# 1 数制基础和机器数运算

“数制”是“数据进制”或“数据制式”的简称，也就是数据逢几进一的意思，如我们常用的十进制就是逢十进位。当然，数制的类型远不只十进制，在计算机系统中常见的还有二进制、八进制和十六进制这三种。

本章主要介绍以上各种不同类型数制的特点及它们之间的相互转换方法、在计算机中参与运算的机器数类型（定点数和浮点数）、机器数的各种表示形式（原码、反码、补码、阶码和移码），以及它们的算术/逻辑运算方法。

本章内容是为我们学习本书以后各章涉及数制方面的内容打基础的，如后面章节中将要介绍的各种信息编码方式、IPv4/IPv6 地址格式转换、IPv4 网络地址的计算、IPv4 子网划分与聚合、MAC 地址格式转换等。另外，本章所讲内容对于我们在日常故障排除中所进行的数据包分析也是非常必要的。当然，如果你对这些内容已有掌握，可直接跳过。

**本章重点与难点：**

- 十进制数与非十进制数间的相互转换（要区分整数和小数部分的不同转换方法）
- 机器数的五种编码方式（原码、反码、补码、阶码和移码）
- 机器数的表示形式及存储结构
- IEEE 754 标准浮点数的计算
- 二进制数的算术、逻辑运算方法
- 补码和浮点数的加、减法运算方法

## 1.1 数制概述

“数制”是“数据进制”的简称，是指数据的进位计数规则，又称为“进位计数制”，简称“进制”。本节先来简单地了解一些常见的数制类型及其特点。

### 1.1.1 常见数制类型及表示方法

日常生活中我们经常使用的数是十进制的，如我们拿的 3000 元工资、市场 1.5 元/斤的菜价等。之所以称其为十进制，是因为这类数是逢十进一的。除了十进制计数以外，还有许多其他进制的计数方法。在计算机中常见的有二进制、八进制、十六进制等制式，这三种进制的数在进行加法运算中分别是逢二、八、十六进一，这就是前面所说的进位计数规则。关于如何理解这些不同数制类型数据的加法运算，在本章后面将有专门介绍。

其实数制类型远不止这么几种，如我们以 60 分钟为 1 小时，60 秒为 1 分钟，用的就是六十进制计数法；一天之中有 24 小时，用的是二十四进制计数法；一星期有 7 天，用的是七进制计数法。



虽然数据制式可以有很多种，但在计算机通信中通常遇到的仍是以上提到的二进制、八进制、十进制和十六进制这 4 种。在一种数制中所能使用的数码的个数称为该数制的“基数”，也就对应数制类型的名称，如二进制的基数为“2”，八进制的基数为“8”，十进制的基数为“10”，十六进制的基数也就是“16”。这里所说的“基数”其实就是前面所说的进位计算规则，如我们常见的十进制数是逢十进一，二进制数是逢二进一，……。

既然有不同的数制，那么在计算机程序中给出一个数时就必须指明它属于哪一种数制，否则计算机程序就不知道该把它看成是哪种数了（当然，在计算机及其他计算机网络设备内部都是以二进制进行运算的）。如 12300 这个数，既可能是十进制又可能说是八进制或者十六进制，所以数需要有专门的标志来进行区别。

（1）十进制（Decimal）。

十进制是日常生活中常用的数制类型，基数是 10，也就是它有 10 个数字符号，即 0、1、2、3、4、5、6、7、8、9。其中最大数码是“基数”减 1，即 10-1=9，最小数码是 0。十进制数的标志为 D，如(1250)D，也可用下标“10”来表示，如$(1250)_{10}$（注意是下标）。也可以不加这些标志，默认就是十进制。

（2）二进制（Binary）。

二进制是计算机运算时所采用的数制，基数是 2，也就是说它只有两个数字符号，即 0 和 1。如果在给定的一个数的表示形式中，除 0 和 1 外还有其他数（如 1061），那它就绝不会是一个二进制数了。二进制数的最大数码也是基数减 1，即 2-1=1，最小数码也是 0。二进制数的标志为 B，如(1001010)B，也可用下标“2”来表示，如$(1001010)_2$（注意是下标）。

（3）八进制（Octal）。

八进制的基数是 8，也就是说它有 8 个数字符号，即 0、1、2、3、4、5、6、7。对比十进制可

以看出，它比十进制少了两个数“8”和“9”，这样当一个数的表示形式中出现“8”和（或）“9”时（如 23459），那它也就绝不是八进制数了。八进制数的最大数码也是基数减 1，即 8-1=7，最小数码也是 0。八进制数的标志为 O 或 Q（注意它特别一些，可以有两种标志），如(4603)O（注意是字母 O，不是数字 0）、(4603)Q，也可用下标“8”来表示，如$(4603)_8$（注意是下标）。在 C、C++这类编程语言中规定，一个数如果要指明它采用八进制，必须在它前面加上一个 0，如 123 是十进制，但 0123 则表示采用八进制。

（4）十六进制（Hexadecilnal）。

十六进制数用得比较少，最新的 IPv6 地址就是采用十六进制来表示的（IPv4 地址通常采用的是十进制表示）。在注册表中也会用到十六进制，所以了解十六进制还是非常必要的。

十六进制的基数是 16，也就是说它有 16 个数码，除了十进制中的 10 个数码可用外，还使用了 6 个英文字母（分别代表 10、11、12、13、14、15），这样一来，十六进制的这 16 个数码依次是 0、1、2、3、4、5、6、7、8、9、A、B、C、D、E、F（不区分大小写）。对比前面对其他几种数制的介绍可以看出：如果一个数的表示式中出现了字母，如 63AB，则一定不会是二进制、八进制或十进制，而是十六进制了。

十六进制的最大数码也是基数减 1，即 16-1=15（为 F），最小数码也是 0。十六进制数的标志为 H，如(4603)H，也可用下标“16”来表示，如$(4603)_{16}$（注意是下标）。十六进制数也常用前缀 0x 来表示（注意是数字 0，而不是字母 O）。在 C、C++这类编程语言中也规定，十六进制数必须以 0x 开头。比如 0x10 表示一个十六进制数，而不是八进制或者十进制的 10。


既然在计算机中使用的是二进制，那为什么还要十进制、八进制和十六进制呢？其实这都不是计算机自身要求的，因为在计算机运算中全都是使用二进制。之所以还需要这些数制，完全是出于表达和识别的方便性考虑的。

因为一个较大的数用二进制表示的话就太长了，如一个 C、C++等编程语言中的 int（整数）类型的数据要占用 4 个字节，也就是 32 位。比如 100，用 int 类型的二进制数表达将是：0000 0000 0000 0000 0110 0100。这还是一个比较小的数，如果数更大，则会更复杂。试想一下，要写这么长，估计没几个人会喜欢，于是就有了可以更简便表示的十进制、八进制和十六进制，因为数制越大，表示一个数所需的数码位数就越少。所以，像 C、C++这类语言没有提供在代码中直接输入二进制数的方法，而是普遍采用八进制或十六进制。

那为什么不是其他进制类型，如九进制或二十进制呢？原因就在于 2、8、16 分别是 2 的 1 次方、3 次方、4 次方，这就使得这三种进制之间可以非常直接地互相转换。八进制或十六进制缩短了数的表示位数，但保持了二进制数的表达特点。在下面关于进制转换的介绍中你就可以发现这一点。

以上 4 种进制的基本特点比较如表 1-1 所示。

表 1-1　4 种进制的基本特点比较

| 数制 | 基数（数码个数） | 数码 | 表示方式 |
|---|---|---|---|
| 二进制 | 2 | 0、1 | 数后面加 B 或下标 2 |
| 八进制 | 8 | 0、1、2、3、4、5、6、7 | 数后面加 O 或 Q 或下标 8 |
| 十进制 | 10 | 0、1、2、3、4、5、6、7、8、9 | 数后面加 D 或下标 10，也可以不加 |
| 十六进制 | 16 | 0、1、2、3、4、5、6、7、8、9、A、B、C、D、E、F | 数后面加 H 或下标 16 |

### 1.1.2　不同数制之间的对应关系

表 1-2 所示是以上介绍的二进制、十进制、八进制和十六进制这 4 种常在计算机中使用的数制的对应关系。注意，因为八进制没有 8 和 9，所以二进制 1000 对应的八进制是 10，而不是想象中的 8，二进制 1001 对应的八进制是 11，而不是想象中的 9。这个表很重要，大家最好全部记下来，特别是这几种数制的对应关系，这样可以方便以后不同数制间转换时的计算。



表 1-2　不同数制的对应关系

| 二进制数 | 对应的十进制数 | 对应的八进制数 | 对应的十六进制数 |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |
| 10 | 2 | 2 | 2 |
| 11 | 3 | 3 | 3 |
| 100 | 4 | 4 | 4 |
| 101 | 5 | 5 | 5 |
| 110 | 6 | 6 | 6 |
| 111 | 7 | 7 | 7 |
| 1000 | 8 | 10 | 8 |
| 1001 | 9 | 11 | 9 |
| 1010 | 10 | 12 | A |
| 1011 | 11 | 13 | B |
| 1100 | 12 | 14 | C |
| 1101 | 13 | 15 | D |
| 1110 | 14 | 16 | E |
| 1111 | 15 | 17 | F |

## 1.2 不同数制间的相互转换

同一个数在一些环境中（如要进行子网划分、计算子网掩码时、对信息进行编码时等），可能要用不同数制形式来表示，这就涉及到数制间的转换问题了。下面是常见的十进制、二进制、八进制、十六进制之间的转换方法。

### 1.2.1 非十进制数转换成十进制数

非十进制转换成十进制就是把这些非十进制按位以对应的权值（注意要区分整数位和小数位）展开，然后相加即得出相应的十进制值。本节后面介绍的各种非十进制转换成十进制的方法都是按照这种方法进行的。

“权值”是指对应数值位的进制幂次方数，如二进制整数中第 0 位（最低位，也就是整数最右边的那位）的权值是 2 的 0 次方，第 1 位的权值是 2 的 1 次方，……。同理在八进制整数中第 0 位的权值是 8 的 0 次方，第 1 位的权值是 8 的 1 次方，……，依此类推。但每位的权值会因是整数位还是小数位而不同：

- 整数的第 0 位（也就是最低位）的权值为对应进制的 0 次方，最高位的权值为对应进制的 *n*-1 次方。
- 小数的第一位（最靠近小数点的那位，也是小数的最高位）的权值为对应进制的-1 次方，最后一位（最右边的那位，亦即小数的最低位）的权值为对应进制的-*n* 次方。

1. 二进制转换为十进制

二进制转换成十进制的方法大家可能早就有所了解了，如在 IPv4 地址计算时就经常进行这样的操作。转换的方法比较简单，只需按它的权值展开即可。展开的方式是把二进制数首先写成加权系数展开格式，然后按十进制加法规则求和。这种做法称为“按权相加”法。

二进制整数部分的一般表现形式为：$b_{n-1}\ldots b_1b_0$（共 n 位），按权相加展开后的展开格式为（注意，展开式中从左往右各项的幂次是从高到低下降的，最高位的幂为 n-1，最低位的幂为 0）：

$$b_{n-1}\times2^{n-1}+ b_{n-2}\times2^{n-2}+\ldots+b_1\times2^1+b_0\times2^0$$

**【示例 1】**二进制数$(11010)_2$的按权相加展开格式为：

$$1\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 0\times2^0 = 16+8+0+2+0=(26)_{10}$$

二进制小数部分的幂次是反序排列的（也就是与整数部分的幂次序列相反，从右往左其绝对值是从低到高上升的），且为负值，最高位幂次（也就是最靠近小数点的第一个小数位的幂次）为“-1”。如二进制小数部分的格式为：$0.b_{n-1}\ldots b_1b_0$，则按权相加后的展开格式为：

$$b_{n-1}\times2^{-1}+ b_{n-1}\times2^{-2}+\ldots+b_1\times2^{-(n-1)}+b_0\times2^{-n}$$

**【示例 2】**$(0.1011)_2$的按权相加展开格式为：

$$1\times2^{-1}+0\times2^{-2}+1\times2^{-3}+1\times2^{-4}=0.5+0+0.125+0.0625=(0.6875)_{10}$$

2. 八进制转换为十进制

八进制转换成十进制也是采取“按权相加”法，只是这里的权值是 8 的相应幂次方。如八进制整数部分的格式为：$b_{n-1}\ldots b_1b_0$，则按权值相加展开后的格式就为（从左往右幂次是从高到低下降的）：

$$b_{n-1}\times8^{n-1}+b_{n-2}\times8^{n-2}+\ldots+b_1\times8^1+b_0\times8^0$$

【示例 3】八进制数$(26356)_8$的按权值相加展开格式为：

$$2\times8^4+6\times8^3+3\times8^2+5\times8^1+6\times8^0=8192+3072+192+40+6=(11502)_{10}$$

八进制小数部分的幂次也是反序排列的（也就是与整数部分的幂次序列相反，从右往左其绝对值是从低到高上升的），且为负值，最低幂次（也就是最靠近小数点的第一个小数位的幂次）为“-1”。如八进制小数部分的格式为：$0.b_{n-1}\ldots b_1b_0$，则按权相加后的展开格式为：

$$b_{n-1}\times8^{-1}+b_{n-2}\times8^{-2}+\ldots+b_1\times8^{-(n-1)}+b_0\times8^{-n}$$

【示例 4】$(0.257)_8$按权相加的展开格式为：

$$2\times8^{-1}+5\times8^{-2}+7\times8^{-3}=0.25+0.078125+0.013671875=(0.341796875)_{10}$$

