## 3.3　计算机运算出错的原因

在了解了将二进制数表示的小数转换成十进制数的方法后，计算机运算出错的原因也就容易理解了。这里我先把答案告诉大家，计算机之所以会出现运算错误，是因为“有一些十进制数的小数无法转换成二进制数”。例如，十进制数 0.1，就无法用二进制数正确表示，小数点后面即使有几百位也无法表示。接下来，我们就来看一下不能正确表示的原因。

图 3-2 中，小数点后 4 位用二进制数表示时的数值范围为 0.0000～0.1111。因此，这里只能表示 0.5、0.25、0.125、0.0625 这四个二进制数小数点后面的位权组合而成（相加总和）的小数。将这些数值组合后能够表示的数值，即为表 3-1 中所示的无序的十进制数。

表 3-1　小数点后 4 位能够用二进制数表示的数值
二进制数是连续的，十进制数是非连贯的

| 二进制数 | 对应的十进制数 |
|---|---|
| 0.0000 | 0 |
| 0.0001 | 0.0625 |
| 0.0010 | 0.125 |
| 0.0011 | 0.1875 |
| 0.0100 | 0.25 |
| 0.0101 | 0.3125 |
| 0.0110 | 0.375 |
| 0.0111 | 0.4375 |
| 0.1000 | 0.5 |
| 0.1001 | 0.5625 |
| 0.1010 | 0.625 |
| 0.1011 | 0.6875 |
| 0.1100 | 0.75 |
| 0.1101 | 0.8125 |
| 0.1110 | 0.875 |
| 0.1111 | 0.9375 |

表 3-1 中，十进制数 0 的下一位是 0.0625。因此，这中间的小数，就无法用小数点后 4 位数的二进制数来表示。同样，0.0625 的下一位数一下子变成了 0.125。这时，如果增加二进制数小数点后面的位数，与其相对应的十进制数的个数也会增加，但不管增加多少位，2 的 −〇〇次幂怎么相加都无法得到 0.1 这个结果。实际上，十进制数 0.1 转换成二进制后，会变成 0.00011001100…（1100 循环）这样的**循环小数**[①]。这和无法用十进制数来表示 1/3 是一样的道理。1/3 就是 0.3333…，同样是循环小数。

至此，大家应该明白了为什么用代码清单 3-1 的程序无法得到正确结果了吧。因为无法正确表示的数值，最后都变成了近似值。计算机这个功能有限的机器设备，是无法处理无限循环的小数的。因此，在遇到循环小数时，计算机就会根据变量数据类型所对应的长度将数值从中间截断或者四舍五入。我们知道，将 0.3333…这样的循环小数从中间截断会变成 0.333333，这时它的 3 倍是无法得出 1 的（结果是 0.999999），计算机运算出错的原因也是同样的道理。

## 3.4 什么是浮点数

像 1011.0011 这样带小数点的表现形式，完全是纸面上的二进制数表现形式，在计算机内部是无法使用的。那么，实际上计算机是以什么样的表现形式来处理小数的呢？我们一起来看一下。

很多编程语言中都提供了两种表示小数的数据类型，分别是双精度浮点数和单精度浮点数。双精度浮点数类型用 64 位、单精度浮点数

① 像 0.3333…这样相同数值无限循环的值称为循环小数。计算机是功能有限的机器，无法直接处理循环小数。

类型用 32 位来表示全体小数[①]。在 C 语言中，双精度浮点数类型和单精度浮点数类型分别用 double 和 float 来表示。不过，这些数据类型都采用浮点数[②]来表示小数。那么，浮点数究竟采用怎样的方式来表示小数呢？接下来就让我们一起来看一下。

**浮点数**是指用符号、尾数、基数和指数这四部分来表示的小数（图 3-3）。因为计算机内部使用的是二进制数，所以基数自然就是 2。因此，实际的数据中往往不考虑基数，只用符号、尾数、指数这三部分即可表示浮点数。也就是说，64 位（双精度浮点数）和 32 位（单精度浮点数）的数据，会被分为三部分来使用（图 3-4）。

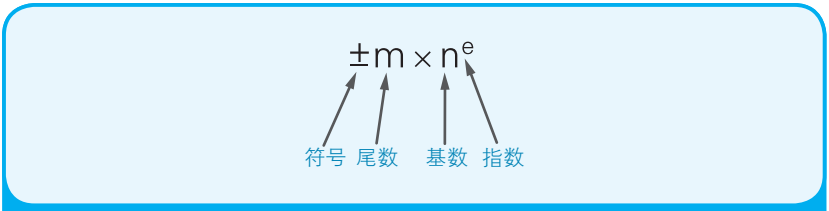


图 3-3　浮点数的表现形式。由符号、尾数、基数、指数四部分构成

① 双精度浮点数能够表示的正数范围是 $4.94065645841247 \times 10^{-324}$～$1.79769313486232 \times 10^{308}$，负数范围是 $-1.79769313486232 \times 10^{308}$～$-4.94065645841247 \times 10^{-324}$。单精度浮点数能够表示的正数范围是 $1.401298 \times 10^{-45}$～$3.402823 \times 10^{38}$，负数范围是 $-3.402823 \times 10^{38}$～$-1.401298 \times 10^{-45}$。不过，正如正文中所介绍的那样，在这些范围中，有些数值是无法正确表示的。

