位数值部分所能表示的数N的范围(原码表示，下同)为:





图3.1

当小数点位于数符和第一数值位之间时，机器内的数为纯小数。当小数点位于数值位之后时，机器内的数为纯整数。采用定点数的机器称为定点机。数值部分的位数n决定了定点机中数的表示范围。若机器数采用原码，小数定点机中数的表示范围是–(1 - 2n) ~ (1 - 2-n)，整数定点机中数的表示范围是-(2n-1) ~ (2n - 1)。

在定点机中,由于小数点的位置固定不变,故当机器处理的数不是纯小数或纯整数时.必须乘上一个比例因子,否则会产生“溢出”。

### 3.4.2浮点数概况

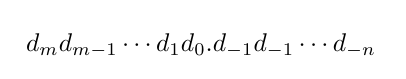
浮点表示对形如V= x × 2y的有理数进行编码。它对执行涉及非常大的数字(|V|>>0)、非常接近于0(|V|<<1)的数字，以及更普遍地作为实数运算的近似值的计算是非常有用的。

直到20世纪80年代，每个计算机制造商都设计了自己的表示浮点数的规则，以及对浮点数执行运算的细节。但是它们不会太多关注运算的精确性，而是更看重实现的速度和简便性。

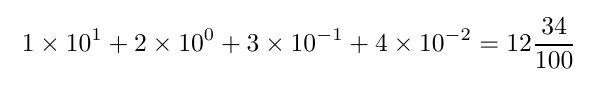
大约在1985年，这些情况随着IEEE标准754的推出而改变了。这是一个仔细制订的表示浮点数及其运算的标准。这项工作是从1976年开始由Intel赞助的，与8087的设计同时进行，8087是一种为8086处理器提供浮点支持的芯片。他们请William Kahan(加州大学伯克利分校的一位教授)作为顾问，帮助设计未来处理器浮点标准。他们支持Kahan加入一个IEEE资助的制订工业标准的委员会。这个委员会最终采纳的标准非常接近于Kahan为Intel设计的标准。目前，实际上所有的计算机都支持这个后来被称为IEEE浮点的标准。这大大提高了科学应用程序在不同机器上的可移植性。

在本节中，我们将看到IEEE浮点格式中数字是如何表示的。我们还将探讨舍入(rounding)的问题，即当一个数字不能被准确地表示为这种格式时，就必须向上调整或者向下调整。然后，我们将探讨加法、乘法和关系运算符的数学属性。

### 3.4.3二进制小数

理解浮点数的第一步是考虑含有小数值的二进制数字。我们首先来看更熟悉的十进制表示法。十进制表示法使用如下形式的表示：

其中每个十进制数di的取值范围是0~9。上述表达式的数值d定义如下：

数字的权重定义与十进制小数的小数点符号相关，这意味着小数点左边的数字的权是10的正幂，得到整数值。而小数点右边的数字的权是10的负幂，得到小数值。例如，对于十进制小数12.34来说，其表示的含义如下：

类似的，我们可以考虑一个形如如下二进制小数的表示法，如下图3.2所示。

其中每个二进制数字称为位，bi的取值范围为0或1，用这种方法表示的数b定义如下：

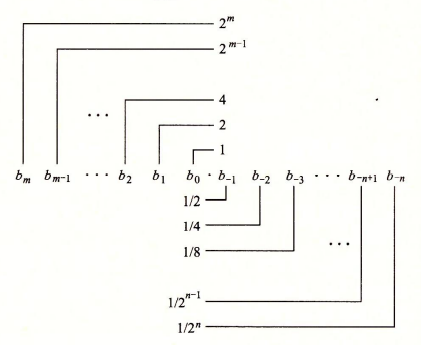
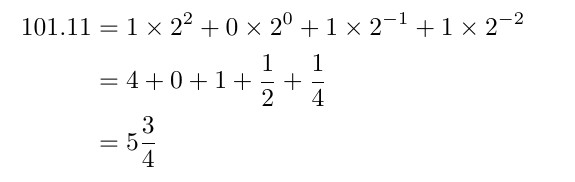


图3.2

现在的小数点符号变成了二进制的小数点。小数点左边的位权变为了2的正幂，小数点右边的位权变成了2的负幂。同样的，例如二进制小数101.11表示的十进制数字如下：

从式(3.15)可以看出，二进制小数点向左移动一位相当于这个数被2除。例如，101.112表示数5.75，而10.1ll表示数2.875。类似，二进制小数点向右移动一位相当于将该数乘2。例如1011.1表示数11.5。

注意，形如0.11…1的二进制小数表示的刚好是小于1的数。例如，0.111111表示的是我们将用简单的表达法1.0-ε来表示这样的数值。

假定我们仅仅考虑有限长度的编码，那么十进制表示法不能准确地表达像

这样的数。



类似，小数的二进制表示法只能表示那些能够被写成x × 2y的数。其他的值只能够被近似地表示。例如，十进制小数0.20可以精确表示。不过，我们并不能把它准确地表示为一个二进制小数，我们只能近似地表示它，增加二进制表示的长度可以提高表示的精度。

### 3.4.4 IEEE浮点表示

前一节中谈到的定点表示法不能很有效地表示非常大的数字。例如，表达式5×2100是用101后面跟随100个零的位模式来表示。相反，我们希望通过给定x和y 的值，来表示形如x×2y的数。

IEEE(Institute of Electrical and Electronics Engineers,电气工程师协会)在1985年制定的IEEE 754(IEEE Standard for Binary Floating-Point Arithmetic，ANSI/IEEE Std 754—1985)二进制浮点运算规范，是浮点运算部件事实上的工业标准。

IEEE浮点标准用如下的公式来表示一个数：

* 符号(sign) s决定这数是负数(s=1)还是正数(s=0)，而对于数值0的符号位解释作为特殊情况处理。
* 尾数(significand) M是一个二进制小数，它的范围是1~2-ε，或者是0~1-ε。
* 阶码(exponent) E的作用是对浮点数加权，这个权重是2的E次幂(可能是负数)。将浮点数的位表示划分为三个字段，分别对这些值进行编码:
* 一个单独的符号位s直接编码符号s 。
* k位的阶码字段exp=ek-1…e1e0编码阶码E。
* n位小数字段frac=fn-1…f1f0。编码尾数M，但是编码出来的值也依赖于阶码字段的值是否等于0。

图3.2 标准浮点格式

图3.2给出了将这三个字段装进字中两种最常见的格式。在单精度浮点格式(C语言中的float)中，s、exp和frac字段分别为1位、k=8位和n=23位，得到一个32位的表示。在双精度浮点格式(C语言中的double)中，s、exp和frac字段分别为1位、k=11位和n=52位，得到一个64位的表示。

给定位表示，根据exp的值，被编码的值可以分成三种不同的情况(最后一种情况有两个变种)。图3.3说明了对单精度格式的情况。

图3.3 单精度浮点数值的分类

1. 规格化的值

这是最普遍的情况。当exp 的位既不全为0(数值0)，也不全为1(单精度数值为255，双精度数值为2047)时，都属于这类情况。在这种情况中，阶码字段被解释为以偏置(biased)形式表示的有符号整数。也就是说，阶码的值是E=e-Bias，其中e是无符号数，其位表示为ek-1…e1e0，而Bias是一个等于2k-1-1(单精度是127，双精度是1023)的偏置值。由此产生指数的取值范围，对于单精度是-126~＋127，而对于双精度是-1022～+1023。

小数字段frac被解释为描述小数值f，其中0≤f<l，其二进制表示为0,fn-1…f1f0，也就是二进制小数点在最高有效位的左边。尾数定义为M=1+f。有时，这种方式也叫做隐含的以1开头的(implied leading 1)表示，因为我们可以把M看成一个二进制表达式为1.fn-1 fn-2…f0的数字。既然我们总是能够调整阶码E，使得尾数M在范围1≤M<2之中(假设没有溢出)，那么这种表示方法是一种轻松获得一个额外精度位的技巧。既然第一位总是等于1，那么我们就不需要显式地表示它。