3. 十六进制转换为十进制

十六进制转换成十进制的方法也是采取“按权相加”法，只是这里的权值是 16 的相应幂次方。如十六进制整数部分的格式为：$b_{n-1}b_{n-2}\ldots b_1b_0$，则按权相加展开后的格式就为（从左往右幂次是从高到低下降的）：

$$b_{n-1}\times16^{n-1}+b_{n-2}\times16^{n-2}+\ldots+b_1\times16^1+b_0\times16^0$$

【示例 5】十六进制数$(26345)_{16}$的按权相加展开后的格式为：

$$2\times16^4+6\times16^3+3\times16^2+4\times16^1+5\times16^0=131072+24576+768+64+5=(156485)_{10}$$

十六进制小数部分的幂次也是反序排列的（也就是与整数部分的幂次序列相反，从右往左其绝对值是从低到高上升的），且为负值，最低幂次（也就是最靠近小数点的第一个小数位的幂次）为“-1”。如十六进制小数部分的格式为：$0.b_{n-1}\ldots b_1b_0$，则按权相加后的展开格式为：

$$b_{n-1}\times16^{-1}+b_{n-2}\times16^{-2}+\ldots+b_1\times16^{-(n-1)}+b_0\times16^{-n}$$

【示例 6】$(0.25A)_{16}$按权值相加展开后的格式为：

$$2\times16^{-1}+5\times16^{-2}+10\times16^{-3}=0.125+0.01953125+0.00244140625=(0.14697265625)_{10}$$

### 1.2.2 十进制数转换成非十进制数

十进制数转换成非十进制数的方法是：整数部分的转换用“除基取余法”，也就是用基数相除，然后从最后的商开始反序（由后向前）取余数；小数部分的转换用“乘基取整法”，也就是用基数相乘，然后正序（由前向后）取得到的整数。这里的“基数”就是对应的数制，如二进制的基数为 2，八进制的基数为 8，十六进制的基数为 16。

1. 十进制转换为二进制

这里分别对十进制整数和十进制小数转换成二进制进行介绍。

（1）十进制整数转换成二进制的方法。

十进制整数转换为二进制的方法是采用“除 2 逆序取余”法（采用短除法进行）。也就是先将十进制数除以 2，得到一个商数（也是下一步的被除数）和余数；然后再将商数除以 2，又得到一个商数和余数；依此类推，直到商数为小于 2 的数为止。然后从最后一步得到的小于 2 的商开始将其他各步所得的余数（也都是小于 2 的 0 或 1）排列起来（俗称“逆序排列”）就得到了对应的二进制数。

**【注意】**这里与下面的小数转换有些不一样，这里要包括最后得到的小于 2 的商数，而小数转换中是不需要包括最后的积的，只包括各步得到的整数部分，后面的十进制整数转换为八进制、十六进制也一样。

**【示例 1】**图 1-1 所示为十进制整数 48 转换成二进制数时依次除 2 的过程。

在每步的最右边显示的是各步商数除 2 所得到的余数，最后一步的商数为 1，因为它小于 2，所以不能再除了。然后从最后得到的商数（1）开始依次向上把其他各步除 2 得到的余数排列起来，就得到最后 48 转换成二进制时的结果为$(110000)_2$。同理，图 1-2 所示的十进制数 250 转换成二进制数后的结果就为$(11111010)_2$。

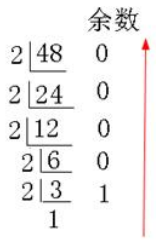



图 1-1　十进制整数 48 转换成二进制整数的步骤

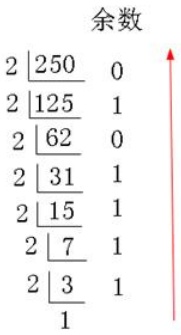



图 1-2　十进制整数 250 转换成二进制整数的步骤

（2）十进制小数转换成二进制的方法。

十进制小数转换为二进制的方法是采用“乘 2 正序取整”法。也就是用 2 乘十进制小数，得到一个积，然后将积的整数部分取出作为相应步骤得到的整数；再用 2 乘余下的小数部分，又得到一个积，然后再将这个积的整数部分取出；依此类推，直到积中的小数部分为零，或者达到所要求的精度为止；最后把各步取出的整数部分（仅需要各步得到的整数部分，而不需要最后没有取整的小数部分）按正序排列起来，即先取的整数作为二进制小数的高位，后取的整数作为低位。

**【示例 2】**图 1-3 中左、右图所示的是分别将十进制小数 0.125 和 0.625 转换成二进制的过程。

最后得到的二进制数就是从最开始得到的整数值开始一直到最后得到的整数值（也就是自上而下的顺序，与整数转换中取余的顺序相反）。0.125 和 0.625 最后的二进制值分别为$(0.001)_2$和$(0.101)_2$（注意，一定要记得在整数部分加上“0.”，因为十进制小数转换成二进制后仍是小数）。

**【说明】**有些十进制小数乘以 2 后是个无穷循环数，永远不会有完整的整数，此时就要看所需的精度如何了，按所需位数精度取值即可。如 0.825 就是这样一个数，如果仅要求是小数点后 3

位，则相应的二进制数为$(0.110)_2$，如果要求为 4 位，则对应的二进制值为$(0.1101)_2$，具体如图 1-4 所示。

如果一个十进制同时有整数和小数部分，则要对整数和小数部分分别按以上介绍的对应方法进行二进制转换。

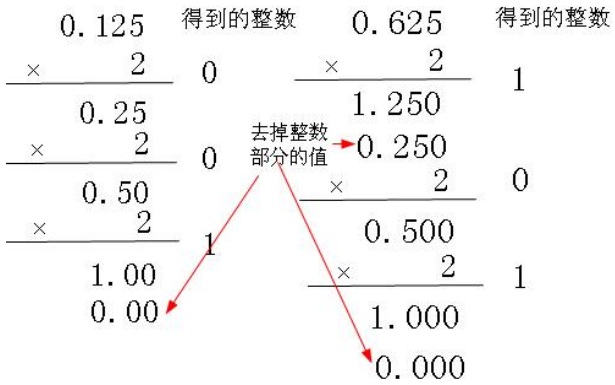



图 1-3　0.125 和 0.625 十进制小数转换成二进制小数的步骤

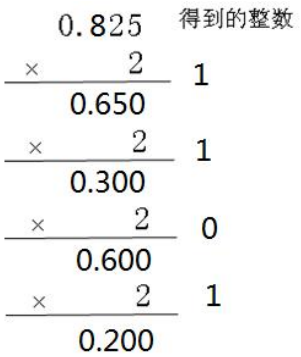



图 1-4　不同精度要求的取值示例

2. 十进制转换成八进制

十进制转换成八进制的方法与前面介绍的十进制转换成二进制的方法类似，只不过这里的基数是 8（而不再是 2）。十进制转换成八进制当然也分整数部分和小数部分两种不同的转换方法。

十进制整数转换为八进制整数采用“除 8 逆序取余”法，直到所得的商小于 8，然后把余数（包括最后一步中得到的小于 8 的商）按逆序排列即可；十进制小数转换为八进制小数采用“乘 8 正序取整”法，直到所得到的积小数部分为 0 或者在规定的精度范围内，然后把所得到的整数正序排列起来即可。

**【示例 3】**图 1-5 中左、右图所示的是分别将十进制整数 65 和 2467 按“除 8 逆序取余”法转换成八进制的步骤，得到的结果分别是$(101)_8$和$(4643)_8$。

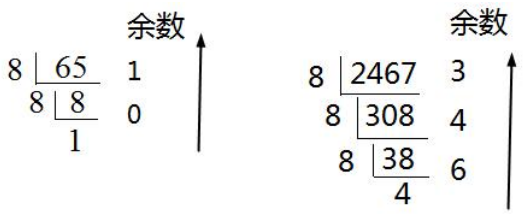



图 1-5　两个十进制整数转换成八进制的步骤

**【示例 4】**图 1-6 中左、右图所示的是两个十进制小数 0.125 和 0.8125 通过“乘 8 正序取整”法转换成八进制的步骤，得到的结果分别是$(0.1)_8$和$(0.64)_8$（是正序排列，一定要记得在整数部分加上“0.”）。

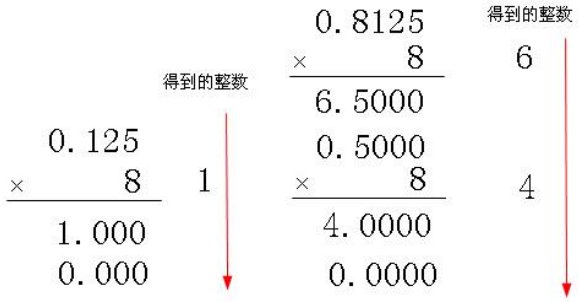



图 1-6　两个十进制小数转换成八进制小数的步骤

3. 十进制转换成十六进制

十进制转换成十六进制与十进制转换成二进制类似，将十进制整数转换为十六进制的方法是采用“除 16 逆向取余”法，直到所得的商小于 16，然后把余数（包括最后一步中得到的小于 16 的商）按逆序排列即可；十进制小数转换为十六进制的方法是采用“乘 16 正序取整”法，直到所得到的积小数部分为 0 或者在规定的精度范围内，然后把所得到的整数正序排列起来。

**【示例 5】**图 1-7 左、右图所示的是分别将十进制整数 45 和 3456 按“除 16 逆序取余”法转换成十六进制的步骤，得到的结果分别是$(2D)_{16}$和$(D80)_{16}$（注意，其中的 13 用十六进制的 D 表示了）。

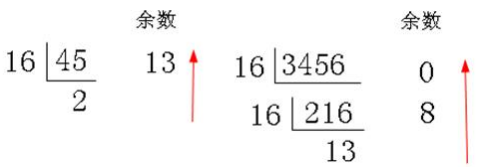



图 1-7　两个十进制整数转换成十六进制的步骤

**【示例 6】**图 1-8 中左、右图所示的是将十进制小数 0.125 和 0.825 通过“乘 16 正序取整”法转换成十六进制的步骤，得到的结果分别是$(0.2)_{16}$和$(0.D33)_{16}$（精确到小数点后面三位）（注意：是

正序排列，也一定要记得在整数部分加上“0.”，仍是小数）。

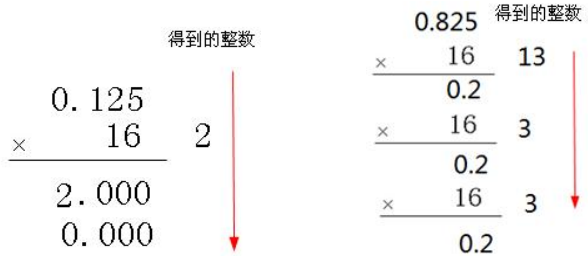



图 1-8　两个十进制小数转换成十六进制小数的步骤

### 1.2.3　非十进制数之间的相互转换

从表 1-2 可以得出这样一个规律：1 位八进制数对应 3 位二进制数，而 1 位十六进制数对应 4 位二进制数。因此，二进制数与八进制数之间、二进制数与十六进制数之间的相互转换十分容易。

1. 八进制数转换成二进制数的方法

将每 1 位八进制数直接用相应的 3 位二进制来表示；二进制数转换成八进制数的方法是：以小数点为边界，整数部分向左，小数部分向右将每 3 位二进制分成一组，若不足 3 位则用 0 补足 3 位；然后将每一组二进制数直接用相应的 1 位八进制来表示。

**【示例 1】**将$(3456.2262)_8$转换为二进制数的方法是依次把八进制的每 1 位用 3 位二进制表示（如图 1-9 所示），最后的结果为$(11100101110.010010110010)_2$（整数部分最前面的 0 可以省略）。


图 1-9　八进制转换成二进制示例

**【示例 2】**将$(1101011.10111)_2$转换为八进制数，可把整数部分从右向左每 3 位分为一组，最后不足 3 位时加 0 补上，然后把小数部分从左向右（与整数部分的划分顺序相反）同样以每 3 位分为一组，最后不足 3 位时加 0 补上（如图 1-10 所示），最后的结果为$(153.56)_8$。

2. 十六进制数转换成二进制数的方法

将每 1 位十六进制数直接用相应的 4 位二进制来表示；二进制数转换成十六进制数的方法是：以小数点为边界，整数部分向左，小数部分向右将每 4 位二进制数分成一组，若不足 4 位则用 0 补足 4 位；然后将每一组二进制数直接用相应的 1 位十六进制表示。

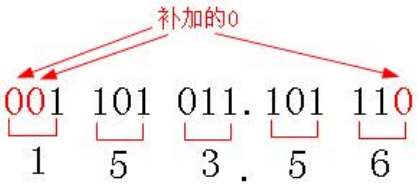



图 1-10　二进制转换成八进制示例

**【示例 3】**将$(4AF.51)_{16}$转换成二进制数的最后结果为$(100\ 1010\ 1111.0101\ 0001)_2$；将二进制数$(11\ 0110\ 1110.\ 1010\ 1010\ 1\mathbf{000})_2$（最后加粗部分的 0 是划分位时补上去的，下同）转换为十六进制数的最后结果为$(36E.AA8)_{16}$。

3. 八进制与十六进制的相互转换方法

先把其中一个转换成二进制，然后再把所得到的二进制转换成另一个进制的数。

**【示例 4】**将八进制数$(6237.431)_8$转换成十六进制的步骤如下：

（1）将$(6237.431)_8$ 转换成二进制（每 1 位用 3 位二进制表示），得到的二进制结果是$(110010011111.100011001)_2$。