② 像 $0.12345 \times 10^{3}$ 和 $0.12345 \times 10^{-1}$ 这样使用与实际小数点位置不同的书写方法来表示小数的形式称为浮点数。与浮点数相对的是定点数，使用定点数表示小数时，小数点的实际位置固定不变。例如，$0.12345 \times 10^{3}$ 和 $0.12345 \times 10^{-1}$ 用定点数来表示的话即为 123.45 和 0.012345。

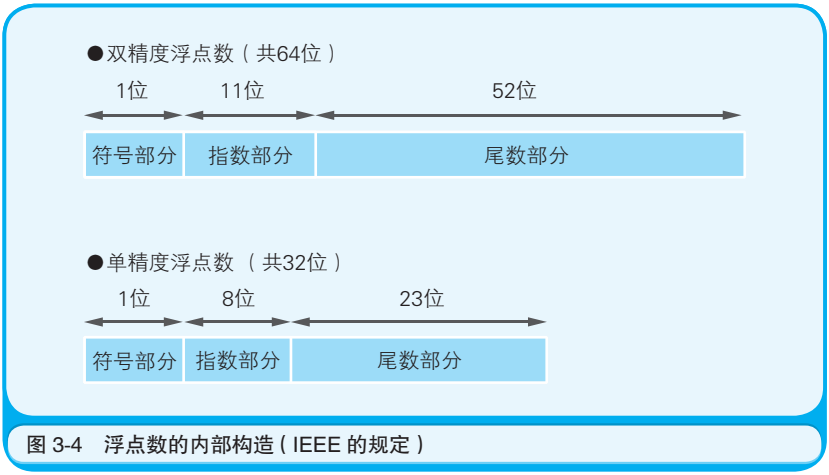


图 3-4 浮点数的内部构造（IEEE 的规定）

浮点数的表现方式有很多种，这里我们使用最为普遍的 IEEE[①] 标准。双精度浮点数和单精度浮点数在表示同一个数值时使用的位数不同。此外，双精度浮点数能够表示的数值范围要大于单精度浮点数。

**符号部分**是指使用一个数据位来表示数值的符号。该数据位是 1 时表示负，为 0 时则表示“正或者 0”。这和用二进制数来表示整数时的符号位是同样的。数值的大小用尾数部分和指数部分来表示。例如，小数就是用“尾数部分 × 2 的指数部分次幂”这样的形式来表示的。讲到这里，大家是不是多少有点概念了呢。

下面的内容可能稍微有点复杂，因为尾数部分和指数部分并不只是单单存储着用整数表示的二进制数。**尾数部分**用的是“将小数点前面的值固定为 1 的正则表达式”，而**指数部分**用的则是“EXCESS 系统表现”。此外，接下来还会涉及大量的新术语，大家可能会因此产生逃避心理。不过，这些其实并不难，因此请大家一定要耐心地阅读下去。

① IEEE（Institute of Electrical and Electronics Engineers）是指美国电气和电子工程师协会。该协会制定了计算机领域的各种规定。读作“eye-triple-e，I-3E”。

## 3.5　正则表达式和 EXCESS 系统

**尾数部分**使用**正则表达式**[①]，可以将表现形式多样的浮点数统一为一种表现形式。例如，十进制数 0.75 就有很多种表现形式，如图 3-5 所示。虽然它们表示的都是同一个数值，但因为表现方法太多，计算机在处理时会比较麻烦。因此，为了方便计算机处理，需要制定一个统一的规则。例如，十进制数的浮点数应该遵循“小数点前面是 0，小数点后面第 1 位不能是 0”这样的规则。根据这个规则，0.75 就是“0.75×10 的 0 次幂”，也就是说，只能用尾数部分是 0.75、指数部分是 0 这个方法来表示。根据这个规则来表示小数的方式，就是正则表达式。

$$0.75 = 0.75 \times 10^{0}$$

$$0.75 = 75 \times 10^{-2}$$

$$0.75 = 0.075 \times 10^{1}$$

图 3-5　浮点数可以用不同的形式来表现同一个数值

刚才以十进制数为例进行了说明，二进制数也是同样的道理。在二进制数中，我们使用的是“**将小数点前面的值固定为 1 的正则表达式**”。具体来讲，就是将二进制数表示的小数左移或右移（这里是逻辑移位。因为符号位是独立的[②]）数次后，整数部分的第 1 位变为 1，第 2