1. 非规格化的值

当阶码域为全0时，所表示的数是非规格化形式。在这种情况下，阶码值是E=1-Bias，而尾数的值是M=f，也就是小数字段的值，不包含隐含的开头的1。使阶码值为1-Bias而不是简单的-Bias，这种方式提供了一种从非规格化值平滑地转换到规格化值地方法。

非规格化数有两个用途。首先，它们提供了一种表示数值0的方法，因为使用规格化数，我们必须总是使M≥1，因此我们就不能表示0。实际上，＋0.0的浮点表示的位为全0：符号位是0，阶码字段全为0(表明是一个非规格化值)，而小数域也全为0，这就得到M=f=0。但是，当符号位为1，而其他域全为0时，我们得到值-0.0。根据IEEE的浮点格式，值+0.0和-0.0在某些方面被认为是不同的，而在其他方面是相同的。

非规格化数的另外一个功能是表示那些非常接近于0.0的数。它们提供了一种属性，称为逐渐溢出(gradual underflow)，其中，可能的数值分布均匀地接近于0.0。

1. 特殊值

首先在图3.3中，NaN(Not a Number)代表未定义或不可表示地值。最后―类数值是当指阶码全为1的时候出现的。当小数域全为0时，得到的值表示无穷，当s=0时是＋oo，或者当s=1时是-oo。当我们把两个非常大的数相乘，或者除以零时，无穷能够表示溢出的结果。当小数域为非零时，结果值被称为“NaN”。一些运算的结果不能是实数或无穷，就会返回这样的NaN值，比如当计算或，时。在某些应用中，表示未初始化的数据时，它们也很有用处。

### 3.4.5 数字示例

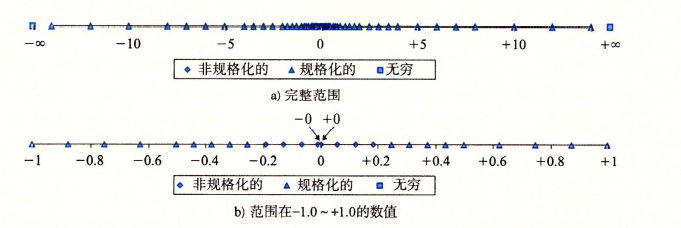
图3.4展示了一组数值，它们可以用假定的6位二进制来表示，有k=3的阶码位和n=2的尾数位。偏置量是23-1-1=3。图中的a部分显示了所有可表示的值(除了NaN)。两个无穷值在两个末端。最大数量值的规格化数是±14。非规格化数聚集在0的附近。图的b部分中，我们只展示了介于-1.0和＋1.0之间的数值，这样就能够看得更加清楚了。两个零是特殊的非规格化数。可以明显发现那些可表示的数并不是均匀分布的——越靠近原点处它们越稠密。

图3.4 6位浮点格式表示的值(k=3的阶码位和n=2的尾数位，偏执量是3)

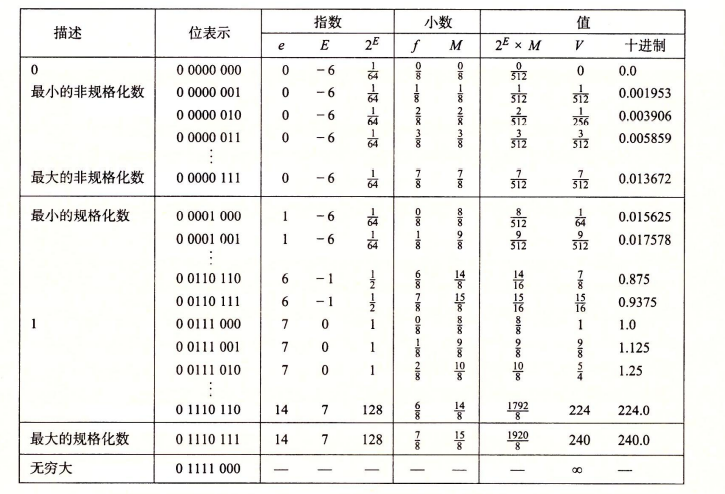
图3.5展示了假定的8位浮点格式的示例，其中有k=4的阶码位和n=3的小数位。偏置量是24-1-1=7。图被分成了三个区域，来描述三类数字。不同的列给出了阶码字段是如何编码阶码E的，小数字段是如何编码尾数M的，以及它们一起是如何形成要表示的值V=2E×M的。从0自身开始，最靠近0的是非规格化数。这种格式的非规格化数的E=1-7=-6，得到权2E=。小数f的值的范围是0，，…，，从而得到数V的范围是0～ × =

图3.5 8位浮点格式的非负值示例

这种形式的最小规格化数同样有E=1-7=-6，并且小数取值范围也为0，，…，，然而，尾数范围在1+0=1和1+=之间，得出数V在范围=和之间。我们可以观察到最大非规格化数和最小规格化数之间的转变。这种平滑性归功于我们对非规格化数的E的定义。通过将E定义为1-Bias而不是-Bias，我们可以补偿非规格化的尾数没有隐含的开头的1。

当增大阶码时，我们成功地得到更大地规格化值，通过1.0后得到最大规格化数。这个数具有阶码E=7，得到一个权为2E = 128。小数等于得到尾数M=。因此，数值是V=240.超出这个值就会溢出到＋oo。

这种表示具有一个有趣的属性，假如我们将图3.5中的值的位表达式解释为无符号整数，它们就是按升序排列的，就像它们表示的浮点数一样。这不是偶然的——IEEE格式如此设计就是为了浮点数能够使用整数排序函数来进行排序。当处理负数时，有一个小的难点，因为它们有开头的1，并且它们是按照降序出现的，但是不需要浮点运算来进行比较也能解决这个问题。

### 十进制数的编码

如前所述，在计算机中是使用二进制代码工作的。但是由于长期的习惯，在日常生活中，人们最熟悉的数制还是十进制。为了解决这一矛盾，提出了一种比较适合于十进制系统的二进制代码的特殊形式，即将1位十进制的0~9这10个数字分别用4位二进制码的组合来代表，在此基础上，可按位对任意十进制数进行编码。这就是二进制编码的十进制数，简称BCD码(Binary-Coded Decimal)。BCD码可分为两类：有权码和无权码。为什么使用BCD码呢，因为这种编码技巧最常用于会计系统的设计里，因为会计制度经常需要对很长的数字串作准确的计算。相对于一般的浮点式记数法，采用BCD码，既可保存数值的精确度，又可免去使计算机作浮点运算时所耗费的时间。此外，对于其他需要高精确度的计算，BCD编码亦很常用。

### 3.5.1有权码

8421 BCD码是最基本和最常用的BCD码，它和四位自然二进制码相似，各位的权值为8、4、2、1，故称为有权BCD码。和四位自然二进制码不同的是，它只选用了四位二进制码中前10组代码，即用0000~1001分别代表它所对应的十进制数，余下的六组代码不用。

5421 BCD码和2421 BCD码为有权BCD码，它们从高位到低位的权值分别为5、4、2、1和2、4、2、1。这两种有权BCD码中，有的十进制数码存在两种加权方法，例如，5421 BCD码中的数码5，既可以用1000表示，也可以用0101表示；2421 BCD码中的数码6，既可以用1100表示， 也可以用0110表示。这说明5421 BCD码和2421 BCD码的编码方案都不是惟一的，

### 3.5.2无权码

余三码（余3码）是由8421BCD码加上0011形成的一种无权码 \*\*，由于它的每个字符编码比相应的8421码多3，故称为余三码。BCD码的一种。余3码的特点：当两个十进制数的和是10时，相应的二进制编码正好是16，于是可自动产生进位信号,而不需修正。0和9， 1和8，……5和4的余3码互为反码,这在求对于10的补码很方便。

余三码是一种对9的自补代码，因而可给运算带来方便。其次，在将两个余三码表示的十进制数相加时，能正确产生进位信号，但对“和”必须修正。修正的方法是：如果有进位，则结果加3；如果无进位，则结果减3。

在一组数的编码中，若任意两个相邻的代码只有一位二进制数不同，则称这种编码为格雷码（Gray Code），另外由于最大数与最小数之间也仅一位数不同，即“首尾相连”，因此又称循环码或反射码。