（2）再将$(1100\ 1001\ 1111.1000\ 1100\ 1\mathbf{000})_2$转换成十六进制（以小数点为边界，整数部分从右向左每 4 位分为一组，最后不足 4 位时加 0 补上，然后把小数部分从左向右将每 4 位二进制数分成一组，不足 4 位则用 0 补足），得到最终的十六进制结果为$(C9F.8C8)_{16}$。

**【示例 5】**将十六进制$(3AB.11)_{16}$转换成八进制的步骤与上面的步骤类似，具体转换过程如下：

（1）先将十六进制数$(3AB.11)_{16}$转换成二进制，得到$(001110101011.00010001)_2$。

（2）再将$(001\ 110\ 101\ 011.000\ 100\ 01\mathbf{0})_2$转换成八进制，以小数点为边界，整数部分从右向左每 3 位分为一组，最后不足 3 位时加 0 补上，然后把小数部分从左向右将每 3 位二进制数分成一组，不足 3 位则用 0 补足，最后的结果为$(1653.042)_8$。

以上二进制、八进制、十六进制的对应关系可参见表 1-2，根据该表可直接进行转换。

## 1.3　机器数基础

在本节之前介绍的数制之间的转换没有考虑二进制数是带有符号位的情况，当然这不影响我们对前面介绍的数制间转换方法的学习，因为数制之间的转换是不包括符号位的，即符号位保持不变。本节我们专门来介绍带符号的二进制数——机器数的各方面的知识。

“机器数”是计算机中参与运算且带有正（+）、负（-）属性的二进制数。即机器数是有符号位的二进制数，并且规定最高位表示符号位，用 0 表示正数，用 1 表示负数，所以机器数也称为符号数值化的二进制数。如+1101=01101，-1101=11101。没有符号位的二进制数称为“无符号数”，也不是机器数，不参与各种运算，如各种信息编码、ID 编号等。

### 1.3.1 机器数的真值

前面说了，机器数是带有符号的二进制数，这样一来在计算机中参与运算的机器数涉及到两个问题：一是机器数的真实数值（即“真值”）如何得出，因为机器数中的符号位也是用 1、0 来表示的；二是存储器的单位存储长度（即“字长”），因为机器数的存储和读取是受到计算机中 CPU 一次可以处理的数据位数或者内存中用于存储数据的基本单位限制的。本节先介绍什么是机器数的“真值”。

在机器数中规定最高位用来表示数据符号，其中 1 代表为负，0 代表为正。这样一来，存储在内存中的机器数本身就不能代表对应数据的真正数值了，只有去掉最高的符号位后才是数值本身，这就是机器数的“真值”。

**【示例】**机器数 10000101 的最高位 1 代表负（-），所以余下来的“0000101”才是数值本身，所以其真正数值（即“真值”）是-5。如果是无符号数，则 10000101 所代表的是 133。

为了区别，带符号位的机器数对应的真正数值称机器数的“真值”。例如，00100001 的真值 = 0 0100001 = +33（正号可以不写，可以直接写成 33），10100011 的真值 = 1 0100011 = -35。



### 1.3.2 机器数的字长

“字长”可以简单理解为用来表示一个机器数所用的二进制位数。如果确定了字长，则每个机器数都是用固定的二进制位数来表示，不管这个机器数的大与小。但字长不同，同一个机器数的表示形式也不一样。如果字长为 8 位，十进制中的数+5 转换成二进制就是 00000101，-5 转换成二进制就是 10000101；但如果字长是 16 位，+5 转换的结果就是 00000000 00000101，而-5 转换成二进制就是 10000000 00000101 了。也就是对应的机器数要转换为字长所代表的位数，因为一个机器数必须单独占用一个字长（非机器数没这个要求，由其对应的编码方式决定），否则计算机无法知道一个存储单元中的机器数到底是多少。

**【注意】**“-0”与“0”的机器数在内存中的存储是不一样的，在 8 位字长中，-0 为 1 0000000，而+0 为 0 0000000；在 16 位字长中，-0 为 1 0000000 00000000，而+0 为 0 0000000 00000000。所以在二进制的机器数中，0 也有两个（-0 和 0），且表示形式并不一样。

另外，字长的选定也不是随意的，它与我们通常所说的 16 位、32 位或 64 位 CPU、内存等硬件体系架构是一一对应的。这里的 16 位、32 位、64 就对应了在相应的计算机系统中的字长。计算机内存中数据的存储是以单位长度进行连续存储的，一个机器数只能在一个存储单元中存储，一个存储单元也只能存储一个机器数，因为计算机中的 CPU 也是以存储单元为单位从内在中读取并处理数据的。所以，“字长”也是指计算机一次可处理的机器数的码位长度，是计算机进行数据存储和数据处理的运算单位。如我们通常所指的 32 位处理器就是指该处理器的字长为 32 位，也就是一次能处理最大数值是 32 位的数。

显然，字长越长，CPU 的数据处理能力越强，因为它一次可以处理的数据越多。我们知道，8 位是一个字节，16 位就是两个字节。现在的计算机设备基本上都是 64 位字长的，也就是在内存的

一个存储单元中可以存 8 个字节，64 位的 CPU 一次也可以从内存中读取并处理 8 个字节。当然，这也决定了不同字长所能表示的有效机器数范围（机器数的最高位为符号位，不参与计算）是不同的，表 1-3 列出了主要字长所能表示的机器数大小范围。

表 1-3　不同字长下机器数表示范围

| 字长 | 真值位长 | 机器数表示范围 |
|---|---|---|
| 8 位 | 7 | $-2^7-1 \leqslant +2^7-1$，即-127≤+127 |
| 16 位 | 15 | $-2^{15}-1 \leqslant +2^{15}-1$，即-32767≤+32767 |
| 32 位 | 31 | $-2^{31}-1 \leqslant +2^{31}-1$ |
| 64 位 | 63 | $-2^{63}-1 \leqslant +2^{63}-1$ |

## 1.4　机器数的编码形式

为了使机器数的运算方法适用于所有机器数，而且运算不会出现二义性，在机器数的运算方法设计过程中出现了 3 种不同的机器数编码方式：原码、反码、补码。另外，在本章后面将要介绍的浮点机器数运算中又引入了“阶码”和“移码”这两个概念。下面对这 5 种表示编码进行具体介绍。



### 1.4.1　原码

对于人脑来说，我们都知道，+表示正数，-表示负数，但在计算机二进制中也引入这两个符号肯定是不行的，因为在计算机中只有 0 和 1 这两个数字，根本不认识“+”和“-”这两个符号。计算机中的任何行为都依赖它的物理结构，它没有思维，所以你得在 0 和 1 之间让计算机识别出对应数的正与负。

1. 原码的编码规则

“原码”就是“原始码位”或“原始编码”的意思，就是对应机器数本身所代表的编码形式。即机器数的最高位代表符号，1 表示负，0 表示正，其余各位为真值位。

**【注意】**一个机器数的原码表示形式是与字长有关的，这在 1.3.2 节已有介绍。当指定了字长时，如果原来的机器数真值二进制码长度+1（符号也要占一位）小于字长时，则包括了整数的机器数，要在真值（除最高位为符号位外，其余各位均为真值位）最前面（最左边）加上 0 来补足；纯小数的机器数，要在真值最后面（最右边）加 0 补足。

比如，+3 的符号位为 0，真值 3 转化为二进制就是 11，如果不考虑字长的话，则+3 的原码就是 011；但如果指定字长为 8，则+3 的原码就是 **0000000**11 了（加粗部分为补的 5 个 0）。同理， -3 的符号位为 1，真值 3 转化为二进制就是 11，如果不考虑字长的话，-3 的原码是 111；但如果指定字长为 8，则-3 的原码就是 **1000000**11（加粗部分为补的 5 个 0）。如 0.11010 的机器数，指定字长为 8，则其原码为 0.1101**000**（加粗部分为补的 2 个 0，小数点是不计数位的）。

在日常的书写中，原码的表示形式是用方括号下面加上一个“原”字下标来区别的，如$[+3]_{原}$

= 00000011，$[-3]_{原}$= 10000011。

2. 原码的优点与缺点

原码的优点是设计简单、容易理解。原码的缺点有如下几个：

- 加减运算规则复杂（符号位要进行单独处理），还可能出错。在采用原码直接进行机器数运算时，同为正数的加、减法运算是没什么问题的，可是异号相加、减时就存在问题了。如 1-1，如果用原码计算的话则如下所示，结果为-2，显然不正确：

$$(0\ 0000001)_{原} + (1\ 0000001)_{原} = (1\ 0000010)_{原} = -2$$

- 原码中的 0 有两种编码方式，存在二义性。如果是 8 位字长，则 0 有 000000000 和 10000000，即有+0 和-0 之分，有二义性。所以直接采用原码进行机器数运算不合适。

### 1.4.2 补码

正因为原码在机器数运算过程中存在以上不足，所以设计者又继续寻找其他更好的编码方式。最后终于找到了一种能解决原码的那些不足的另一种编码方式——补码。很显然，补码不可能再采用机器数的本身编码方式（即原码）来进行了，需要采用新的编码方式。



1. 补码的编码规则

补码的编码规则为：正数的补码和原码相同；负数的补码是通过先把除符号位外（使数的符号位不变，正、负属性保持不变）其他各位取反（就是求下面将要介绍的该数的“反码”），再在末位（最低位）加 1 得到的。这样，我们只要让减数通过一个求反电路，再通过一个+1 电路，然后再通过加法器就可以实现减法运算了。

如一原码为 01001 的机器数，因为最高位为 0，所以是正数，则它的补码与原码一样为 01001。如另一个原码为 11001 的机器数，因为最高位为 1，所以是负数，先对原码除符号位外的其他各位取反，得到 10110，再在结果的最低位加 1，得到 10111，这就是机器数 11001 的补码了。

2. 补码的优点

补码继承了部分原码的特点（可以表示正与负），并且它有两个新的优点：

- 可以把符号位一起运算，符号位不需要单独处理。

**【示例 1】**同样以前面的 1-1 为例，如果采用补码形式，则算式如下，结果完全正确：

$$(0\ 0000001)_{补} + (1\ 1111111)_{补} = 0\ 0000000 = 0$$

- 0 只有一种表示形式，没有二义性。

+0 的补码就是其原码本身，字长为 8 位时为 00000000，-0 的原码为 10000000，其补码是先对低 7 位取反，得到 11111111，再在最低位加 1，如仍是 8 位字长的话，结果为 00000000（最高位进的“1”被溢出了），这就是-0 的补码，也和+0 的补码一样。

3. 补码转换成原码的方法

前面介绍了原码转换成补码的方法，如果要把一个补码转换成原码该如何操作呢？很简单，只需要把原码转换成补码的过程倒过来、反向操作即可。

因为正数的补码与原码一样，所以正数补码的原码就是其补码，而负数补码的原码是先在最后

一位减 1（具体的减法运算方法在本章后面介绍），然后对其除符号位外的其他各位全部取反得到。所以关键还是负数补码与原码之间的转换问题。

**【示例 2】**如已知一个补码为 100110，要求它的原码。按照前面介绍的方法负数的补码是先在补码的最后一位减 1（如图 1-11 所示），得到 100101；然后对以上结果除符号位外的其他各位取反，得到 111010，这就是补码 100110 的原码。

下面再对以上示例通过前面介绍的由原码转换成补码的方法来验证以上由补码求原码的方法的正确性。把前面计算出的原码 111010 转换成补码后是否为 100110。

原码 111010 的最高位为 1，为负数，先对除符号位外的其他各位取反，得到 100101，然后再在最低位加 1（如图 1-12 所示），得到 100110，恰好与最初的补码一致。证明前面介绍的由补码求原码的方法是正确的。

```
  100110
-      1
--------
  100101
```

图 1-11　100100-1 的运算

```
  100101
+      1
--------
  100110
```

图 1-12　100101+1 的运算

**【注意】**只有相同码制的数才能进行操作，结果就是对应的码制，也就是原码数与原码数的运算结果也为原码，反码数与反码数的运算结果也为反码，补码数与补码数的运算结果也为补码。如果结果是负数，要判断结果是否正确，需要再将其对应的码制转换为原码。

### 1.4.3　反码

通过以上介绍我们可以知道，补码是我们的最佳选择，因为它的符号位与真值位一样，都参与运算，可以全面地解决异号二进数之间的加减运算问题。那为什么还有“反码”这种表示形式呢？

其实“反码”是“原码”向“补码”表示形式转变过程中的一个过渡形式，不是可直接用于机器数运算的一种编码方式。之所以当初会想到“反码”，是因为它太容易从电路上来实现了（仅需要取反即可）。

与前面介绍的“补码”一样，其实“反码”也可以说是专门针对负数的，因为正数的反码与补码、原码一样。它的编码规则为：对负机器数除符号位外的其他各位逐位取反，即原来为 1 就变为 0，原来为 0 就变为 1。由此可见，补码编码方式中也用到了“反码”。

**【说明】**其实反码在 IPv4 地址的子网掩码计算时是必须要用到的，因为我们知道，子网掩码是对 IPv4 地址中的网络 ID 部分保持不变，而主机 ID 部分则必须全为 0，这里就可能需要对 IPv4 地址中的网络 ID 部分的位进行取反，由 1 变为 0 了。有关 IPv4 地址方面将在第 9 章中进行介绍。

