关于 C 语言中浮点数的概念及其使用的问题

耿楠

西北农林科技大学信息工程学院，陕西 · 杨凌，712100

2016 年 10 月 19 日

摘要

浮点数类型是 C 语言中一种重要的类型，由于长期以来，C 语言教学中并没有深入分析有关浮点数的细节，从而造成了对浮点数不能直接比较大小、相差过大的浮点数不能加减、浮点数不够精确等现象深入理解的困难。针对这一问题，本文深入分析了有关浮点数在计算机内部表达的细节和规定这一细节的 IEEE 754 标准，以期对浮点数类型的本质进行说明，从而更加有效地和准确地使用浮点数。同时，本文也将为后续其它语言中浮点数的学习提供必要的参考。

一、引言

电子电气工程师协会 (IEEE，Institute of Electrical and Electronics Engineers) 在 1985 年制定的 IEEE 754 标准对二进制浮点数的运算进行了规范，并成为实现浮点运算部件事实上的工业标准。本文通过对浮点数、IEEE 754 标准的浮点数表示方法、规格化处理等进行分析，以期对浮点数类型的本质进行说明，从而更加有效地和准确地使用浮点数。同时，本文也将为后续其它语言中浮点数的学习提供必要的参考。

二、浮点数

在计算机系统的发展过程中，对于实数，曾经提出定点数表示法、有理数 (两个整数的比值) 表示法等过多种方法，但是到目前为止，浮点表示法是使用最为广泛的一种方法。

相对于定点数而言，浮点数利用指数使小数点的位置可以根据需要进行浮动，从而可以灵活地同时表达特别大的数或者特别小的数，扩大了实数的表达范围。

浮点数表示法利用科学计数法来表达实数，通常，将浮点数表示为：

$$\pm\, d.dd\ldots d \times \beta^e \tag{1}$$

式中，$d.dd\ldots d$ 称为有效数字 (Significand，有时也称为尾数—Mantissa，尾数是有效数字的非正式说法)，它具有 p 个数字 (称 p 位有效数字精度)，β 为基数 (Base)，e 为指数 (Exponent)，$\pm$ 表示实数的正负，因此，$\pm d_0.d_1d_2\ldots d_{p-1} \times \beta^e$ 所表达的实数为：

$$\pm\left(d_0 \times \beta^0 + d_1 \times \beta^{-1} + \ldots + d_{p-1} \times \beta^{(-(p-1))}\right) \times \beta^e, (0 \le d_i \le \beta) \tag{2}$$

对实数的浮点表示仅作如上的规定是不够的，因为同一实数的浮点表示仍不是唯一的。例如，1.0×10^2、0.1×10^3 和 0.01×10^4 都可以表示 100.0。为了达到表示单一性的目的，有必要对其进一步规范。规定有效数字的最高位 (即前导有效位) 必须非零，即 $0 < d_0 < \beta$。符合该标准的数称为规格化数（Normalized Numbers），否则称为非规格化数 (Denormalized Numbers)。

三、IEEE 浮点数

IEEE 754 标准中规定一个实数 V 用 $V\ (-1)^s \times M \times 2^E$ 的形式表示，其中：

1）符号 s(sign) 决定实数是正数 $(s=0)$ 还是负数 $(s=1)$，对数值 0 的符号位需要特殊处理。

2）有效数字 M(significand) 是二进制小数，M 的取值范围在 $1 \le M < 2$ 或 $0 \le M < 1$。

3）指数 E(exponent) 是 2 的幂，它的作用是对浮点数加权。

浮点格式是一种数据结构，它规定了构成浮点数的各个字段、这些字段的布局、及其算术解释。IEEE 754 标准浮点数的数据位被划分为 3 个字段，对以上参数值进行编码：

1）一个单独的符号位 s 直接编码符号 s。

2）k 位的偏置指数 $e(e = e_{k-1} \dots e_1 e_0)$ 编码指数 E，用 **移码**表示。

3）n 位的小数 f(fraction)$(f = f_{k-1} \dots f_1 f_0)$ 编码有效数字 M，**原码**表示。

根据偏置指数 e 的值，被编码的浮点数可分成三种类型。

1、规格化数

当有效数字 M 在范围 $1 \le M < 2$ 中且指数 e 的位模式 $e_{k-1} \dots e_1 e_0$ 既不全是 0 也不全是 1 时，浮点格式所表示的数都属于规格化数。这种情况中小数 $f(0 \le f < 1)$ 的二进制表示为 "$0.f_{n-1} \dots f_1 f_0$"。有效数字 $M = 1 + f$，即 "$M = 1.f_{n-1} \dots f_1 f_0$" (其中小数点左侧的数值位称为**前导有效位**) 。通过调整指数 e，可以使得有效数字 M 在范围 $1 \le M < 2$ 中，这样有效数字的前导有效位总是 1，因此该位不需显示表示出来，只需通过指数隐式给出。需要特别指出的是指数 E 需要要加上一个偏置值 $Bias$，转换成无符号的偏置指数 e，也就是说指数 E 要以移码的形式在存放计算机中。且 e、E 和 $Bias$ 三者的对应关系为 $e = E + Bias$，其中 $Bias = 2^{k-1} - 1$。