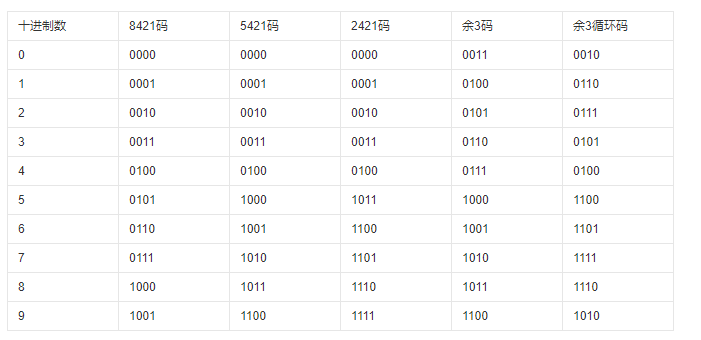
下图3.6展示了编码十进制数的常用BCD码。

图3.6 常用BCD码

### 数的存储

大多数计算机使用8位的块，或者字节(byte)，作为最小的可寻址的内存单位，而不是访问内存中单独的位。机器级程序将内存视为一个非常大的字节数组，称为虚拟内存(virtual memory)。内存的每个字节都由一个唯一的数字来标识，称为它的地址(address)，所有可能地址的集合就称为虚拟地址空间( virtual address space)。顾名思义，这个虚拟地址空间只是一个展现给机器级程序的概念性映像。实际的实现是将动态随机访问存储器(DRAM)、闪存、磁盘存储器、特殊硬件和操作系统软件结合起来，为程序提供一个看上去统一的字节数组。这一小节我们主要介绍在计算机中数值型数据和非数值型数据是以何种方式存储的。

每台计算机都有一个字长(word size)，指明指针数据的标称大小(nominal size)。因为虚拟地址是以这样的一个字来编码的，所以字长决定的最重要的系统参数就是虚拟地址空间的最大大小。也就是说，对于一个字长为w位的机器而言，虚拟地址的范围为0～2w—1，程序最多访问2w个字节。

最近这些年，出现了大规模的从 32位字长机器到64位字长机器的迁移。这种情况首先出现在为大型科学和数据库应用设计的高端机器上，之后是台式机和笔记本电脑，最近则出现在智能手机的处理器上。32位字长限制虚拟地址空间为4千兆字节(写作4GB)，也就是说，刚刚超过4×109字节。扩展到64位字长使得虚拟地址空间为16EB，大约是1.84×1019字节。

大多数64位机器也可以运行为32位机器编译的程序，这是一种向后兼容。

### 3.6.1数字存储

1. 整形

C语言支持多种整型数据类型来表示有限范围的整数。这些类型如表3.4和表3.5所示，其中还给出了“典型”32位和64位机器的取值范围。每种类型都能用关键字来指定大小，这些关键字包括short(至少16位)、int(至少与short一样长，如今一般用32位存储)、long(至少32位，且至少与int一样长)、long long(至少64位，且至少与long一样长)，同时还可以指示被表示的数字是非负数(声明为unsigned)，或者可能是负数(默认)。如表3.6所示，为这些不同的大小分配的字节数根据程序编译为32位还是64位而有所不同。根据字节分配，不同的大小所能表示的值的范围是不同的。这里给出来的唯一一个与机器相关的取值范围是大小指示符long的。大多数64位机器使用8个字节的表示，比 32位机器上使用的4个字节的表示的取值范围大很多。

表3.4 32位程序上C语言整形数据类型的典型取值范围

|  |  |  |
| --- | --- | --- |
| C数据类型 | 最小值 | 最大值 |
| char | -128 | 127 |
| unsigned char | 0 | 255 |
| short | -32 768 | 32 767 |
| unsigned short | 0 | 65 535 |
| int | -2 147 483 648 | 2 147 483 647 |
| unsigned | 0 | 4 294 967 295 |
| long | -2 147 483 648 | 2 147 483 647 |
| unsigned long | 0 | 4 294 967 295 |
| int32\_t | -2 147 483 648 | 2 147 483 647 |
| uint32\_t | 0 | 4 294 967 295 |
| int64\_t | -9 223 372 036 854 775 808 | 9 223 372 036 854 775 808 |
| uint64\_t | 0 | 18 446 744 073 709 551 615 |

表3.5 64位程序上C语言整形数据类型的典型取值范围

|  |  |  |
| --- | --- | --- |
| C数据类型 | 最小值 | 最大值 |
| char | -128 | 127 |
| unsigned char | 0 | 255 |
| short | -32 768 | 32 767 |
| unsigned short | 0 | 65 535 |
| int | -2 147 483 648 | 2 147 483 647 |
| unsigned | 0 | 4 294 967 295 |
| long | -9 223 372 036 854 775 808 | 9 223 372 036 854 775 808 |
| unsigned long | 0 | 18 446 744 073 709 551 615 |
| int32\_t | -2 147 483 648 | 2 147 483 647 |
| uint32\_t | 0 | 4 294 967 295 |
| int64\_t | -9 223 372 036 854 775 808 | 9 223 372 036 854 775 808 |
| uint64\_t | 0 | 18 446 744 073 709 551 615 |

表3.5 基本C数据类型的典型大小

分配的字节数受程序是如何编译的影响而变化

|  |  |  |  |
| --- | --- | --- | --- |
| C声明 | | 字节数 | |
| 有符号 | 无符号 | 32位 | 64位 |
| char | unsigned char | 1 | 1 |
| short | unsigned short | 2 | 2 |
| int | unsigned | 4 | 4 |
| long | unsigned long | 4 | 8 |
| int32\_t | uint32\_t | 4 | 4 |
| int64\_t | uint64\_t | 8 | 8 |
| char \* |  | 4 | 8 |
| float |  | 4 | 4 |
| double |  | 8 | 8 |

表3.4和表3.5中一个很值得注意的特点是取值范围不是对称的——负数的范围比整数的范围大1。这在之前我们对数的码制那一节中有过探讨。

C语言标准定义了每种数据类型必须能够表示的最小的取值范围。如表3.7所示，它们的取值范围与表3.4和表3.5所示的典型实现一样或者小一些。特别地，除了固定大小的数据类型是例外，我们看到它们只要求正数和负数的取值范围是对称的。此外，数据类型int可以用2个字节的数字来实现，而这几乎回退到了16位机器的时代。还可以看到，long的大小可以用4个字节的数字来实现，对32位程序来说这是很典型的。固定大小的数据类型保证数值的范围与表3.5给出的典型数值一致，包括负数与正数的不对称性。

表3.7 C语言的整形数据类型的保证的取值范围。C语言标准要求

这些数据类型至少具有这样的取值范围

|  |  |  |
| --- | --- | --- |
| C数据类型 | 最小值 | 最大值 |
| char | -128 | 127 |
| unsigned char | 0 | 255 |
| short | -32 768 | 32 767 |
| unsigned short | 0 | 65 535 |
| int | -2 147 483 648 | 2 147 483 647 |
| unsigned | 0 | 4 294 967 295 |
| long | -2 147 483 648 | 2 147 483 647 |
| unsigned long | 0 | 4 294 967 295 |
| int32\_t | -2 147 483 648 | 2 147 483 647 |
| uint32\_t | 0 | 4 294 967 295 |
| int64\_t | -9 223 372 036 854 775 808 | 9 223 372 036 854 775 808 |
| uint64\_t | 0 | 18 446 744 073 709 551 615 |

1. 浮点型

各种整数类型对大多数软件开发项目而言够用了。然而，面向金融和数学的程序经常使用浮点数。C语言中的浮点类型有float、double和long double类型。它们与FORTRAN和 Pascal中的real类型一致。前面提到过，浮点类型能表示包括小数在内更大范围的数。浮点数的表示类似于科学记数法(即用小数乘以10的幂来表示数字)。该记数系统常用于表示非常大或非常小的数。表3.8列出了一些示例。

表3.8 记数法示例

|  |  |  |
| --- | --- | --- |
| 数字 | 科学计数法 | 指数计数法 |
| 1 000 000 000 | 1.0×109 | 1.0e9 |
| 123 000 | 1.23×105 | 1.23e5 |
| 322.56 | 3.2256×102 | 3.2256e2 |
| 0.000056 | 5.6×10-5 | 5.6e-5 |