对于负机器数来说，反码只是对原码的除符号位外的其他各位取反（正机器数的反码与它的原码、补码都一样），所以它同样具有原码的一些不足：

（1）运算结果有时不正确。

如“2-3”采用反码运算的格式如下，结果是正确的：

$(0\ 0000010)_{反}+(1\ 1111100)_{反}=(1\ 1111110)_{反}=-1$。

但是在 1-1 的反码运算中，结果就不正确了：

$(0\ 0000001)_{反}+(1\ 1111110)_{反}=(1\ 1111111)_{反}=-0$，本来应该是+0 的，结果就成了-0。

再如十进制数“123-121”，用反码加法运算就得到：

$(0\ 1111011)_{反}+(1\ 0000110)_{反}=(0\ 0000001)_{反}=1$，显然也不正确。

**【注意】**反码是相互的，如 10011001 的反码为 11100110（符号位是不变的），相反 11100110 的反码也是 10011001。另外，与原码一样，在反码表示形式中 0 也有“+0”和“-0”之分，对应的反码分别为 0 0000000 和 1 1111111。

（2）0 也有二义性。

+0 的反码当然就是它的原码 00000000（假设字长为 8），-0 的原码为 10000000，由此可得出它的反码为 11111111。即+0 与-0 的反码不一样。

综上可以得出：正数的原码、反码和补码都是一样的，而负数的这 3 种表示形式就不一样了，负数的反码是对原码中除符号位外的其他各位取反，而负数的补码是再对其反码加 1，也就是先对其原码中除符号位的其他各位取反，然后再在最低位加 1。



### 1.4.4 阶码

“阶码”主要用于本章后面将要介绍的浮点机器数的表示。在计算机存储器中表示一个浮点数时需要给出指数，而不是直接在存储器中存储这个浮点数，因为浮点数中的小数点位置是不确定的。而用来表示具体浮点数中小数点位置的这个指数就是这里所说的“阶码”，是用整数形式表示的。

对于任意一个二进制机器数 N，均可用 $N=S\times 2^{P}$ 的形式表示，其中 S 为尾数，P 为阶码，2 为阶码的底。P、S 都用二进制数表示，S 表示 N 的全部有效数字，P 指明小数点的位置。这类似于十进制数中的科学记数法。当阶码为固定值时，表示的数称为“定点数”，这时阶码的意义不大；当阶码为可变时，表示的数称为“浮点数”，这是阶码的主要应用。

有关“定点数”和“浮点数”将在本章后面具体介绍。

### 1.4.5 移码

“移码”可以理解为“偏移的编码”，它有两个方面的应用：一是方便机器数的大小比较，二是用于浮点数中“阶码”的修正，它们都是通过在原来编码的基础上加上一个常数来实现的（但实现方式不一样）。下面具体就这两方面的应用进行介绍。

1. 在机器数比较方面的应用

我们已经知道，机器数的运算都是采用补码方式，因为采用补码运算时符号位与真值位的运算方法一样。但在进行机器数比较时，当机器数的真值用补码表示时，由于符号位和数值部分一起编码，与习惯上的表示法不同，因此人们很难从补码的形式上直接判断其真值的大小。

如十进制数 X=31 对应的二进制数为+11111，则$[X]_{补}$=011111；十进制数 X=-31，对应的二进制数为-11111，则$[X]_{补}$=100001。看上去好像 100001>011111，其实正好相反。如果我们对每个机器数的原码加上一个 $2^n$（n 代表原来二进制编码的真值部分位数，原码符号位用“+”或“-”表示），则 31 对应的+11111，再加上 $2^5$，即得到 11111+100000=111111；而-31 对应-11111+$2^5$（此时相当于 100000-11111），即得到 000001。然后拿两数转换后得到的结果 111111 和 000001 就可以直接通过二进制代码比较大小了。

移码的计算公式：$X_{移}=X+2^n$。X 代表原机器数的原码（符号位用“+”或“-”表示），n 代表原来二进制编码的真值位数。如果要计算 11010101 这个机器数的移码，首先去掉最高的符号位得出真值一共有 7 位，由此可以得到该数的原码为-1010101，再加上 $2^7$，即-1010101+1000000=00101011，这就是 11010101 这个机器数的移码。

其实以上公式也是移码与阶码的转换关系，即 $X_{移}=X_{阶}+2^n$，因为阶码通常是原码形式。由此可以得出移码与补码的关系：移码是补码的符号位取反，其他位不变。这样一来，在移码中，正数的符号位为 1，负数的符号位为 0，与前面的原码、反码和补码中的符号位规定相反，即补码与移码的关系是符号位互反的关系。

我们再用移码与补码的以上这种关系来计算前面 11010101 这个机器数的移码。此时要先计算出 11010101 这个机器数的补码。补码是除符号位外其他各位取反，然后再加 1 得到。11010101 的除符号位外其他各位取反后的值为 10101010，再加 1 就得到 10101011。再把最高的符号位取反（其他位不变），得到 11010101 的移码为 00101011，与前面通过移码公式计算得出的结果完全一样。

【说明】有关机器数的算术运算方法将在本章后面介绍。

2. 在浮点数阶码中的应用

前面说到了浮点数的“阶码”用来表示指数大小，也有正负之分。但这样一来，一个机器数在存储器中存储时就可能要包括两个符号位：一个为机器数本身的符号位，一个为阶码的符号位。

为了节省存储空间，省去阶码的符号位，就采用了对每个阶码都加上一个相同的、比较大的正常数（称偏移值），这样就使所有阶码值都为正整数（可以省去符号位）。并且在运算时系统会自动对指数部分减去某个固定的常数，就可以保持运算结果不变。这时用来表示浮点数小数点位置的“阶码”就变成“偏移”了的阶码，也就是这里所介绍的“移码”，又称“增码”。

在浮点数的阶码中所加上的这个常数称为“偏移值”，具体将在本章后面介绍。

## 1.5 机器数的分类

前面说到，机器数就是计算机中参与运算的有符号二进制数。其实它与我们平时使用的十进制一样，有整数也有小数。但关键的不同点是在存储器中并没有为小数点提供二进制码位，这就决定了机器数在存储器中的表示形式与我们平时书写十进制数时不太一样，需要对小点数进行特殊处理，这也就有了下面将要介绍的“定点数”与“浮点数”之分。

### 1.5.1 定点数

整数类型的机器数很好表示，只是在最高位多了一个符号位，小数点完全可以忽略，就像我们在书写十进制整数时一样。但如果是小数，小数点是不能不写的，在计算机中也不是像我们平常书写时那样好处理了，因为计算机存储器是没有专门的位来表示小数点的。

为了能使计算机自动识别机器数的小数点位置，正确识别对应机器数的大小，依据机器数中小数点的位置是否固定把整个机器数分为“定点数”（Fixed Point Number）和“浮点数”（Floating Point Number）两大类。本节先介绍定点数，下节再介绍浮点数。

从名称上很容易知道，定点数是小数点固定在某一个位上的机器数。它又可分为“定点整数”和“定点小数”，就像十进制数中包括了纯整数和纯小数一样，当然在一个机器数中也可以同时包括整数和小数。

1. 定点整数

“定点整数”是小数点固定在有效数值部分最低位之后的定点数，是纯整数。如一个定点整数是 n 位，即 $X=X_0X_1X_2\ldots X_{n-1}$，最高位 $X_0$ 为符号位，则它所能表示的数值范围为 $-(2^{n-1}-1)\leqslant X\leqslant 2^{n-1}-1$。例如一个 8 位的定点整数，除了符号位外，用于表示真值的共 7 位，由此可以得出它能表示的最大值是 01111111，即 127，等于 $2^7-1$，所能表示的最小值为 11111111，即-127，等于 $-(2^7-1)$。

2. 定点小数

“定点小数”是小数点位置固定在符号位之后、有效数最高位之前的定点数，是纯小数。如一个定点小数是 n 位，即 $X=X_0X_1X_2\ldots X_{n-1}$，最高位 $X_0$ 同样为符号位，则它所能表示的数值范围为 $-(1-2^{-(n-1)})\leqslant X\leqslant 1-2^{-(n-1)}$。例如一个 8 位的定点小数，除了符号位外，用于表示真值的共 7 位，由此可以得出它能表示的最大值是 0.1111111，即 0.5+0.25+0.125+0.0625+0.03125+0.015625+0.0078125=0.9921875，等于 $1-2^{-7}$，所能表示的最小值为 1.1111111，即-0.9921875，等于 $-(1-2^{-7})$。


根据以上给出的定点整数和定点小数所能表示的数值范围不等式可以得出以下规律:

- 一个 n 位连续为 1 的正二进制整数( 即 n 位正二进制整数所表示的最大数 )的值是 $2^n-1$。例如在 IPv4 地址中，每个字节是 8 位，它的最大值（即每位为 1）就是 $2^8-1=255$。同理，一个 n 位连续为 1 的负二进制整数（即 n 位负二进制整数所表示的最小数）的值是 $-(2^n-1)$。
- 一个 n 位连续为 1 的正二进制小数（即 n 位正二进制小数所表示的最大数）的值是 $1-2^{-n}$，也就是不要把每位转换成十进制再相加了，只需要用 1 来减 2 的-n 次方（此时只需要计算一个二进制小数位的值）。同理，一个 n 位连续为 1 的负二进制小数（即 n 位负二进制小数所表示的最小数）的值是 $-(1-2^{-n})$。

3. 同时包括整数和小数部分的定点数

以上的“定点整数”是纯整数，“定点小数”是纯小数，它们的小数点很好判定，运算起来也很方便。而在实际运算中一般不太可能全是纯整数和纯小数的数，也就是大多数情况下是同时包括了整数和小数的定点数。

对于同时包括了整数和小数的定点数，为了能确定其小数点的位置，在一个字长的存储位中小数点位置必须事先确定下来。这也就固定了一个字长所能表示的机器数的大小精度了。如一个 8 位字长，除了符号位还有 7 位用来表示真值，如果规定小数点是在第 3 位，则用于表示机器数真值的 7 位中，最高 4 位用来表示整数，最低 3 位用来表示小数。这样一来，真值整数部分最大值的绝对值部分就是 $2^4$-1=15，真值的精度就是保留 3 位小数。

很显然，定点数表示法的缺点在于其形式过于僵硬，固定的小数点位置决定了固定位数的整数部分和小数部分，不利于同时表达特别大的数或特别小的数，所以在实际的计算机运算中是很少采用这种格式的，而是采用我们下面将要介绍的“浮点数”。

### 1.5.2 浮点数

“浮点数”是小数点位置可变的机器数，采用了类似十进制数中的科学记数（Scientific Notation）法的表达方式。如十进制数中，123.45 用科学记数法可以表达为 $1.2345\times10^2$，其中 1.2345 为尾数，10 为基数，2 为指数，尾数和指数均为十进制数。如果是负数的话，前面还有一个负号（-）。

如果把十进制数用 N 表示，尾数用 M 表示，指数用 E 表示，十进制数的科学记数法的表达公式就是：$N=\pm M\times10^E$。

按照这个思路，把以上的十进制数换成二进制机器数，则可得出二进制数的类似科学记数法表达公式为：$N=\pm M\times2^E$。其中，基数由原来的 10 换成 2，尾数 M 只能是二进制数（原码格式），指数 E 是对应十进制数的二进制数（也是原码格式，称为“阶码”，在下面将要介绍的 IEEE 754 浮点数中也可用“移码”表示）。浮点数在计算机存储器中存储的格式可以是原码或补码格式，通常是采用补码格式，因为补码格式对于负数也是无异义的。

以上的浮点数表示形式也就决定了浮点数在存储器中的存储格式，因为它的小数点位置是可变的，所以肯定不能直接把浮点数的二进制表示式在存储器中存储，否则就无法知道这个数到底多大了。

从前面得出的浮点数表示形式可以看出，一个浮点数只要知道它的符号位、指数和尾数，就可以最终根据以上公式得出它的实际大小。所以浮点数在存储器中就是把以上包括 1 个符号位（用 S 表示）、指数 E 和尾数 M 这三个部分进行存储，如图 1-13 所示。当然关于 E 和 M 具体各占多少位要视不同的浮点数定义，下节将介绍 IEEE 754 标准下的浮点数定义。


图 1-13 浮点机器数的存储格式

E 的二进制表示形式为定点整数，表示浮点数的小数点位置，决定了浮点数的表示范围；M 为定点小数，表示浮点数的有效值，决定了浮点数的精度。指数 E 为正时，表示转换后尾数部分

的小数点与原数的小数点位置相比左移了 E 位（二进制格式，计算位数时要转换为十进制），指数为负时，表示转换后尾数部分的小数点与原数的小数点位置相比右移了 E 位。

如一个浮点数为$-0.10101\times2^{11}$，则它在 16 位字长的存储器（假设规定 E 用 4 位表示，M 用 11 位表示，M 符号位占 1 位）中的存储格式如图 1-14 所示。

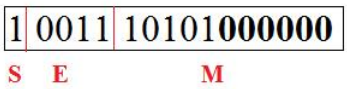



图 1-14　浮点数在存储器中的存储结构示例

由以上的分析可知，浮点数是一个纯小数和一个指数的乘积。但我们知道在十进制数的科学记数法中，同样的数值可以有多种浮点数表达方式，比如上面例子中的 123.45 可以表达为 $12.345\times10$、$1.2345\times10^2$ 或 $0.12345\times10^3$。在浮点数表示中也一样，尾数中的小数点位置也是可随意的。

为了规范起见，在机器数的浮点数表达形式中规定尾数的最高有效位（即小数点右边第一位）不能为 0，必须为 1。把不满足这一要求的尾数变成满足要求的尾数的操作过程称为浮点数的规格化处理，可通过尾数小数点的移位和修改“阶码”来实现。