① 按照特定的规则来表示数据的形式即为正则表达式。除小数之外，字符串以及数据库等，也都有各自的正则表达式。

② 整数是指使用包含表示符号的最高位在内的全体来表示的一个数值。而浮点数是由符号部分、尾数部分和指数部分这三部分独立的数值组合而成的。

位之后都变为 0（这样是为了消除第 2 位以上的数位）。而且，第 1 位的 1 在实际的数据中不保存。由于第 1 位必须是 1，因此，省略该部分后就节省了一个数据位，从而也就可以表示更多的数据范围（虽不算太多）。

单精度浮点数的正则表达式的具体例子如图 3-6 所示。单精度浮点数中，尾数部分是 23 位，但由于第 1 位的 1 被省略了，所以实际上可以表示 24 位的数值。双精度浮点数的表示方法也是如此，只是位数不同而已。

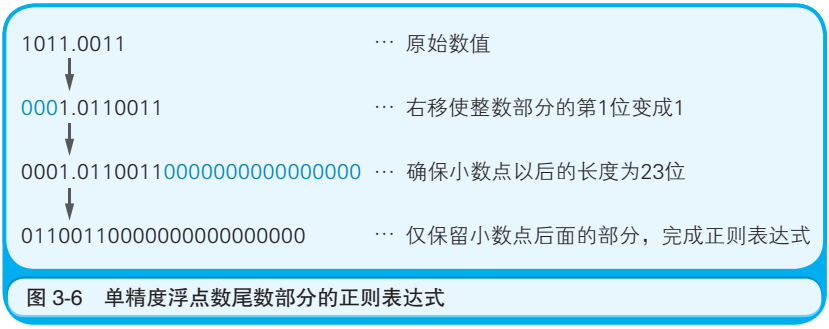


图 3-6 单精度浮点数尾数部分的正则表达式

接下来，让我们一起来看一下**指数部分**中使用的 EXCESS 系统，使用这种方法主要是为了表示负数时不使用符号位。在某些情况下，在指数部分，需要通过“负○○次幂”的形式来表示负数。**EXCESS 系统表现**是指，通过将指数部分表示范围的中间值设为 0，使得负数不需要用符号来表示。也就是说，当指数部分是 8 位单精度浮点数时，最大值 11111111 = 255 的 1/2，即 01111111 = 127（小数部分舍弃）表示的是 0，指数部分是 11 位双精度浮点数时，11111111111 = 2047 的 1/2，即 01111111111 = 1023（小数部分舍弃）表示的是 0。

EXCESS 系统可能不太好理解，下面举例来说明。假设有这样一个游戏，用 1 ～13（A～K）的扑克牌来表示负数。这时，我们可以把

中间的 7 这张牌当成 0。如果扑克牌 7 是 0，10 就表示 +3，3 就表示 −4。事实上，这个规则说的就是 EXCESS 系统。

作为单精度浮点数的示例，表 3-2 中列出了指数部分的实际值和用 EXCESS 系统表现后的值。例如，指数部分为二进制数 11111111（十进制数 255），那么在 EXCESS 系统中则表示为 128 次幂。这是因为 255 − 127 = 128。因此，8 位的情况下，表示的范围就是 -127 次幂～128 次幂。

表 3-2　单精度浮点数指数部分的 EXCESS 系统表现

| 实际的值（二进制数） | 实际的值（十进制数） | EXCESS 系统表现（十进制数） |
|---|---|---|
| 11111111 | 255 | 128（= 255 − 127） |
| 11111110 | 254 | 127（= 254 − 127） |
| ⋮ | ⋮ | ⋮ |
| 01111111 | 127 | 0（= 127 − 127） |
| 01111110 | 126 | − 1（= 126 − 127） |
| ⋮ | ⋮ | ⋮ |
| 00000001 | 1 | − 126（= 1 − 127） |
| 00000000 | 0 | − 127（= 0 − 127） |

## 3.6　在实际的程序中进行确认

读到这里，有人额角冒汗吗？上述内容不是仅仅读一遍就能马上理解的，最好能够在实际的程序中加以确认。因此，我们准备了一个试验用的程序，如代码清单 3-2 所示。接下来，就让我们一起看一下如何用单精度浮点数来表示十进制数 0.75 吧。