C标准规定，float类型必须至少能表示6位有效数字，且取值范围至少是103~10\*。前一项规定指float类型必须至少精确表示小数点后的6位有效数字，如33.333333。后一项规定用于方便地表示诸如太阳质量（2.0e30千克)、一个质子的电荷量（1.6e-19库仑）或国家债务之类的数字。通常，系统储存一个浮点数要占用32位。其中8位用于表示指数的值和符号，剩下24位用于表示非指数部分（也叫作尾数或有效数）及其符号。

C语言提供的另一种浮点类型是double(意为双精度)。double类型和float类型的最小取值范围相同，但至少必须能表示10位有效数字。一般情况下，double占用64位而不是32位。一些系统将多出的32位全部用来表示非指数部分，这不仅增加了有效数字的位数（即提高了精度)，而且还减少了舍入误差。另一些系统把其中的一些位分配给指数部分，以容纳更大的指数，从而增加了可表示数的范围。无论哪种方法，double类型的值至少有13位有效数字，超过了标准的最低位数规定。

C语言的第3种浮点类型是long double，以满足比double类型更高的精度要求。不过，C只保证long double类型至少与double类型的精度相同。

### 3.6.2文本存储

计算机中的文本主要是以ASCII码，每个国家各自的编码以及Unicode这三种型式存储。

1. ASCII

ASCII ((American Standard Code for Information Interchange): 美国信息交换标准代码）是基于拉丁字母的一套电脑编码系统，主要用于显示现代英语和其他西欧语言。它是最通用的信息交换标准，并等同于国际标准ISO/IEC 646。ASCII第一次以规范标准的类型发表是在1967年，最后一次更新则是在1986年，到目前为止共定义了128个字符。

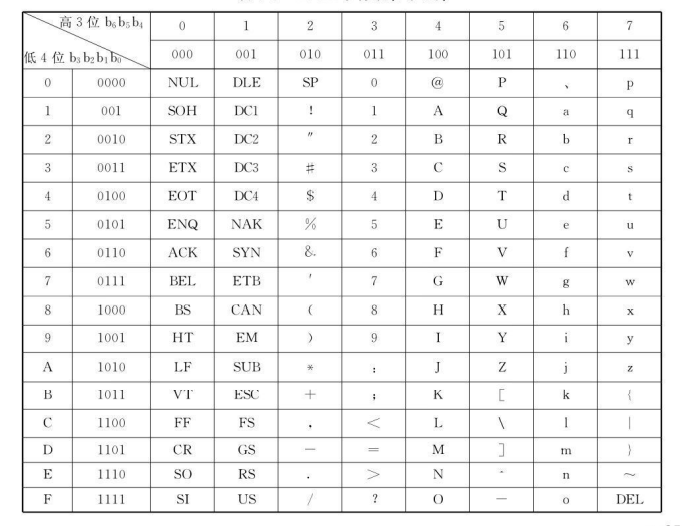
ASCII 由电报码发展而来。第一版标准发布于1963年，1967年经历了一次主要修订，最后一次更新则是在1986年，至今为止共定义了128个字符。其中33个字符无法显示（一些终端提供了扩展，使得这些字符可显示为诸如笑脸、扑克牌花式等8-bit符号），且这33个字符多数都已是陈废的控制字符。控制字符的用途主要是用来操控已经处理过的文字。在33个字符之外的是95个可显示的字符。用键盘敲下空白键所产生的空白字符也算1个可显示字符（显示为空白）。

ASCII 码使用指定的7 位或8 位二进制数组合来表示128 或256 种可能的字符。标准ASCII 码也叫基础ASCII码，使用7 位二进制数（剩下的1位二进制为0）来表示所有的大写和小写字母，数字0 到9、标点符号，以及在美式英语中使用的特殊控制字符。

同时还要注意，在标准ASCII中，其最高位(b7)用作奇偶校验位。所谓奇偶校验，是指在代码传送过程中用来检验是否出现错误的一种方法，一般分奇校验和偶校验两种。奇校验规定：正确的代码一个字节中1的个数必须是奇数，若非奇数，则在最高位b7添1；偶校验规定：正确的代码一个字节中1的个数必须是偶数，若非偶数，则在最高位b7添1。

后128个称为扩展ASCII码。许多基于x86的系统都支持使用扩展（或“高”）ASCII。扩展ASCII 码允许将每个字符的第8 位用于确定附加的128 个特殊符号字符、外来语字母和图形符号。

ASCII码的具体展示如下：



1. 汉字编码字符集

汉字编码国家标准，分为双字节部分和四字节部分；双字节部分和GBK基本完全相同。四字节部分，比GBK多了6582个汉字(27484-20902)。

现有的汉字编码主要有以下四种。

GB2312-1980(信息交换用汉字编码字符集基本集)

GBK-1995(汉字内码扩展规范)

GB1300.1-1993(信息技术通用多八位编码字符集(UCS)第一部分:体系结构与基本多文种平面(idt ISOIEC 10646.1-1993))

GB18030-2000(信息交换用汉字编码字符集基本集的扩充)

1. Unicode

统一码(Unicode)，也叫万国码、单一码，是计算机科学领域里的一项业界标准，包括字符集、编码方案等。Unicode是为了解决传统的字符编码方案的局限而产生的，它为每种语言中的每个字符设定了统一并且唯一的二进制编码，以满足跨语言、跨平台进行文本转换、处理的要求。

如果把各种文字编码形容为各地的方言，那么Unicode就是世界各国合作开发的一种语言。在这种语言环境下，不会再有语言的编码冲突，在同屏下，可以显示任何语言的内容，这就是Unicode的最大好处。就是将世界上所有的文字用2个字节统一进行编码。那样，像这样统一编码，2个字节就已经足够容纳世界上所有的语言的大部分文字了。Universal Multiple-Octet Coded Character Set，简称为UCS。现在用的是UCS-2，即2个字节编码，而UCS-4是为了防止将来2个字节不够用才开发的。Unicode（统一码、万国码、单一码）是一种在计算机上使用的字符编码。它为每种语言中的每个字符设定了统一并且唯一的二进制编码，以满足跨语言、跨平台进行文本转换、处理的要求。1990年开始研发，1994年正式公布。随着计算机工作能力的增强，Unicode也在面世以来的十多年里得到普及。Unicode是基于通用字符集（Universal Character Set）的标准来发展，并且同时也以书本的形式（The Unicode Standard，目前第五版由Addison-Wesley Professional出版，ISBN-10: 0321480910）对外发表。2005年3月31日推出的Unicode 4.1.0。2021年9月14日推出的14.0版本。

大概来说，Unicode编码系统可分为编码方式和实现方式两个层次。

编码方式：Unicode是国际组织制定的可以容纳世界上所有文字和符号的字符编码方案。Unicode用数字0-0x10FFFF来映射这些字符，最多可以容纳1114112个字符，或者说有1114112个码位。码位就是可以分配给字符的数字。UTF-8、UTF-16、UTF-32都是将数字转换到程序数据的编码方案。Unicode字符集可以简写为UCS（Unicode Character Set）。早期的Unicode标准有UCS-2、UCS-4的说法。UCS-2用两个字节编码，UCS-4用4个字节编码。UCS-4根据最高位为0的最高字节分成2^7=128个group。每个group再根据次高字节分为256个平面（plane）。每个平面根据第3个字节分为256行（row），每行有256个码位（cell）。group 0的平面0被称作BMP（Basic Multilingual Plane）。将UCS-4的BMP去掉前面的两个零字节就得到了UCS-2。每个平面有216=65536个码位。Unicode计划使用了17个平面，一共有17×65536=1114112个码位。在Unicode 5.0.0版本中，已定义的码位只有238605个，分布在平面0、平面1、平面2、平面14、平面15、平面16。其中平面15和平面16上只是定义了两个各占65534个码位的专用区（Private Use Area），分别是0xF0000-0xFFFFD和0x100000-0x10FFFD。所谓专用区，就是保留给大家放自定义字符的区域，可以简写为PUA。平面0也有一个专用区：0xE000-0xF8FF，有6400个码位。平面0的0xD800-0xDFFF，共2048个码位，是一个被称作代理区（Surrogate）的特殊区域。代理区的目的用两个UTF-16字符表示BMP以外的字符。在介绍UTF-16编码时会介绍。如前所述在Unicode 5.0.0版本中，238605-65534×2-6400-2408=99089。余下的99089个已定义码位分布在平面0、平面1、平面2和平面14上，它们对应着Unicode定义的99089个字符，其中包括71226个汉字。平面0、平面1、平面2和平面14上分别定义了52080、3419、43253和337个字符。平面2的43253个字符都是汉字。平面0上定义了27973个汉字。