### 1.5.3　IEEE 754 浮点数的分类

浮点数的定义有许多，目前通常是采用 IEEE 754 国际标准中的定义，把浮点数分成了单精度（Single）浮点数和双精度（Double）浮点数两种。在这个标准中规定，单精度浮点数长度为 32 位（4 字节），双精度浮点数长度为 64 位（8 字节）。

IEEE 754 标准中的以上两种浮点数的表示方式与前面介绍的浮点数表示方式在结构上是一样的，只不过在 IEEE 754 中把指数部分用“移码”来表示，而不是用以前的“阶码”，而且 E 和 M 的表示形式一般是要求用补码，而不是用原码，因为补码的符号位都参与运算，计算结果都没有错误。

前面我们知道，浮点数的指数 E 用阶码表示时是可正、可负的，这样就会使得一个浮点数中出现两个符号位：浮点数自身的和浮点数指数部分的。这样的结果是，在比较两个浮点数大小时显然比较麻烦。“移码”的定义在 1.4.5 节中介绍了，是在阶码基础上加一个固定的正常数得到，这个固定的正常数在 IEEE 754 中称为“偏移”。之所以要用移码表示，目的就是为了使指数部分均为无符号的正整数，这样补码与原码是一样的，不用转换。

在 IEEE 754 中，浮点数在存储器中存储的格式以及所指定的偏移值如表 1-4 所示。

表 1-4　单精度浮点数和双精度浮点数各部分在存储器中的结构

| 浮点数类型 | 符号所在位（S） | 指数所在位（E） | 尾数所在位（M） | 偏移值 |
|---|---|---|---|---|
| 单精度浮点数 | 31(1) | 30-23(8) | 22-0(23) | 127（对应二进制为：01111111） |
| 双精度浮点数 | 63(1) | 62-52(11) | 51-0(52) | 1023（对应二进制为：0111111111） |

此时表中的 E 是以“移码”格式表示的，在 IEEE 754 中用于表示浮点数中小数点位置的“移码”=阶码+偏移值。如果用 e 来表示原来的阶码，则移码与阶码的关系为：

E=e+偏移值

在计算浮点数时可用以下公式统一表示：$X=S(1.M)\times 2^{E-偏移}$，X 为浮点数原码，S 为符号位，M 为尾数，E 为移码，“E-偏移”就等于阶码。

### 1.5.4 IEEE 754 浮点数的计算

给定一个机器数，要计算出其 IEEE 754 标准浮点数格式，可直接按上节介绍的公式：$S(1.M)\times 2^{E-偏移}$进行，关键的计算步骤如下：

（1）把要转换的数转换成二进制。

十进制数转换成二进制数要区分整数部分和小数部分，因为它们的转换方法是不一样的，具体转换方法参见 1.2.2 节。

（2）把二进制数转换成 $1.xxxx\times 2^e$ 格式。

这里的 e 就是“阶码”（此时用十进制数表示）。当 e 大于 0 时，表示转换成 $1.xxxx\times 2^e$ 格式后小数点相对原来的小数点位置左移了 e 位；当 e 小于 0 时，表示转换成 $1.xxxx\times 2^e$ 格式后小数点相对原来的小数点位置右移了 e 位。

（3）由原来数的正负属性得出 S 是 0（正数时）还是 1（负数时），并由上步得出的 e 值计算出 E（E=e+偏移），E 为移码。

（4）在 1.xxxx 的小数点右边第一位起插入 E 的二进制形式，同时补足尾数（即 xxxx 部分）对应精度位数，即得出浮点数的二进制表示形式。

**【示例 1】**求十进制数 0.5 的单精度浮点数。

（1）按照 1.2.2 节介绍的方法计算出十进制数 0.5 的二进制表示形式：0.1。

（2）转换为 $1.xxxx\times 2^e$ 格式：$1.0\times 2^{-1}$，即小数点相对原来右移了 1 位，阶码 e 为-1。

（3）根据“E= e+偏移”（E 为移码）得出 E=-1+127=126（单精度的偏移值为 127），转换成二进制后为 01111110。又因为原数是正数，所以 S=0。

（4）在以上 1.0 的小数点右边第一位后面插入 E，然后补足尾数对应精度位数（原来的尾数只有一位：0，单精度的尾数为 23 位）。

最终得到十进制数 0.5 所对应的单精度浮点数为：0 01111110 00000000000000000000000，具体格式如图 1-15 所示。

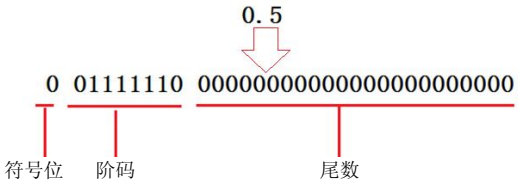



图 1-15 十进制数 0.5 的单精度浮点数格式

【示例 2】求十进制数-1.75 的单精度浮点数。

（1）按照 1.2.2 节介绍的方法计算出十进制数 1.75 的二进制表示形式：1.11。

（2）转换为 1.xxxx×$2^e$ 格式：1.11×$2^0$，故得出 e 为 0，因为小数点没有移位。

（3）根据“E=e+偏移”得出 E=0+127=127，转换成二进制后为 01111111。单精度的偏移值为 127。又因为原数是负数，所以 S=1。

（4）在以上 1.11 的小数点后面第一位起插入 E，然后补足尾数对应精度位数（原来的尾数只有两位：11，单精度的尾数为 23 位），即需要在后面加 21 个 0。

最终得到十进制数-1.75 所对应的单精度浮点数为：1 01111111 11000000000000000000000，具体格式如图 1-16 所示。但此时尾数是原码格式，又因为尾数是负数（移码总是正数，其补码与原码格式一样），还需要转换成补码格式，最终得到该数的补码形式单精度格式为：1 01111111 01000000000000000000000。

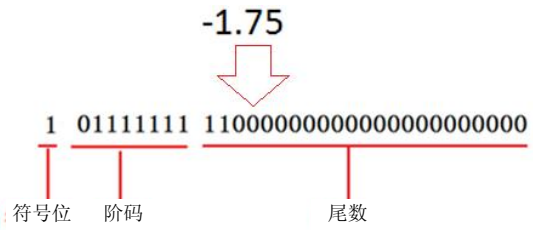



图 1-16　十进制数-1.75 的单精度浮点数格式

### 1.5.5　浮点数表示形式的转换

通过前面的学习已经知道，浮点数的表示形式还不是固定的，指数既可以用阶码表示，又可以用移码表示，尾数可以是原码形式，也可以是补码形式。本节就来举例说明这些不同表示形式之间的转换方法。

【示例】如有两个浮点数 X=$2^{010}$×0.11011011，Y=$2^{100}$×(-0.10101100)，当前指数采用的是二进制阶码格式，尾数采用原码格式。如果要把这两个浮点数转换成指数用 4 位移码，尾数用 9 位（含符号位）补码形式表示，该如何转换呢？

这样的题就有三项工作要做：一是把阶码转换成移码，这个在 1.4.5 节有介绍，并且给出了一个公式，即 $X_{移}=X_{阶}+2^n$（n 为原阶码的真值位数）；二是把原码形式的尾数转换成补码形式；三是确定浮点数的符号位。具体步骤如下：

（1）把阶码转换成移码。

把 X 的阶码$(010)_2$转换成 4 位移码的方法是$(010)_2+(2^2)_2$（阶码中只有 2 位是真值，也要转换成二进制形式）=$(010)_2+(1000)_2=(1010)_2$。

用同样的方法可以把 Y 的阶码$(100)_2$转换成 4 位移码，等于$(100)_2+(2^2)_2=(100)_2+(1000)_2=(1100)_2$。

（2）求尾数的补码形式。

先要得到尾数的原码形式。从两数的表示形式可以得出 X 的尾数（M）为 11011011，Y 的尾数（M）为 10101100，且题中已说明已是原码形式。

因为 X 为正数，所以符号位（S）为 0，而 Y 为负数，所以符号位（S）为 1。通过前面的学习已经知道，浮点数的格式为：S E M，S 为符号位，E 为阶码或移码，M 为尾数。此时如果浮点数中指数用移码表示，尾数 M 用原码表示，则 X 和 Y 的格式分别为：$X_{浮}$=0 1010 11011011，$Y_{浮}$=1 1100 10101100。

此时如果要把尾数 M 用 9 位补码表示，则需要将 X 和 Y 尾数部分转换成补码形式。

因为 X 是正数，所以它的尾数 M 的补码与原码一样，等于 11011011（加上符号位正好为 9 位）。Y 是负数，它的尾数 M 原码转换成补码时先要对除符号位外的其他各位取反（取反后得出低 8 位为 01010011），然后再加 1，得到 01010100（加上符号位也正好为 9 位）。

（3）得出最终的补码表示形式。

通过以上步骤以及浮点数的符号位可以最终得出尾数 M 用补码形式表示的 X、Y 浮点数的存储格式为（正数 X 的补码格式与原码格式是完全一样的）：$X_{浮}$=0 1010 11011011，$Y_{浮}$=1 1100 01010100。



## 1.6 二进制数的运算

本节所介绍的二进数运算主要包括无符号二进制数四则算术运算、机器数补码形式的加/减法运算和无符号二进制数的逻辑运算。比较复杂的浮点数运算将在本章后面介绍。其实无符号二进制数的算术和逻辑运算方法同样适用于补码形式二进制数、浮点数中真值部分二进制数的对应运算。

**【说明】**通过前面的学习我们已经知道，因为补码形式的二进制数符号位也是直接参与运算的，不需要单独处理，所以在计算机中都是以补码形式来进行二进制数运算的。也正因为如此，在进行各种二进制运算时都建议把对应的二进制数转换为补码形式。

### 1.6.1 二进制数的算术运算

无符号二进制数的加、减、乘、除四则算术运算法则其实与十进制数的四则算术运算法则是一一对应的。你理解了十进制数的四则算术运算法则，无符号二进制数的四则算术运算就一点都不难了。

1. 加法运算

加法运算法则：0+0=0，0+1=1，1+0=1，1+1=10。

即当两个相加的二进制位仅一位为 1 时，相加的结果为 1；如果两个二进制位全是 0，相加的结果仍为 0；如果两个相加的二进制位均为 1，则结果为 10（相当于十进制中的 2），要向高位进 1，也就是“逢 2 进 1”规则，与十进制中的“逢 10 进 1”道理一样。

【说明】在进行二进制加减法运算时，最关键的一点就是逢 2 进 1，进 1 当 1，而借 1 当 2。大家联想一下我们经常使用的十进制数加法运算法则，那就是逢 10 进 1，进 1 当 1，而借 1 当 10，这样一来我们就好理解了，无符号二进制数的加法运算法则只是把原来十进制数加法运算法则中的 10 改成了 2。

【示例 1】计算$(10010)_2$+$(11010)_2$的过程如图 1-17 所示（注意两数要从最低位开始对齐）。

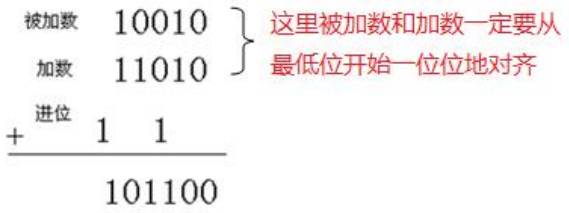



图 1-17　二进制加法运算示例

（1）进行最低位相加，这里加数和被加数都为“0”，根据加法原则可以知道，相加后为“0”。

（2）进行倒数第二位相加，这里加数和被加数都为“1”，根据加法原则可以知道，相加后为“$(10)_2$”，此时把后面的“0”留下，而把第一位的“1”向高一位进“1”。

（3）进行倒数第三位相加，这里加数和被加数都为“0”，根据加法原则可以知道，本来结果应为“0”，但倒数第二位已向这位进“1”了，此时就要同时把“被加数”“加数”和“进位”这三个数加起来，所以结果应为 0+0+1=1。

（4）进行倒数第四位相加，这里加数和被加数分别为“1”和“0”，倒数第三位也没有进位，根据加法原则可以知道，相加后为“1”。

（5）最高位相加，这里加数和被加数都为“1”，根据加法原则可以知道，相加后为“$(10)_2$”。同样需要把第一位的“0”留下并向高位进 1，这样会产生新的最高位，值为“1”（但如果超出了字长的限制，则新产生的最高位将溢出）。

这样$(10010)_2$+$(11010)_2$的最后运算结果为 101100。

2. 减法运算法则

减法运算法则：1-1=0，1-0=1，0-0=0，0-1=-1。

即当两个相减的二进制位中同为 0 或 1 时，相减的结果为 0；如果被减数的二进制位为 1，而减数的二进制位为 0，则相减的结果仍为 1；如果被减数的二进制位为 0，而减数的二进制位为 1，则需要向高位借 1，但此时是借 1 当 2，与十进制中的借 1 当 10 道理一样。

【示例 2】计算$(111010)_2$-$(101011)_2$的过程如图 1-18 所示（注意两数要从最低位开始对齐）。

（1）最低位相减，这里被减数为“0”，减数为“1”，不能直接相减，需要向高位（此时为倒数第二位）借“1”，这样相当于得到了十进制中的“2”，用 2 减 1 结果就得到 1。

（2）对倒数第二位相减，此时本来被减数和减数均为“1”，但是被减数的该位被上一步借走了 1，所以最后就变为“0”（1-1）了。此时也不能直接与减数相减了，又需要向高位（此时为倒