代码清单 3-2　用于确认单精度浮点数表示方法的 C 语言程序

```
#include <stdio.h>
#include <string.h>

void main() {
    float data;
    unsigned long buff;
```

```
    int i;
    char s[34];

    // 将 0.75 以单精度浮点数的形式存储在变量 date 中。
    data = (float)0.75;

    // 把数据复制到 4 字节长度的整数变量 buff 中以逐个提取出每一位。
    memcpy(&buff, &data, 4);

    // 逐一提取出每一位
    for (i = 33; i >= 0; i--) {
        if(i == 1 || i == 10) {
            // 加入破折号来区分符号部分、指数部分和尾数部分。
            s[i] = '-';
       } else {
            // 为各个字节赋值 '0' 或者 '1'。
            if (buff % 2 == 1) {
                s[i] = '1';
            } else {
                s[i] = '0';
            }
            buff /= 2;
        }
    }
    s[34] = '\0';

     // 显示结果。
     printf("%s\n", s);
}
```

该程序执行后，十进制数 0.75 用单精度浮点数来表示就变成了 0-01111110-10000000000000000000000（图 3-7）。加入破折号（-）是为了区分符号部分、指数部分、尾数部分。这里，符号部分为 0，指数部分为 01111110，尾数部分为 10000000000000000000000。因为 0.75 是正数，所以符号位是 0。指数部分的 01111110 是十进制数 126，用 EXCESS 系统表现就是 −1（126 − 127 = − 1）。根据正则表达式的规则，小数点前面的第 1 位是 1，因此尾数部分 10000000000000000000000 实际上表示的是 1.10000000000000000000000 这个二进制数。将尾数部分的二进制数转换成十进制数，结果就是（1 × 2 的 0 次幂）+（1 × 2 的 −1

次幂）= 1.5。因此，0-01111110-10000000000000000000000 这个单精度浮点数，表示的就是“+1.5 × 2 的 −1 次幂”。2 的 −1 次幂是 0.5，+1.5 × 0.5 = + 0.75。正好吻合，结果正确。

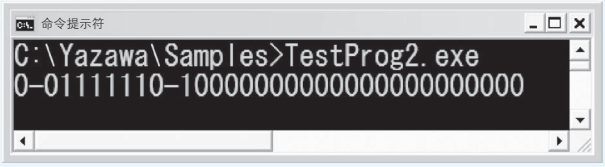


图 3-7　代码清单 3-2 的运行结果

接下来，我们继续使用该程序来看一下如何用单精度浮点数表示十进制数 0.1。运行后就会发现结果为 0-01111011-10011001100110011001101（只需将 data = (float)0.75; 的部分变成 data = (float)0.1; 即可）。这时，如果反过来计算一下这个数值的十进制数，估计大家又要冒汗了，结果居然不是 0.1。

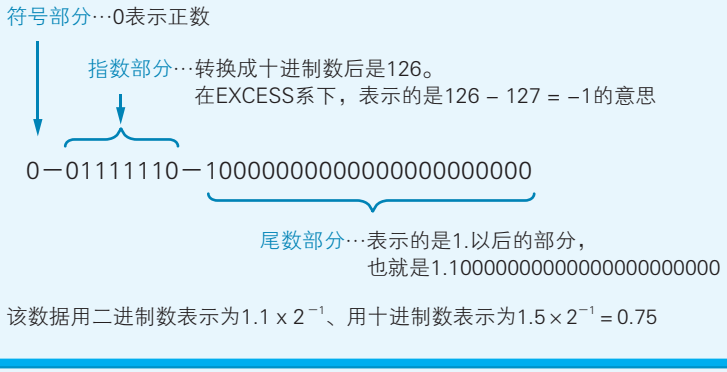


图 3-8　用单精度浮点数表示的数据

## 3.7 如何避免计算机计算出错

计算机计算出错的原因之一是，采用浮点数来处理小数（另外，也有因“位溢出”而造成计算错误的情况）。作为程序的数据类型，不管是使用单精度浮点数还是双精度浮点数，都存在计算出错的可能性。接下来将介绍两种避免该问题的方法。

首先是回避策略，即无视这些错误。根据程序目的的不同，有时一些微小的偏差并不会造成什么问题。例如，假设使用计算机设计工业制品。将 100 个长 0.1 毫米的零件连接起来后，其长度并非一定要是 10 毫米，10.000002 毫米也没有任何问题。一般来讲，在科学技术计算领域，计算机的计算结果只要能得到近似值就足够了。那些微小的误差完全可以忽略掉。

另一个策略是把小数转换成整数来计算。计算机在进行小数计算时可能会出错，但进行整数计算（只要不超过可处理的数值范围）时一定不会出现问题。因此，进行小数的计算时可以暂时使用整数，然后再把计算结果用小数表示出来即可。例如，本章一开头讲过的将 0.1 相加 100 次这一计算，就可以转换为将 0.1 扩大 10 倍后再将 1 相加 100 次的计算，最后把结果除以 10 就可以了（代码清单 3-3）。

**代码清单 3-3　将小数替换成整数来计算的 C 语言程序**

```
#include <stdio.h>

void main() {
    //int 是整数的数据类型
    int sum;
    int i;

    // 将保存总和的变量清 0
    sum = 0;
```