实现方式：在Unicode中：汉字的“字”对应的数字是23383。在Unicode中，我们有很多方式将数字23383表示成程序中的数据，包括：UTF-8、UTF-16、UTF-32。UTF是“UCS Transformation Format”的缩写，可以翻译成Unicode字符集转换格式，即怎样将Unicode定义的数字转换成程序数据。例如，“汉字”对应的数字是0x6c49和0x5b57，而编码的程序数据是：BYTE data\_utf8[] = {0xE6, 0xB1, 0x89, 0xE5, 0xAD, 0x97}； WORD data\_utf16[] = {0x6c49, 0x5b57}；DWORD data\_utf32[] = {0x6c49, 0x5b57}；这里用BYTE、WORD、DWORD分别表示无符号8位整数，无符号16位整数和无符号32位整数。UTF-8、UTF-16、UTF-32分别以BYTE、WORD、DWORD作为编码单位。“汉字”的UTF-8编码需要6个字节。“汉字”的UTF-16编码需要两个WORD，大小是4个字节。“汉字”的UTF-32编码需要两个DWORD，大小是8个字节。根据字节序的不同，UTF-16可以被实现为UTF-16LE或UTF-16BE，UTF-32可以被实现为UTF-32LE或UTF-32BE。其中UTF-8、UTF-16、UTF-32、字节序和BOM的具体实现方式可以由读者另外查阅文献或者上网浏览了解。。

### 3.6.3音频存储

声音本身是模拟信号，而计算机只能识别数字信号，要在计算机中处理声音，就需要将声音数字化，这个过程叫经模数转换（A/D变换）。最常见的方式是透过脉冲编码调制PCM(Pulse Code Modulation) 。

而工作中常见的音频文件根据是压缩程度可以分为三种类型：未压缩的音频格式、无损压缩音频格式、有损压缩音频格式。

未压缩音频格式：音频文件是根据真实世界的声波转换成数字格式保存下来，不需要进行压缩和其他处理。未压缩音频格式优点：可以保留录制音频的详细信息、真实性很好。缺点：未压缩的原始音频文件会占用大量空间、不利于本地保存和流式多媒体的网络传输。主要的未压缩音频文件格式包括：wav、aiff、pcm。

无损压缩音频格式：无损压缩指的是通过使用拥有高级算法的无损压缩技术，用户可以在缩小文件体积的同时保留原始数据。理想情况下，无损压缩技术可以使文件大小减小2到5倍，同时仍保留原始数据。常见的无损压缩音频格式包括：flac、alac、wma。平时使用最为广泛的是flac格式。就文件自身而言，FLAC和ALAC都是免版税的压缩技术，而WMA则不是。此外，WMA遵循严格的DRM限制，传播性比不上FLAC、ALAC。

有损压缩音频格式：日常生活中，大多数人并不希望音乐文件占用大量设备空间，因此人们常常使用有损压损的音频格式。它们采用有损压缩技术，可以大大减小文件体积，但音频的原始数据也会受到损害。有时，以此格式存储的音乐文件听起来甚至与原始音频毫不相像。常见的有损压缩音频格式包括：mp3、aac、ogg、wma。其中wma可以根据用户需求采用有损压缩技术和无损压缩技术。

表3.9 各种音频文件的扩展名

|  |  |  |
| --- | --- | --- |
| 音频类型 | 格式 | 扩展名后缀 |
| 未压缩音频格式 | wav | .wav |
| aiff | .aiff .aif .aifc |
| pcm | .pcm .aiff .au .wav |
| 无损压缩音频格式 | flac | .flac |
| alac | .m4a .caf |
| wma | .wma .wmv |
| 有损压缩音频格式 | mp3 | .mp3 |
| ogg | .ogg .oga .mogg |
| aac | .aac |

其中，Wave格式是微软公司开发的一种声音文件格式，它符合PIFF（Resource Interchange File Format） 文件规范，用于保存WINDOWS平台的音频信息资源，被WINDOWS平台及其应用程序所支持。“\*.WAV”格式支持多种音频位数、采样频率和声道，是PC上流行的声音文件格式，其文件尺寸比较大，多用于存储简短的声音片段。

Audio文件是Sun Microsystems公司推出的一种经过压缩的数字音频格式，是Internet中常用的声音文件格式。Audio文件原先是UNIX操作系统下的数字声音文件。由于早期INTERNET上的WEB服务器主要是基于UNIX的，所以，AU格式的文件在如今的INTERNET中也是常用的声音文件格式。

MPEG（Moving Picture Experts Group，动态图像专家组）代表运动图像压缩标准，这里的音频文件格式指的是MPGE标准中的音频部分，即MPGE音频层（MPEG Audio Layer），因其音质与存储空间的性价比较高，至于2011年使用最多的是MP3格式。

MP3是一种音频压缩技术，其全称是动态影像专家压缩标准音频层面3（Moving Picture Experts Group Audio Layer III），简称为MP3。它被设计用来大幅度地降低音频数据量。利用 MPEG Audio Layer 3 的技术，将音乐以1:10 甚至 1:12 的压缩率，压缩成容量较小的文件，而对于大多数用户来说重放的音质与最初的不压缩音频相比没有明显的下降。它是在1991年由位于德国埃尔朗根的研究组织Fraunhofer-Gesellschaft的一组工程师发明和标准化的。用MP3形式存储的音乐就叫作MP3音乐，能播放MP3音乐的机器就叫作MP3播放器。MP3是利用人耳对高频声音信号不敏感的特性，将时域波形信号转换成频域信号，并划分成多个频段，对不同的频段使用不同的压缩率，对高频加大压缩比（甚至忽略信号）对低频信号使用小压缩比，保证信号不失真。这样一来就相当于抛弃人耳基本听不到的高频声音，只保留能听到的低频部分，从而将声音用1∶10甚至1∶12的压缩率压缩。由于这种压缩方式的全称叫MPEG Audio Player3，所以人们把它简称为MP3。

### 3.6.4图像存储

在初中物理中我们接触到三基色（红绿蓝）。三种基色的光可以合成所有颜色的光（而如果三种基色的光都没有，就没有光，就呈现出黑色），这就是我们可以看到彩色图片（显示器上的，发光的）的基础。

另外，物理学上还有三原色，是红黄蓝，三基色和三原色的区别是他们的原理，一种利用加色法（三基色）进行颜色合成，另一种利用减色法（三原色）进行颜色合成，通过减色法无法合成白色。三原色是我们可以看到的彩色图片（纸质照片，不发光，依靠反射光的）的基础

物理学上喜欢将一件物体无限的往最小方向分解，直到分解不了为止，因此我们看到了原子。在图像处理上，我们也将图片进行了分解，分解到最后，只剩下一个点，这个点就叫像素点。而一张图片，由很多很多的像素点组成，以的摄像头像素（1200万像素）为例，一张照片就有1200万个像素点。

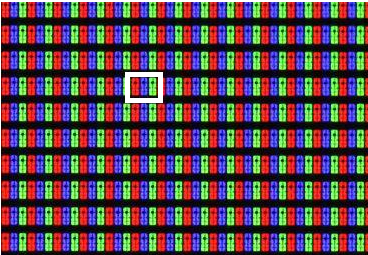


图3.2

如上图3.2所示，每三个发光单元构成一个像素点。

有了物理基础，那么计算机只要通过数据控制每个像素点就可以了。上文提到过，每个像素点都有三种颜色构成，每张图片由很多很多的像素点组成，我们将像素点进行排列，就可以得到一个像素点矩阵（可以理解为一个二维数组）。而每个像素点需要记录这个颜色的信息，那就又回到了物理学的范畴了。三基色调整颜色是通过三种颜色的光的发光强度不同来实现光的混合的。那么我们在每个像素点中记录每个像素对应的rgb光的强度值，就能实现混合得到的光的颜色。