数第三位)借 1。同样是借 1 当 2，相当于该位总共为 0+2=2。这样倒数第二位相减后的结果为 2-1=1。

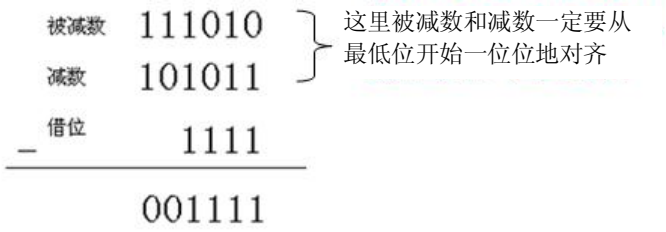



图 1-18　二进制减法运算示例

（3）用上一步同样的方法计算倒数第三位和倒数第四位的减法运算，结果都为 1。

（4）计算倒数第五位的减法运算，此时被减数原来为“1”，可是已被倒数第四位借走了 1，所以成了“0”（1-1），而此时减数也为“0”，可以直接相减，得到的结果为“0”。

（5）计算最高位的相减，被减数和减数均为“1”，可以直接相减，得到的结果为“0”。

这样一来，$(111010)_2$-$(101011)_2$ 的结果是$(001111)_2$，由于整数的前导 0 可以不写，所以最后结果就是$(1111)_2$。

3. 乘法运算法则

乘法运算法则：0×0=0，0×1=0，1×0=0，1×1=1。

即只有当两个相乘的二进制数都为 1 时，相乘的结果才为 1；两个相乘的二进制数中只要有一位为 0（也包括是两位同时为 0），则相乘的结果都为 0。也可以这么理解：1 与任何数相乘的结果就是对应的被乘数；而 0 与任何数相乘的结果都为 0。这与十进制中的乘法运算法则也是一样的。

**【说明】**在乘法运算中，乘数的每一位要与被乘数的每一位分别相乘，而不仅是对应位相乘，而且每一位乘数与被乘数的每一位相乘的结果的最低位要与对应的被乘数位一样。当然这与十进制的乘法运算法则也是一样的，很好理解。

**【示例 3】**计算$(1010)_2$×$(101)_2$ 的过程如图 1-19 所示（注意两数要从最低位开始对齐）。

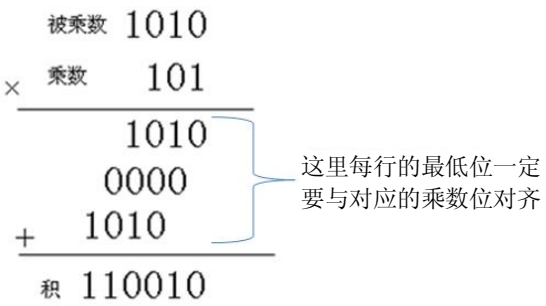



图 1-19　二进制乘法运算示例

（1）乘数的最低位与被乘数的所有位相乘，因为乘数的最低位为“1”，根据乘法原则可以得

出，结果其实就是被乘数本身，直接复制下来即可。此处结果就为 1010。

（2）乘数的倒数第二位与被乘数的所有位相乘，因为乘数的这一位为“0”，根据乘法运算法则可以得出，结果均为“0”。此处结果就为 0000。

（3）乘数的最高位与被乘数的所有位相乘，此时乘数为“1”，结果就是被乘数本身。此处结果就为 1010。

（4）按照前面介绍的二进制加法原则对以上三步所得的结果按位相加（但这里的位对齐方式与单纯的二进制数相加不一样，最低位一定要与对应被乘数位一致。这也与十进制的乘法运算方法一样），结果得到$(110010)_2$。

4. 除法运算法则

当被除数大于除数时，商是“1”；当被除数小于除数时，不够除，商只能是“0”，这与十进制的除法也类似。二进制只有两个数（0 和 1），因此它的商也只能是 1 或 0。

**【示例 4】**计算$(11001)_2 \div (101)_2$的过程如图 1-20 所示。

```
        商   101
除数  101 ) 11001  被除数
          - 101
            101
          -  101
          -------
             000
```

图 1-20　二进制除法运算示例

（1）因为除数为“101”，有 3 位，所以在被除数中也至少要取 3 位（从最高位开始取）。被除数的高 3 位为“110”，恰好比除数“101”大，可以直接相除，但商只能是 1（因为二进制的最大数元就是 1），然后把被除数与用商“1”与除数相乘后的结果进行相减，得到的值为“1”。

（2）从被除数中取下一位“0”下来，与上一步的差“1”值组成新的被除数，为“10”，显然它比除数“101”小，不够除。于是在商的对应位置上输入“0”。

（3）继续从被除数中取下一位“1”下来，与上一步的余数“10”值组成新的被除数，为“101”，此数正好与除数“101”相等，所以此时的商取“1”正好除尽。

这样一来$(11001)_2 \div (101)_2$所得的商就是$(101)_2$。

### 1.6.2　补码的加/减法运算

通过前面的学习我们已经知道，机器数的补码可由原码和反码得到。如果机器数是正数，则该机器数的补码与原码、反码一样；如果机器数是负数，则该机器数的补码是对它的反码在末位加 1 而得到的。

1. 补码的加法运算

补码的加法运算法则：$[X+Y]_{补}= [X]_{补}+[Y]_{补}$。

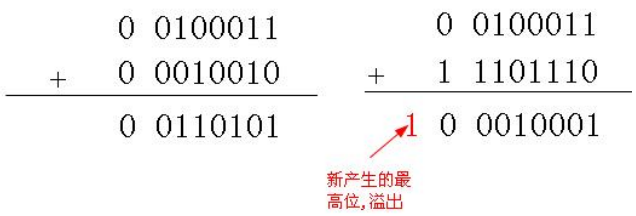
该式表明，两个机器数相加的补码可以通过先分别对两个机器数求补码，然后相加得到。在采用补码形式表示时，进行加法运算可以把符号位和数值位一起进行运算（若符号位有进位，则溢出不管），结果为两数之和的补码形式。

**【示例 1】**如要求两个十进制数：35+18 的补码（假设字长为 8）。根据上面的补码加法运算法则可以得知，只需分别求 35 和 18 这两个数的补码，然后相加即可。又因为这两个数都是正数，所以它们的补码与原码一样。

这样一来，这道题实际上也就是求 35 和 18 这两个十进制数的原码和。35 的原码为 0 0100011（注意：最高位为符号位），18 的补码为 0 0010010。所以 35+18 的补码就等于(0 0100011)B+(0 0010010)B=(00110101)B，如图 1-21（a）所示。如果转换成十进制则等于 53，结果正确。如果相加后有超过字长的位溢出，则直接丢弃。

**【示例 2】**如果要求两个十进制数：35+(-18)和的补码也是直接求正数 35 和负数(-18)的补码和。正数 35 的补码与其原码一样，前面已计算出，为 0 0100011；而后面那个“-18”因为是负数，不能直接从它的原码得到补码了。需要先就要求（-18）的原码（1 0010010），然后对其除符号位外的其他各位取反，得到其反码（为 1 1101101），最后再在其末位（最低位）加 1，最终得到其补码为 1 1101110。

这样一来，“35-18”的补码就是(00100011)B+(11101110)B，结果为 00010001，如图 1-21（b）所示。这里要注意，两个补码相加后产生了第 9 位（为 1）的溢出，直接丢弃，所以结果就是(00010001)B。如果转换成十进制则等于 17，结果正确。




（a） （b）

图 1-21 两个补码加法运算示例

2. 补码的减法运算

补码的减法运算法则：$[X-Y]_{补}=[X]_{补}+[-Y]_{补}$。

该公式表明，求两个机器数的差值（如$[X-Y]_{补}$）的补码可以通过求被减数的补码（如$[X]_{补}$）与减数的负值的补码（$[-Y]_{补}$）的和得到。

负数补码的求解方法我们在前面已有介绍，那就是先对除符位号外的其他各位取反，最后加 1。其实还有一个更简单的方法，那就是负数的补码等于该正数的补码（也等于该正数的原码）全部位（含符号位）取反，然后再加 1（因为正数与负数的符号也正好相反）。如已知$[15]_{补}=00001111$，则$[-15]_{补}=11110000+1=11110001$。

**【示例 3】**现假设 X=+35，Y=+18，要求$[X-Y]_{补}$（假设字长为 8）。

先根据正数的补码与原码一样的规则，求得$[X]_{补}$= 00100011，$[Y]_{补}$ = 00010010；再根据以上介绍的负数求补码操作规则（全部位取反，再加 1），可以得到$[-Y]_{补}$=11101110。

最后用$[X]_{补}$+$[-Y]_{补}$公式即可得到最终的$[X-Y]_{补}$ = 00010001，如图 1-22（a）所示。转换成十进制也可得到结果为 17，正确，且与上面使用加法法则运算的结果一样。注意，这里相加的结果也产生了溢出的第 9 位（1），直接丢弃。

**【示例 4】**如有两个十进制数 X=-35，Y=-18，现要求$[X+Y]_{补}$的值。

根据前面介绍的加法法则得知，可先求得$[X]_{补}$和$[Y]_{补}$的值，然后再相加，即$[X]_{补}$+$[Y]_{补}$。根据本章前面介绍的求负数补码的方法来计算-35 和-18 的补码。

上一示例中已求出了 35 和 18 的补码（也就是它们的原码）分别为 00100011 和 00010010，然后分别对它们各位取反，再加 1，由此可得出$[-35]_{补}$为 1 1011101，$[-18]_{补}$=11101110。

最后$[-35]_{补}$+$[-18]_{补}$的运算过程如图 1-22（b）所示，结果为 11001011。注意这是一个有符号位数，所以结果为-75。这样看起来结果是不正确的，因为-35-18 应该等于-53。这时就要特别注意了，这个-75 是一个补码形式，要看最终的结果还得把它转换成原码。按照上面介绍的方法立即可以得到$[-75]_{补}$的原码为 10110101（注意它也是一个有符号位数，最高位的 1 代表为负数）=$[-53]_{原}$。这样结果就正确了。

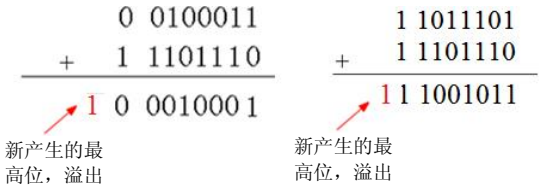



图 1-22　两个补码减法运算示例

### 1.6.3　二进制数的逻辑运算

逻辑运算是指对因果关系进行分析的一种运算，这也是在计算机中经常采用的一种二进制运算。如在通过 IPv4 地址和子网掩码得出其网络地址时就要用到逻辑“与”运算（具体将在本书后面相应章节介绍）。逻辑运算的结果并不表示数值大小，而是表示一种逻辑概念，若成立则为“真”，或用“1”表示；若不成立，则为“假”，或用“0”表示。

二进制数的逻辑运算主要有“与”“或”“非”和“异或”4 种。

1. “与”运算（AND）

“与”运算又称逻辑乘，用符号“.”或“∧”来表示。运算法则如下：

0∧0 = 0　　0∧1 = 0　　1∧0 = 0　　1∧1 = 1

即只要两个参加“与”运算的数的对应位的数中有一个为 0，则运算结果均为 0；仅当两数的

对应位均为 1 时结果才为 1，很容易判断。这与前面介绍的二进制乘法运算是一样的。

**【示例 1】**图 1-23 所示是两个“与”的逻辑运算示例。左图所示的是两个位数不一样的二进制数进行“与”运算，这时要求两个数从最低位开始对齐，在位数少的二进制数的最高位前面加上“0”补齐，使得它与位数多的二进制数有一样的位数。

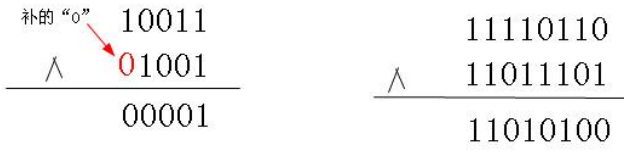



图 1-23　两个“与”逻辑运算示例

2. “或”运算（OR）

“或”运算又称逻辑加，用符号“+”或“∨”表示。运算法则如下：

0∨0＝0　　0∨1＝1　　1∨0＝1　　1∨1＝1

即只要两个参加“或”运算的数的对应位的数中有一个为 1，则运算结果为 1，仅当两数的对应位均为 0 时结果才为 0，也很容易判断。

**【示例 2】**如图 1-24 所示是两个“或”逻辑运算的示例。同样，进行“或”运算时要求两数从最低位开始对齐，位数少的数在最高位前面加“0”补齐，最终使两个二进制数的位数相同。

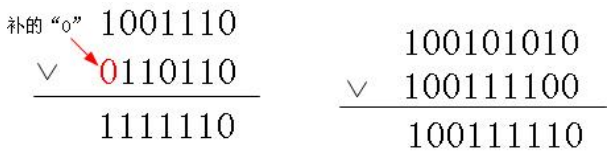



图 1-24　两个“或”逻辑运算示例

3. “非”运算（NOT）

“非”运算就是逐位求反的运算，其运算法则为：“0”的反值为“1”，“1”的反值为“0”，也就是“0”与“1”互为反。注意：“非运算”只是针对一个二进制数进行的，这与前面的“与”和“或”运算不一样。如“101110101”进行“非”运算后就得到“010001010”（可简写为“10001010”）。

4. “异或”运算（XOR）

“异或”运算用符号“⊕”来表示。运算法则如下：

0 ⊕ 0＝0　　0 ⊕ 1＝1　　1 ⊕ 0＝1　　1 ⊕ 1＝0

即当两个参加“异或”运算的二进制数对应位相同时运算结果为 0，不同时运算结果为 1。

**【示例 3】**图 1-25 所示是两个“异或“逻辑运算的示例。同样，进行“异或”运算时要求两数从最低位开始对齐，位数少的数在最高位前面加“0”补齐，最终使两个二进制数的位数相同。

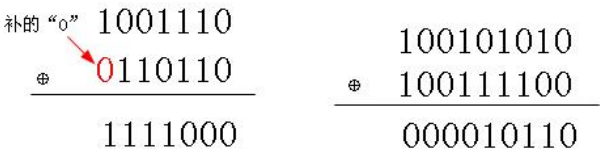



图 1-25　两个“异或”逻辑运算示例

## 1.7　浮点数的加/减法运算

浮点数与定点数相比有两个明显的特点：一是小数点位置不固定，而我们知道在做加减运算时，两数的小数点位置必须对齐；二是在存储器中不是直接把浮点数本身直接存储，而是在中间插入了一个用于表示小数点位置的阶码或移码，所以不能直接把在存储器中存储的浮点数相加减，只需对代表浮点数有效数部分的尾数进行加减运算。

正因为如此，浮点数的加减法运算相比定点数的运算就要复杂许多，当然从事网络职业的人士也没必要做太深的了解（从事程序开发的人士是必须要掌握的）。有关浮点数的加减法运算包括以下 5 个基本步骤：对阶、尾数运算、规格化处理、舍入处理、溢出处理。



### 1.7.1　对阶

所谓“对阶”就是比较参与运算的浮点数的阶码大小，然后使它们的阶码（或移码）都一样，其最终目的是使参与运算的数的小数点位置对齐，以便正确地计算出这些浮点数之间的加减法运算结果。

既然是要使参与运算的各浮点数的阶码一样，这时就要以某一个浮点数的阶码为标准了。通常是以大阶码为准，即阶码小的数要向阶码大的数对齐，即把小阶码的值调成与大阶码的值一样。这时为了确保小阶码的数最终值与原来保持不变，肯定要对小阶码的浮点数的尾数的小数点进行移位。就像十进制数中的 $1.2345\times10^3$，现在把指数由原来的 3 改为 5，则原来的尾数 1.2345 中的小数点肯定不能还保持不变，要向左移两位（因为指数增加 2），变成 0.012345，最终的形式为 $0.012345\times10^5$。

因为是以大阶码为准，相当于小阶码的数的阶码要增大，所以其尾数肯定要做相同倍数的减小（这样才能保持原数值不变），即阶码小的尾数要向右移（小数点是向左移）对应的位数。移位原则是：在尾数最高位前面加对应位数的 0（小了多少位就加多少个 0），然后原尾数最后的 n 位被丢弃（移了多少位就丢弃最后面多少位），因为用于存储尾数部分的存储器位数是固定的。


至于对阶中的尾数移位可以借助我们常用的十进制数科学记数法来帮助理解。假设十进制数 1234.5 原来表示成 $1.2345\times10^3$，现在要表示成$\times10^6$的格式，则对应的格式为 $0.0012345\times10^6$格式（在

原来的小数点右边最高位前面移了 3 位，补加了 2 个 0）。又假设小数点后面最多只能 4 位（就相当于浮点数中用来保存尾数的位置是固定的），这样一来，最终的存储格式就为 $0.0012\times10^6$，原来最后面的“345”就丢弃了。当然这样会与原数在大小上有一些区别（因为最终成了 1200），但仍能最大限度地保持不变。

**【示例】**如 $X=2^{0010}\times0.11000101$，$Y=2^{0100}\times0.10101110$，现在对这两个浮点数进行加法或减法运算，首先就要对这两个数进行对阶。阶码和尾数均已采用补码表示形式（尾数的符号位体现在浮点数的符号位上）。

X 的阶码为 0010，对应的十进制数为 2，Y 的阶码为 0100，对应的十进制数为 4，由此可知 X 的阶码比 Y 的阶码小 2。此时把 X 的阶码也变为与 Y 的阶码一样，即都调成 4，同时把 X 的尾数小数点左移 2 位（在小数点右边最高位前面加两个 0，原来最低的 2 位被丢弃），这样才能使 X 的值与原来保持基本不变。

X 原来的尾数为 11000101（符号位体现在浮点数的符号位上，此数为正），向右移两位（仍要保持原来的总位数不变，则要在前面补相应位数的 0，原来最右边的对应两位要丢弃），则变成 00110001。这样一来，X 的阶码也变成 0100，最终 X 在存储器中的位结构变成了 0 0100 **00110001**（原来为 0 0010 **11000101**）。



### 1.7.2 尾数运算

参与运算的浮点数的小数点位置对齐后，就可以直接把经过移位的尾数进行相加或相减运算。这一步很简单，可以直接按照 1.6.1 节介绍的方法进行加减法。

**【示例】**继续前面的示例，即 $X=2^{0010}\times0.11000101$，$Y=2^{0100}\times0.10101110$，现要求 X+Y。

在上节介绍的“对阶”步骤中我们已把 X、Y 的阶码以 Y 的阶码为标准进行了统一，也求出了对阶后 X 的尾数为 00110001，而 Y 的尾数不变，为 10101110，把两者直接相加，得到的结果为 11011111，如图 1-26 所示。

$$\begin{array}{r} 00110001 \\ +\ 10101110 \\ \hline 11011111 \end{array}$$

图 1-26　X 和 Y 的尾数相加

### 1.7.3 规格化处理

进行加减法运算后的尾数可能不是规格化的数，就需要进行规格化处理。规格化的尾数格式要求如下：

- 尾数采用原码表示形式时：正数的规格化格式为：0.1xxxx，负数的规格化格式为：1.1xxxx。
- 尾数采用补码表示形式时：正数的规格化格式为：0.1xxxx，负数的规格化格式为：1.0xxxx。

以上最高位均代表符号位（0 代表正数，1 代表负数），关键是看小数点后面的数的格式。

- 对于双符号位（用两位来表示符号）的补码形式尾数，正数的规格化格式为 00.1xxxx，负数的规格化格式为：11.0xxxx，以上最高两位代表符号位（00 代表正数，11 代表负数），关键是看小数点后面的数的格式。

【说明】引入“双符号位”的设计的目的就是为了能快速检测出运算结果是否有溢出，因为双符号位规定了“00”代表正数，“11”代表负数，如果最高两位不是两种形式，而是“01”或者“10”就能快速地知道有溢出了。符号位为 01 时，称为上溢，即最高真值位相加后有进 1；为 10 时，称为下溢，即最高真值位相减后有借 1。在此我们仅介绍单符号位形式。

凡不符合以上格式要求的尾数均要进行规格化处理。对以上规格化格式进行总结可以得出：符号位与尾数最高位不一致才算是规格化，一致为非规格化。如 1.0xxxx、0.1xxxx 之类的尾数（最前面的为符号位）就是规格化的数，而 1.1xxxx，0.0xxxx 之类的数则是非规格化的数。另外，如果尾数大于 1，也是非规格化数。

对于非规格化的尾数需要进行相应的处理，这些处理方式又分“左规”和“右规”。所谓“左规”就是尾数要向左移位，每移 1 位阶码值减 1，直到为规格化数为止；对应“右规”就是尾数要向右移位，每移 1 位阶码值加 1，直到为规格化数为止。

具体什么情况下要采用左规，什么情况下要采用右规，基本原则如下：

- 运算结果产生溢出（由于原来两尾数的最高有效位相加有进位或者相减有借位时形成的）时，必须进行右规。如双符号位情形下的运算结果为 10.xxxx 或 01.xxxx 格式就是不符合规格化要求了，因为双符号位时符号位为“10”和“01”都是不正确的。右规时最高有效位前补相应位数的 0。此时 10.xxxx 格式右规后的格式为 11.0xxx，而 01.xxxx 格式右规后的格式为 00.1xxx。
- 如运算结果出现 0.0xxx 或 1.1xx（即符号位与尾数最高有效位相同）时，必须进行左规。左规时最低有效位后补相应位数的 0。

在前面的例子中已得出 X 和 Y 的尾数之和为 11011111，又因为它是正数，所以可表示成 0.11011111，已是规格化，不用再进行处理了。

假设某 X 与 Y 浮点数相减，并且得出的尾数之差为负数，如 1.11011111，这时就要进行规格化处理了，因为符号位 1 与尾数的最高位 1 相同。这时要对尾数左规，即向左移位。11011111 要向左移 2 位才可能使尾数的最高有效位与前面的符号位 1 不一样，即移位后的尾数是 01111100（最后两个 0 是补上去的）。同时阶码要从原值（0100）相应减 2（即减 0010），得到 0010。

### 1.7.4 舍入处理

在“对阶”和向左、向右规格化处理时，尾数要向左、向右移位。这样被移位的尾数与实际的尾数之间就可能存在一定误差，因此要进行舍入处理，以尽可能减小这种误差。

简单的舍入方法有以下两种：

（1）“0 舍 1 入”法。

类似于我们十进制数中的“四舍五入”法。即如果左规或右规时被丢弃的数位为 0，则舍去不计，反之要将尾数的末位加“1”。

如前面提到的 X 与 Y 的尾数和为 1.11011111（最高位为符号位），需要左规，尾数要向左移两位，得到 1.01111100，由此可见左移时去掉的是前面两位 1，需要在尾数的最后一位加 1。这样进行舍入处理后就得到 1.01111101。

（2）“恒置 1”法。

只要有数位被左规右规丢弃掉，就要在尾数的末位恒置“1”。此法精确度不高，因为从概率上来讲，丢弃的 0 和 1 各为 50%。

**【说明】**在 IEEE 754 标准中有更多的舍入模式，但比较复杂，在此不做介绍。

### 1.7.5 溢出处理

是否有溢出，在本章前面介绍了，如果采用双符号位补码形式，出现的双符号位为“01”或“10”时就会自动判定有溢出。但是如果采用的是单符号位，就不好判定了。这时要根据在对尾数进行规格化处理后“阶码”所能表示的取值范围是否超出了对应阶码位所能表示数的范围来定了。

前面已介绍到，右规后阶码要加相应的值，尾数每向右移一位加 1，最终可能造成新的阶码超出了原来用于表示阶码位数所能表示的最大数值（称为“上溢”）。

如某$[X+Y]_{补}$浮点数经过“对阶”和尾数相加后得到的尾数为 10.101110011，显然它需要右规，结果为 11.01110011，相当于尾数向右移了一位，原来尾数的最高有效位“1”被丢弃了，同时阶码也要相应加 1。再假设用来存储阶码的位数为 4（包括一位阶码符号位，所能表示的最大阶码数为+7），因原阶码为 0111，尾数右移一位后，阶码要加 1 就为 8 了，而这超出了 4 位阶码中的 3 个真值位所能表示的最大数+7，所以这时就要做溢出处理了。

当然，当尾数被左规时也可能造成溢出，因为左规后可能超出了用于存储阶码的存储位所能表示的最小值（称为“下溢”）。如左规后得到的新阶码为-5，假设阶码规定是用 4 位来存储的，现在要求尾数再向左移 3 位，这样阶码也要减小 3，得到-8，而-8 超出了 4 位阶码（最高位为阶码符号位）所能表示的最小数-7。

如果发现有溢出，计算机设备会根据以下原则进行处理：

- 如果是上溢，则停止运算，做中断处理。
- 如果是下溢，整个尾数按 0 处理。

## 1.8 信息编码

“数”不仅仅用来表示“量”，它还能作为代码（Code）来使用。例如，在我国实行的公民身份证制度，身份证上有一组 18 位的数为“身份证号码”。这就是用数字进行编码的例子。再如每一个学生入学后都会有一个学号，这也是一种编码。

编码的目的之一是为了便于标记特定的对象。为了便于记忆和查找，在设计编码时需要按照一

定的规则。这里介绍最常用的几种计算机编码，如 ASCII 码和一些汉字编码。

### 1.8.1 西文编码

西文编码就是对西文字母、符号或者操作进行的编码，常见的有 ASCII 码（American Standard Code for Information Interchange，美国标准信息交换代码）和 Unicode 码两种。

1. ASCII 码

ASCII 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。它是现今最通用的单字节编码系统，并等同于国际标准 ISO/IEC 646。目前分为“标准 ASCII 码”和“扩展 ASCII 码”两种。

（1）标准 ASCII 码。

标准 ASCII 码采用 7 位二进制进行编码（共 8 位，但仅由低 7 位对信息进行编码，最高位为奇偶校验位），可组合表示 128（$2^7$）种状态，对应 128 个（对应十进制取值为 0～127）字符（或控制符）。其中包括 26 个英文大写字符、26 个英文小写字符、10 个数字字符、33 个标点符号和 33 个控制符。例如大写 A 的 ASCII 码是 65，小写 a 是 97。



【说明】奇偶校验是用来校验信息在传输过程中是否出现了差错的一种方法，分奇校验和偶校验两种。这两种校验方法分别是在信息传输前通过先在信息码的最高位前面加上校验码，使得最终所传输的信息码“1”的个数分别为奇数或偶数，然后在接收端再验证信息码中“1”的个数是否仍为奇数或偶数，以此来判别信息在传输过程中是否出现了差错。具体将在第 6 章介绍。

为便于书写和记忆，有时也将 ASCII 写作十六进制形式（一个字节恰好可以表示两个十六进制），即将某字符的 ASCII 码二进制数形式转换成十六进制数的形式，再标以 H 表示这是一个十六进制的数。例如 A 字母的 ASCII 码为 01000001，写成十六进制即 41H；C 字母的 ASCII 码为 01000011，写成十六进制即 43H。