通过上述原理，数码图像最重要的格式——rgb模式就介绍得差不多了。rgb模式的原理就是通过一个二维数组来记录每个像素点的位置，每个像素点都有rgb三个属性值用来记录对应的数值（实际上应该是一个三维数组）。如下图所示的灰度图像，可以很清晰的看出他的像素矩阵。

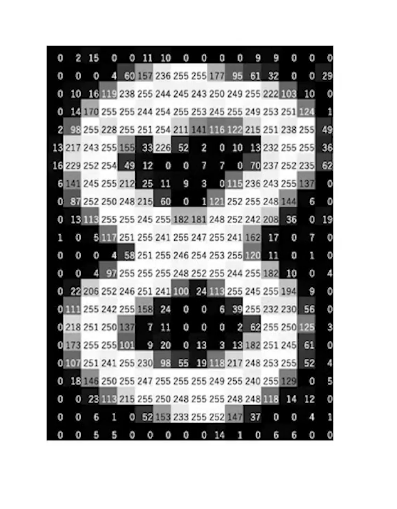


图3.3 灰度图像数字“8”

在rgb模式中，每个属性值用一八位的二进制数值（也就是一字节，而一字节可以表示256种状态），一个像素点有三个这样的属性值，也就是说他可以合成256\*256\*256（1677多万）种颜色，也就是我们所说的真彩色。

rgb的出现是基于彩色显示屏的，在没有彩色显示屏的时代，其实使用的图片记录方式也很相似。比如以下要介绍的几种模式：

位图模式:我们在rgb模式中介绍到，rgb每个像素点用三个八位的二进制数值进行表现，而位图只用一个一位的二进制数值表现，所以位图只有两种颜色，也就是白色和黑色。

灰度模式：与rgb模式类似，但它采用的是一个八位的二进制数值进行表现，这个数值只控制白色的强度（也就是灰度）。

CMYK模式：CMYK模式是用来打印或印刷的模式，它是相减的模式。当C、M、Y三值达到最大值时，在理论上应为黑色，但实际上因颜料的关系，呈显的不是黑色，而是深褐色。为弥补这个问题，所以加进了黑色K。由于加了黑色，CMYK共有四个通道，正因为如此，对于同一个图像文件来说，CMYK模式比RGB模式的信息量要大四分之一。但RGB模式的色域范围比CMYK模式大。因为印刷颜料在印刷过程中不能重现RGB色彩。

上文介绍了灰度图像在计算机中的存储，接下来让我们看一个彩色图像的示例。



图3.4 彩色图像示例

如上图3.4所示，该图像由许多颜色组成，几乎所有颜色都可以从三种原色（红色，绿色和蓝色）生成。我们可以说每个彩色图像都是由这三种颜色或3个通道（红色，绿色和蓝色）这就意味着在彩色图像中，矩阵的数量或通道的数量会更多，如下如3.5所示的多通道矩阵，可以看出各个通道上每一位上的值(每个数字代表像素强度)，在这3个通道组合在一起就构成了一副彩色图像。

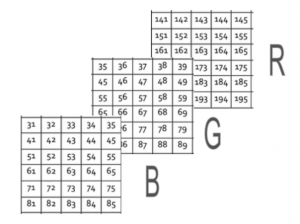


图3.5 彩色图像多通道矩阵示例

接下来让我们看看一些常见的图像存储格式。

JPEG（ Joint Photographic Experts Group）即联合图像专家组，是用于连续色调静态图像压缩的一种标准，文件后缀名为.jpg或.jpeg，是最常用的图像文件格式。其主要是采用预测编码（DPCM）、离散余弦变换（DCT）以及熵编码的联合编码方式，以去除冗余的图像和彩色数据，属于有损压缩格式，它能够将图像压缩在很小的储存空间，一定程度上会造成图像数据的损伤。尤其是使用过高的压缩比例，将使最终解压缩后恢复的图像质量降低，如果追求高品质图像，则不宜采用过高的压缩比例。

然而，JPEG压缩技术十分先进，它可以用有损压缩方式去除冗余的图像数据，换句话说，就是可以用较少的磁盘空间得到较好的图像品质。而且JPEG是一种很灵活的格式，具有调节图像质量的功能，它允许用不同的压缩比例对文件进行压缩，支持多种压缩级别，压缩比率通常在10；1到40；1，压缩比越大，图像品质就越低；相反地，压缩比越小，图像品质就越高。同一幅图像，用JPEG格式存储的文件是其他类型文件的1/10~1/20，通常只有几十KB，质量损失较小，基本无法看出。JPEG格式压缩的主要是高频信息，对色彩的信息保留较好，适合应用于互联网；它可减少图像的传输时间，支持24位真彩色；也普遍应用于需要连续色调的图像中。

JPEG格式可分为标准JPEG、渐进式JPEG及JPEG2000三种格式。

1. 标准JPEG格式；此类型在网页下载时只能由上而下依序显示图像，直到图像资料全部下载完毕，才能看到图像全貌。

2. 渐进式JPEG；此类型在网页下载时，先呈现出图像的粗略外观后，再慢慢地呈现出完整的内容，而且存成渐进式JPG格式的文档比存成标准JPG格式的文档要来得小，所以如果要在网页上使用图像，可以多用这种格式。

3. JPEG2000；它是新一代的影像压缩法，压缩品质更高，并可改善在无线传输时，常因信号不稳造成马赛克现象及位置错乱的情况，改善传输的品质。

GIF格式的名称是Graphics Interchange Format的缩写，是在1987年由Compu Serve公司为了填补跨平台图像格式的空白而发展起来的。GIF可以被PC和Mactiontosh等多种平台上被支持。

GIF是一种位图。位图的大致原理是：图片由许多的像素组成，每一个像素都被指定了一种颜色，这些像素综合起来就构成了图片。GIF采用的是Lempel-Zev-Welch（LZW）压缩算法，最高支持256种颜色。由于这种特性，GIF比较适用于色彩较少的图片，比如卡通造型、公司标志等等。如果碰到需要用真彩色的场合，那么GIF的表现力就有限了。GIF通常会自带一个调色板，里面存放需要用到的各种颜色。在Web运用中，图像的文件量的大小将会明显地影响到下载的速度，因此我们可以根据GIF带调色板的特性来优化调色板，减少图像使用的颜色数（有些图像用不到的颜色可以舍去），而不影响到图片的质量。

GIF格式和其他图像格式的最大区别在于，它完全是作为一种公用标准而设计的，由于Compu Serve网络的流行，许多平台都支持GIF格式。Compu Serve通过免费发行格式说明书推广GIF，但要求使用GIF文件格式的软件要包含其版权信息的说明。

### 数据运算（逻辑运算、移位运算、算术运算）

### 3.7.1移位运算

1. 移位的意义

移位运算在日常生活中常见。例如,15 m可写成1 500 cm，单就数字而言，l500相当于15相对于小数点左移了两位，并在小数点前面添了两个0。同样15也相当于1 500相对于小数点右移了两位，并删去了小数点后面的两个0。可见,当某个十进制数相对于小数点左移n位时，相当于该数乘以10n;右移n位时，相当于该数除以10n。

计算机中小数点的位置是事先约定的，因此二进制表示的机器数在相对于小数点作n位左移或右移时,其实质就是该数乘以或除以2n(n = 1 ,2,…" ,n )。

移位运算称为移位操作,对计算机来说,有很大的实用价值。例如，当某计算机没有乘(除)法运算线路时,可以采用移位和加法相结合,实现乘(除)运算。

计算机中机器数的字长往往是固定的，当机器数左移n位或右移n位时，必然会使其n位低位或n位高位出现空位。那么，对空出的空位应该补0还是1呢?这与机器数采用有符号数还是无符号数有关。对有符号数的移位称为算术移位。

1. 算术移位

对于正数来说，因为[x]S=[x]O=[x]t=真值(整数的原码，反码，补码一样)，故移位后出现的空位均以0添之。对于负数而言 ，由于原码、补码和反码的表示形式不同，故当机器数移位时，对其空位的添补规则也不同。表3.10列出了三种不同码制的机器数(整数或小数均可)，分别对应正数或负数移位后的添补规则。值得注意的是：不论是正数还是负数，移位后其符号位均不变，这是算术移位的重要特点。

表3.10 不同码制机器数算术移位后的空位添补规则

|  |  |  |
| --- | --- | --- |
| 真值 | 码制 | 添加位 |
| 正数 | 原码/补码/反码 | 0 |
| 负数 | 原码 | 0 |
| 补码 | 左移添0 |
| 右移添1 |
| 反码 | 1 |

由表3.10可知：

1. 在机器数为正时，不论是左移还是右移，补的位都是0。
2. 由于负数的原码在数值部分与真值相同，所以在移位时只要使最高位符号位不变，移位产生的空位均补0即可。
3. 由于负数的反码除符号位外，各位与负数原码正好相反，故移位后所添的代码应与原码相反，即全部补1.
4. 对于负数的补码来书，当对其由低位向高位找到第一个1，在此1左边的各位均与对应的反码相同，而在此1右边的各位(包括该1)均与对应的原码相同。故负数的补码左移时，因空位出现在低位，则添补的位与原码相同，即补0。而右移时，因空位出现在最高位，则补的位应与反码相同，即1.