以上 128 个标准 ASCII 码字符分为两部分：

- 第一部分：由 00H 到 1FH 共 32 个，一般用于通信或作为控制之用，有些可以显示在屏幕上，有些则不能显示，但能看到其效果（如换行、退格）。
- 第二部分：由 20H 到 7FH 共 96 个，这 96 个字符用来表示阿拉伯数字、英文字母大小写和下划线、括号等符号，都可以显示在屏幕上。

这 128 个字符的编码规则如图 1-27 所示，$D_n$ 代表码位。例如大写字母 C 的 ASCII 码，只需在图中对应于字符 C 的位置找出其横坐标 D6D5D4 和纵坐标 D3D2D1D0，依次按 D6D5D4D3D2D1D0 的顺序排列出来，再在最高位补以 0，即得 C 的 ASCII 码为 01000011。

（2）扩展 ASCII 码。

扩展 ASCII 码采用 8 位二进制进行编码，是对标准 ASCII 码的扩展，是由 IBM 制定的，并非标准的 ASCII 码。扩展 ASCII 码由 80H 到 0FFH 共 128 个字符，用来表示框线、音标和其他欧洲非英语系的字母。

| $D_3D_2D_1D_0$ \ $D_6D_5D_4$ | 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |
|---|---|---|---|---|---|---|---|---|
| 0000 | NUL | DLE | SP | 0 | @ | P | 、 | p |
| 0001 | SOH | DC1 | ! | 1 | A | Q | a | q |
| 0010 | STX | DC2 | " | 2 | B | R | b | r |
| 0011 | ETX | DC3 | # | 3 | C | S | c | s |
| 0100 | EOT | DC4 | $ | 4 | D | T | d | t |
| 0101 | ENQ | NAK | % | 5 | E | U | e | u |
| 0110 | ACK | SYN | & | 6 | F | V | f | v |
| 0111 | BEL | ETB | ' | 7 | G | W | g | w |
| 1000 | BS | CAN | ( | 8 | H | X | h | x |
| 1001 | HT | EM | ) | 9 | I | Y | i | y |
| 1010 | LF | SUB | * | : | J | Z | j | z |
| 1011 | VT | ESC | + | ; | K | [ | k | { |
| 1100 | FF | FS | , | < | L | \ | l | \| |
| 1101 | CR | GS | - | = | M | ] | m | } |
| 1110 | SO | RS | . | > | N | ^ | n | ~ |
| 1111 | SI | US | / | ? | O | _ | o | DEL |

图 1-27　标准 ASCII 码

2. Unicode 码

Unicode 码最初是 Apple 公司发起制定的通用多文种字符集，后来被 Unicode 协会开发为能表示几乎世界上所有书写语言的字符编码标准。Unicode 码包括 UTF-8、UTF-16 和 UTF-32 这 3 种编码格式，分别指使用 8 位、16 位和 32 位表示字符，在此不作详细介绍。

### 1.8.2　中文编码

计算机中汉字的表示也是用二进制编码。根据应用目的的不同，汉字编码分为外码、交换码、机内码和字形码。

1. 外码

“外码”也叫输入码，用来解决汉字输入的编码问题，是可用来将汉字输入到计算机中的编码方式。常用的外码（或输入码）有：拼音码、五笔字型码、自然码、表形码、认知码、区位码和电报码等。

2. 交换码

交换码（又称“国标码”）是由中国标准总局于 1981 年制定的中华人民共和国国家标准 GB2312－80《信息交换用汉字编码字符集——基本集》，是在大陆及海外使用简体中文的地区（如新加坡等）强制使用的唯一中文编码。

在 GB2312－80 标准中共收录 6763 个汉字，其中一级汉字 3755 个，二级汉字 3008 个；同时收录了包括拉丁字母、希腊字母、日文平假名及片假名字母、俄语西里尔字母在内的 682 个字符。交换码仅是在计算机内部的一个中文汉字、字符集标准，每个汉字的具体编码实现方式则由下面将要介绍的“区位码”解决。

3. 区位码

“区位码”是国标码的具体编码实现方式。它把国标 GB2312－80 中的汉字、图形符号组成一个 94×94 的方阵，即 94 个区，每个区中又有 94 个位，对应 94 个汉字或字符，总区位数为 8836 个，即总共可编码 8836 个汉字或字符。区位码用来解决计算机识别汉字的能力，也可作为输入码（如区位码输入法），但它并不是计算机内部处理汉字的编码方式，在计算机内部汉字编码是由下面将要介绍的“机内码”进行的。

根据不同区所表示的字符类型可以把整个区位码分成 4 个部分：①01～09 区为特殊符号；②16～55 区为一级汉字，按拼音排序；③56～87 区为二级汉字，按部首/笔画排序；④10～15 区及 88～94 区目前暂未使用，留待以后扩展汉字或字符使用。

4. 机内码

“机内码”是根据“国标码”的规定确定在计算机内部表示一个汉字的二进制代码。在计算机内部和磁盘中汉字代码都用机内码表示。

5. 字形码

字形码是汉字的输出码，是解决汉字输出的编码方式。输出汉字时都采用图形方式，无论汉字的笔画多少，每个汉字都可以写在同样大小的方块中。通常用 16×16 点阵来显示汉字。



## 1.9 课后自测题

为了巩固本章所学内容，请认真做下面的自测题，参考答案见附录。

### 1.9.1 填空题

1．计算机运算中主要有_____、_____、_____和_____4 种数据制式，分别用_____、_____、_____和_____字符标识，它们的基数分别是_____、_____、_____和_____。其中对应 3 位二进制码的制式是_____。

2．有一个共 12 位的二进制数，如果用 $b_n$（n 表示位号）表示各个二进制位值的话，整个二进制数可表示为_____，如果把它转换成十进制，则它的计算展开式为_____。如果要把这个 12 位二进制数转换成十六进制，则这个十六进制数共有_____位。假设现有一个从 A4000H 到 CBFFFH 的内存地址段，则这个地址段最多可以存放_____字节数据。

3．二进制的算术运算中：1+1=_____，0-1=_____，借 1 当_____，进 1 当_____；二进制的逻辑运算中：1∧0＝_____，1∨0＝_____，1⊕0＝_____。

4．机器数的最高位是_____，其余各位是该数的_____；反码的编码规则是：正数的反码是_____，负数的反码是_____；补码的编码规则是：正数的补码是_____，负数的补码是_____。

5．机器数 10010110 的真值为_____（用十进制表示），原码为_____，反码为_____，补码为_____（各种编码用二进制数表示）。

6．8 位定点整数 X 的取值范围为_____，8 位定点小数 X 的取值范围为_____。机器数 10110011 的移码是_____。

7．单精度浮点数一共_____位，指数一共_____位，偏移值为_____，双精度浮点数一共_____位，尾数一共_____位，偏移值为_____。

8．中文汉字编码方式有很多种，我们平时使用的 QQ 拼音输入法属于_____，区位码输入法属于_____，在计算机磁盘中表示汉字所采用的是_____，在屏幕上显示汉字所采用的是_____，为汉字输入、输出提供基本汉字、字符集标准的是_____。

### 1.9.2 选择题（可多选）

1．8 个二进制位至多可表示（　　）个数据。

A．8　　B．64　　C．255　　D．256

2．对于 R 进制数，在每一位上的数字可以有（　　）种。

A．R/2　　B．R-1　　C．R　　D．R+1

3．假设用 12 个二进制位表示数据，它能表示的最大无符号整数为（　　）。

A．2047　　B．2048　　C．4095　　D．4096

4．下列无符号数中最大的是（　　）。

A．$(205)_D$　　B．$(001001010010)_B$

C．$(0CD)_H$　　D．$(11100011)_B$

5．与二进制数 100101.001101 等值的十进制数是（　　），等值的十六进制数是（　　）。

A．25.203125　　B．25.34　　C．37.203125　　D．37.34

6．与十进制数 28.625 等值的十六进制数为（　　），与十进制数 254 等值的二进制数是（　　）。

A．112.10　　B．1C.A　　C．1C.5　　D．112.5

E．11111110　　F．11101111　　G．11111011　　H．11101110

7．与十六进制数 AC.E 等值的十进制数是（　　），等值的八进制数是（　　）。

A．112.875　　B．162.875　　C．254.7　　D．172.875

8．与十六进制数 23.4 等值的十进制数为（　　），0x1000 转换成十进制是（　　）。

A．35.5　　B．23.4　　C．35.75　　D．35.25

E．4096　　F．1024　　G．2048　　H．8192

9．下列关于浮点数说法不正确的是（　　）。

A．浮点数中的小数点位置可变

B．浮点数可用一个纯小数和一个指数的乘积表示

C．浮点数的尾数的最高有效位必须为 1

D．浮点数的尾数必须用阶码表示

10．对于两个机器数 55H 和 AAH，运算结果相同的逻辑操作是（　　）。

A．逻辑与　　B．逻辑或　　C．逻辑非　　D．逻辑异或

11．若 X=10111001，Y=11110011，则它们分别进行“逻辑与”和“逻辑异或”运算后的结果分别为（　　）。

A．110101100，000001101　　B．001010011，111110010

C．10110001，01001010　　D．01001110，11111011

12．二进制整数采用机器码表示时，（　　）的表示范围最大。

A．原码　　B．补码　　C．反码　　D．移码

13．在计算机加减法运算中，最常使用的是（　　）。

A．原码　　B．补码　　C．反码　　D．ASCII 码

14．反码的作用是（　　）。

A．作为求补码的中间手段　　B．作为求原码的中间手段

C．能将负数转换为正数　　D．能将减法转化为加法

15．下面关于补码的作用说法不正确的是（　　）。

A．使机器数的码制更简单　　B．使计算机的运算效率更高

C．能将负数转换为正数进行运算　　D．能将减法运算转化为加法运算

16．将-33 以单符号位补码形式存入 8 位寄存器中，寄存器中的内容为（　　）。

A．DFH　　B．A1H　　C．5FH　　D．DEH

17．对+0 和-0 表示形式唯一的机器数表示形式是（　　）。

A．原码　　B．补码　　C．反码　　D．移码

18．若用 8 位机器码表示十进制数-101，则其原码表示形式为（　　），补码表示形式为（　　）。

A．11100101　　B．10011011　　C．11010101　　D．11100111

19．多项式 $2^{14}+2^{11}+2^{4}+2^{1}+2^{0}$ 表示的十六进制数为（　　），表示的十进制数为（　　）。

A．4813H　　B．8026H　　C．2410H　　D．EB410H

E．18448　　F．9232　　G．18451　　H．36902

20．设字长 8 位并用定点整数表示，模为 $2^{8}$，若$[X]_{补}$=11111010，则 X 的原码及真值分别为（　　）。

A．$[X]_{原}$=00000110，X=+0000110　　B．$[X]_{原}$=10000110，X=-0000110

C．$[X]_{原}$=01111010，X=+1111010　　D．$[X]_{原}$=11111010，X=-0000110

21．真值 X=-127D，则其 8 位反码及真值分别为（　　）。

A．$[X]_{反}$=11111111，X=-1000000　　B．$[X]_{反}$=10000000，X=-1000000

C．$[X]_{反}$=11111111，X=-1111111　　D．$[X]_{反}$=10000000，X=-1111111

22．若$[X]_{补}$=CCH，机器字长为 8 位，则$[X/2]_{补}$=（　　）。

A．34H　　B．66H　　C．98H　　D．E6H

### 1.9.3　计算题

1．按要求进行下列数制转换（假设全部为无符号数）：

（1）把(01110100)B、(11101001000.10111)B 转换成十进制。

（2）把(1076)O、(6374.65)Q 转换成十进制。

（3）把 0x7A8C、0x259B.25 转换成十进制。

（4）把十进制数 825、10815.6（精确到小数点后 4 位）转换成二进制。

（5）把十进制数 658、9240.65（精确到小数点后 4 位）转换成八进制。

（6）把十进制数 2508、5420.82（精确到小数点后 4 位）转换成十六进制。

（7）把(1011011)B、(10110111.0010)B 转换成八进制。

（8）把(1100111011)B、(11101001.101)B 转换成十六进制。

（9）把(756)O、(6265.42)O 转换成十六进制。

（10）把 0xA58C、0x8152.78 转换成八进制。

2．求下列二进制算术运算结果（假设全部为无符号数）：

（1）求(011101)B +(10010)B、(100111)B +(110110)B 的值。

（2）求(1110101)B -(110010)B、(1101011)B -(10001)B 的值。

（3）求(1110)B ×(1001)B、(1100)B ×(10111)B 的值。

（4）求(110010)B ÷(1010)B、(100110101)B ÷(1011)B 的值。

（5）求(11001)B∧(1011)B、(10011)B∧(10101)B 的结果。

（6）求(11001)B∨(1011)B、(10011)B∨(10101)B 的结果。

（7）求(1100110)B 非和(1000111)B 非的结果。

（8）求(1100110)B ⊕(1011)B、(100111)B ⊕(10101)B 的结果。

3．求下列补码运算结果（以二进制补码形式表示）：

（1）求$[85+24]_{补}$、$[152+35]_{补}$的值。

（2）求$[185-56]_{补}$、$[52-135]_{补}$的值。

4．求以下浮点数：

（1）求十进制数 20.5 的单精度浮点数。

（2）两浮点数 $X=2^{101}\times0.11011011$，$Y=2^{111}\times(-0.10101100)$，指数和尾数均为二进制原码。现求在存储器中该数的尾数以 9 位补码形式（单符号位），指数以 4 位补码形式（单符号位）的移码的表示形式。