下面我们来看正数移位的例子。机器数字长为8位(含1位最高位符号位)。以数字A=26为例。

首先，其原码，反码，补码分别如下表示：



移位结果如下表3.11所示。

表3.11 +26的移位结果

|  |  |  |
| --- | --- | --- |
| 移位操作 | 机器数 | 对应真值 |
| 移位前 | 0001 1010 | 26 |
| 左移一位 | 0011 0100 | 52 |
| 左移两位 | 0110 1000 | 104 |
| 右移一位 | 0000 1101 | 13 |
| 右移两位 | 0000 0110 | 6 |

还可以从上表发现，在左移的时候，每次移的位数就等于该数乘以移的位数。而右移刚好相反，为除以。左移两位时就等于26×4=104，右移时对应除法，但是在右移两位时可以发现对应的真值出错，这是因为是整除的关系。这样的关系在接下来我们讨论的乘除法中会有进一步介绍。

对于负数而言，表3.12列举了数-26移位的一些情况。

表3.12 -26的移位结果

|  |  |  |  |
| --- | --- | --- | --- |
| 移位操作 | 机器数 | | 真值 |
| 移位前 | 原  码 | 1001 1010 | -26 |
| 左移一位 | 1011 0100 | -52 |
| 左移两位 | 1110 1000 | -104 |
| 右移一位 | 1000 1101 | -13 |
| 右移两位 | 1000 0110 | -6 |
| 移位操作 | 机器数 | | 真值 |
| 移位前 | 反  码 | 1110 0101 | -26 |
| 左移一位 | 1100 1011 | -52 |
| 左移两位 | 1001 0111 | -104 |
| 右移一位 | 1111 0010 | -13 |
| 右移两位 | 1111 1001 | -6 |
| 移位操作 | 机器数 | | 真值 |
| 移位前 | 补  码 | 1110 0110 | -26 |
| 左移一位 | 1100 1100 | -52 |
| 左移两位 | 1001 1000 | -104 |
| 右移一位 | 1111 0011 | -13 |
| 右移两位 | 1111 1001 | -7 |

可见,对于负数,三种机器数算术移位后符号位均不变。负数的原码左移时，高位丢1，结果出错；右移时，低位丢1，影响精度。负数的补码左移时，高位丢0，结果出错；右移时，低位丢1，影响精度。负数的反码左移时，高位丢0,结果出错；右移时，低位丢0，影响精度。

另外在移位过程中，移除的最高位或者最低位会被保存至计算机中的进位标志中，用C(Carry flag)表示，如下图3.6所示。



图3.6 算术移位运算示意图

1. 算术移位与逻辑移位的区别

有符号数的移位称为算术移位，无符号数的移位称为逻辑移位。逻辑移位的规则是：逻辑左移时，高位移丢，低位添0；逻辑右移时，低位移丢，高位添0。例如，寄存器内容为0101 0011，逻辑左移为1010 0110。算术左移为0010 0110(最高数位1移丢)。又如，寄存器内容为1011 0010，避辑右移为0101 1001，若将其视为补码，算术右移为1101 1001。显然，两种移位的结果是不同的。上例中为了避免算术左移时最高数位丢1，可采用带进位C的移位，其示意图如图3.7所示。



图3.7 用带进位的移位实现算术左移

### 3.7.2算术运算

1. 加法与减法运算(补码实现)

现代计算机中用二进制补码系统表示有符号整数，因为它可以将减法运算转换为对减数的补数的加法运算，所以计算机中都采用补码做加减法运算。

因为涉及到补码的一些知识，本节将再给出补码的一些相关概念。一个数与它的补数之和是一个常数；例如一个一位十进制数与它的补数之和总是9；2的补数是7，因为2+7=9。在n位二进制算术中，数P的补数为Q，且P+Q=2n。

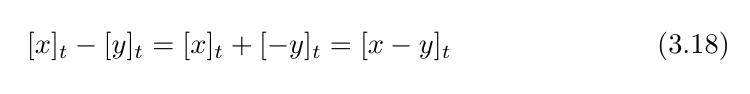
在二进制算术中，求一个数补数的方法是将其各位取反并加1。例如，01100101的补数为10011010+1=10011011。人们之所以对补码算术感兴趣，就是因为减一个数等于加上这个数的补数。因此，一个二进制数减01100101等价于它加上补数10011011。

补码加法的运算规则为：



也就是说，两数补码的和等于两数和的补码。不论被加数，加数是正数还是负数，只要直接用它们的补码直接相加即可。当结果不超出补码所表示的范围时，计算结果便是正确的补码形式。但是当计算结果超出补码表示范围时，结果会出错，这种情况称为溢出，在后面一节中我们会讨论此种情况。

同理，我们给出补码减法的运算规则：



因此，若机器数采用补码，当求A-B时，只需要先求出-B的补码(把-B的补码称为求补后的减数)，就可以按照补码加法的规则进行。要想求-B的补码有两种方法：一是直接根据B来求，直接在B的补码连同符号位在内，每位取反，末位加1；第二种就是直接写出-B的原码，在转变为补码。这两种方式都是正确的。

也就是说，无论被减数是正数还是负数，上述补码减法的规则都是正确的。同样，由最高位向更高位的进位会自动丢失而不影响运算结果的正确性。

计算机中带符号数用补码表示时会有很多优点。可以将减法运算变为加法运算，因此可以使用同一个运算器实现加法和减法运算，简化数字电路。另外，无符号数和带符号数的加法运算可以用同一个加法器实现，结果都是正确的。

下面我们分别来看加法和减法的一些示例，以加深了解。

我们以±5和±7为例，首先将它们全部用补码表示：

|  |  |  |  |
| --- | --- | --- | --- |
| +5=0000 0101 | -5=1111 1011 | +7=0000 0111 | -7=1111 1001 |

现在要求7-5，其过程如下所示：

|  |  |  |
| --- | --- | --- |
|  | 0000 0111 | 7 |
| + | 1111 1011 | -5 |
|  | 10000 0010 | 2 |

结果为一个九位二进制数1 0000 0010，如果忽略掉最左边一位(也叫做“进位位”)，结果为0000 0010=+2，这符合正确结果。

我们再来求5-7，其过程如下所示：

|  |  |  |
| --- | --- | --- |
|  | 0000 0101 | 5 |
| + | 1111 1001 | -7 |
|  | 01111 1110 | -2 |

同样结果为一个9位二进制数0 1111 1110，如果忽略掉进位位0，那么得到得到就是-2的补码1111 1110，也符合正确结果。

再例如，设机器字长为8位，最高位用作符号表示，A=-93，B=+45，求A-B的补。

同样A，-B的补码表示如下：

|  |  |
| --- | --- |
| A = 1010 0011 | -B=1101 0011 |

运算过程如下：

|  |  |  |
| --- | --- | --- |
|  | 1010 0011 | -93 |
| + | 1101 0011 | -45 |
|  | 10111 0110 | 118 |

当把最左边的进位1丢掉时，会发现结果为正数118，两个负数相加结果得到正数，这显然发生了错误。这是因为A-B=-138超出了8位机器字长下所能表示的数的范围(对于n为二进制补码而言，其数的表示范围是-2n-1 ~ 2n-1-1，式(3.12)也有说明)。所以对于8位二进制补码来说，其表示范围为-128~127，共有256个数。所以如果有-129，那么它实际因为循环表示应是127，所以-138就对应118。

1. 进位与溢出

所谓进位，是指运算结果的最高位向更高位的进位，用来判断无符号数运算结果是否超出了计算机所能表示的最大无符号数的范围。

溢出是指带符号数的补码运算溢出，用来判断带符号数补码运算结果是否超出了补码所能表示的范围。例如，字长为n位的带符号数，它能表示的补码范围为-2n-1 ~ 2n-1-1，如果运算结果超出此范围,就叫补码溢出，简称溢出。

显然，溢出只能出现在两个同号数相加，或两个异号数相减的情况下。

接下来我们将介绍溢出的判断方法：

PC中常用的溢出判别法是双高位判别法。假设引进两个附加的符号，即

Cs：它表示最高位(符号位)的进位情况，如有进位，Cs=1，否则为0。

Cp：它表示数值部分最高位的进位情况，如有进位,Cp=1，否则为0。

当符号位向前有进位时，Cs=1，否则，Cs=0；当数值部分最高位向前有进位时，Cp=1，否则，Cp=0。单符号位法就是通过该两位进位状态的异或结果来判断是否溢出的(异或运算：两个值不同结果为1，两个值相同结果为0，都是针对二进制数)。



若OF=1，说明结果溢出;若OF=0,则结果未溢出。也就是说，当符号位和数值部分最高位同时有进位或同时没有进位时，结果没有溢出，否则,结果溢出。

具体地讲，对于加运算，如果次高位(数值部分最高位)形成进位加入最高位，而最高位(符号位)相加(包括次高位的进位)却没有进位输出时；或者反过来，次高位没有进位加入最高位，但最高位却有进位输出时，都将发生溢出。因为这两种情况分别是：两正数相加，结果超出了范围，结果形式上变成了负数；两负数相加，结果超出了范围，结果形式上变成了正数。

对于减运算，当次高不需从最高位借位，但最高位却需要借位(正数减负数，差超出范围)；或者反过来，次高位需要从最高位借位，但最高位不需借位(负数减正数，差超出范围)，也会出现溢出。

下面我们来看一个示例。

求73+98，其过程如下，都用补码表示：

|  |  |  |
| --- | --- | --- |
|  | 0100 1001 | 73 |
| + | 0110 0010 | 98 |
|  | 1010 1011 | -86 |

它们的此高位(数值位部分最高位)是两个1，相加有进位，Cp=1，但是符号位向前无进位，Cs=0，异或运算得出OF=1，说明发生溢出。

发生溢出一般会通过循环得出一个值，例如在8位二进制情况下，如下图3.8所示。



图3.8 8位二进制补码的循环表示

所以73+98=171，超出了127，那么得到的值会通过循环用负数表示，也就是-86。

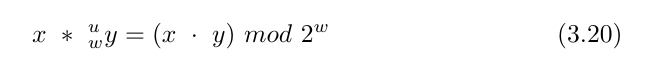
1. 乘法运算
2. 无符号乘法

范围在内的整数x和y可以被表示为w位的无符号数，但是它们的乘积的取值范围为0到之间。这可能需要位来表示。不过，C语言中的无符号乘法被定义为产生w位的值，就是2w位的整数乘积的低w位表示的值，我们把计算得到的值用表示(u代表无符号，w代表x和y的二进制位数)。

将一个无符号数截断为w位等价于计算该值模，得到：

无符号数乘法原理：

对满足的x和y有：

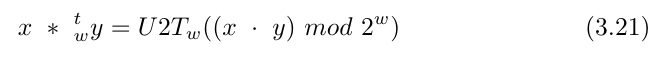


1. 补码乘法

范围在，内的整数x和y可以被表示为w位的补码，但是它们的乘积的取值范围为到之间，要想用补码表示这个乘积，可能需要2w位。然而，C语言中的有符号乘法是通过将2w位的乘积截断为w位来实现的。同上面的记法一样，我们将得到的数值表示为(t代表补码)。将一个补码数截断为w位相当于先计算该值模，在把无符号数转换为补码，得到：

补码乘法原理：

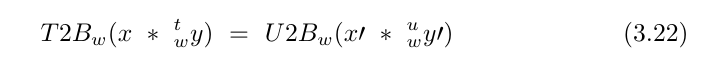
对满足的x和y有：



我们认为对于无符号和补码乘法来说，乘法运算的位级表示都是一样的，下面给出公式说明：

原理：无符号和补码乘法的位级等价性

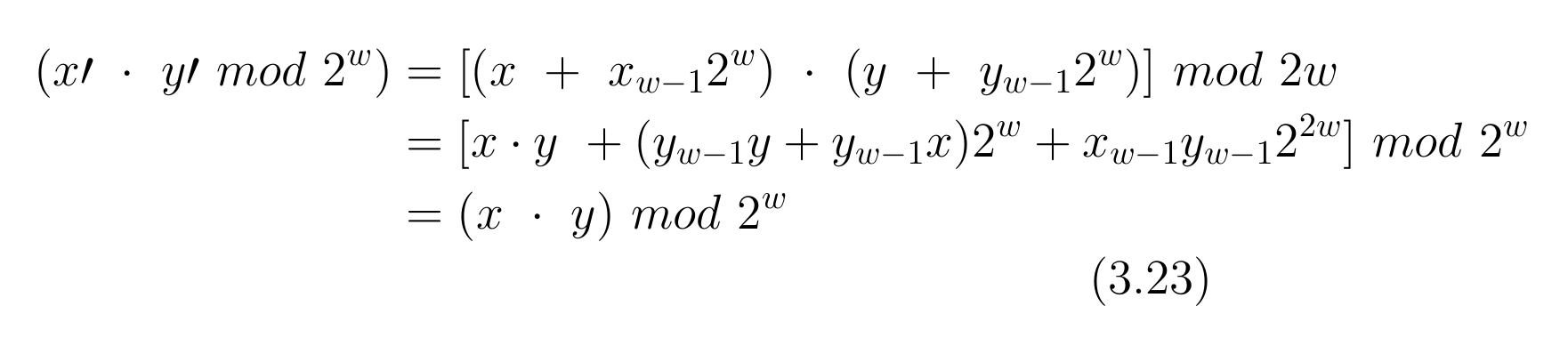
给长度位w的二进制位向量和，用补码形式的二进制位向量表示来定义整数x和y：(这个转换在补码那一节中有过说明)。用无符号形式的二进制位向量来定义非负整数和：。则有：



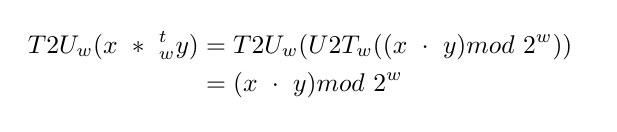
作为说明，图3.9给出了不同3位数字的乘法结果。对于每一对位级运算数，我们执行无符号和补码乘法，得到6位的乘积，然后再把这些乘积截断到3位。无符号的截断后的乘积总是等于。虽然无符号和补码两种乘法乘积的6位表示不同，但是截断后的乘积的位级表示都相同。

推导：无符号和补码乘法的位级等价性。

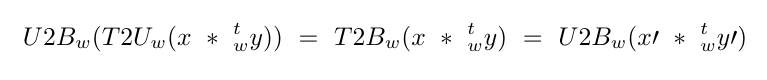
根据公式(3.20)，我们有和。计算这些值的乘积模得到以下结果：



由于模运算符，所有带有权重和的项都将被丢弃。根据公式(3.21)，我们有。对等式两边应用操作有：



将上述结果与公式(3.21)和公式(3.23)结合起来得到。然后对这个等式的两边应用，得到



1. 乘以常数
2. 除法运算
3. 数字逻辑与数字系统

### 4.1基本逻辑关系

真值表，离散数学，

### 4.2逻辑门

与或非，及其组合

### 4.3逻辑代数与逻辑函数