2、非规格化数

当指数 e 的位模式 $e_{k-1} \dots e_1 e_0$ 全为零 (即 $e = 0$) 时，浮点格式所表示的数是非规格化数。这种情况下，$E = 1 - Bais$，有效数字 $M = f = 0.f_{n-1} \dots f_1 f_0$，有效数字的前导有效位为 0。非规格化数的引一是提供了一种表示数值 0 的方法，二是可用来表示那些非常接近于 0.0 的数。

3、特殊数

当指数 e 的位模式 $e_{k-1} \dots e_1 e_0$ 全为 1 时，小数 f 的位模式 $f_{n-1} \dots f_1 f_0$ 全为 0(即 $f = 0$) 时，该浮点格式所表示的值表示无穷，$s = 0$ 时是 $+\infty$，$s = 1$ 时是 $-\infty$。当指数 e 的位模式 $e_{k-1} \dots e_1 e_0$ 全为 1 时，小数 f 的位模式 $f_{n-1} \dots f_1 f_0$ 不为 $0(f_{n-1}, \dots, f_1, f_0$ 至少有一个非零即 $f \ne 0)$ 时，该浮点格式所表示的值被称为 NaN（Not a Number)。浮点数中的特殊值主要用于特殊情况或者错误的处理，比如，在程序对一个负数进行开平方时，将返回一个 NaN 值用于标记这种错误。如果没有这样的特殊值，对于此类错误只能粗暴地终止计算。

四、 IEEE 754 标准浮点存储格式

浮点格式是一种数据结构，用于指定包含浮点数的字段、及这些字段的布局和其算术解释。浮点存储格式指定如何将浮点格式存储在内存中。IEEE 754 标准定义了这些格式，但具体选择哪种存储格式由实现工具决定。在 C/C++ 语言中，定义了`float`(单精度)、`double`(双精度)、`long double`(双精度扩展) 三种浮点数数据类型.

1、单精度格式

IEEE 754 标准单精度格式由 1 位符号 s，8 位偏置指数 e，23 位小数 f 三个字段组成，共占 32 位，4 个字节，其内存的存储结构如图 1所示。

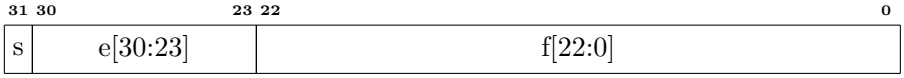

图 1　单精度浮点数的内存结构

在 23 位小数 f 中，第 0 位是最低有效位 LSB(the least significant bit)，第 22 位是最高有效位 MSB(the most significant bit)。

在 8 位指数 e 中，第 23 位是指数的最低有效位 LSB，第 30 位是最高有效位 MSB。

第 31 位的符号位，s 为 0 表示正数，为 1 则表示负数。

2、双精度格式

IEEE 754 标准双精度格式也由三部分构成，分别是 1 位符号 s，11 位偏置指数 e，52 位小数 f，共占 64 位，8 个字节。在 Intel x86 结构计算机中，其内存存储结构如图 2所示。

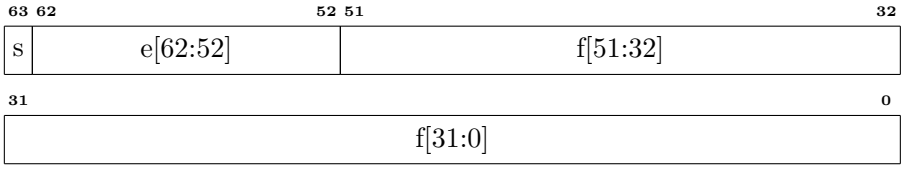

图 2　Intel x89 结构双精度浮点数的内存结构

在图 2中，f[31:0] 存放小数 f 的低 32 位，其中第 0 位存放整个小数 f 的最低有效位 LSB，第 31 位存放小数 f 的低 32 位的最高有效位 MSB。

在另外的 32 位里，第 32 到 51 位，即 f[51:32]，存放小数 f 的高 20 位，其中第 32 位存放这 20 位最高有效数中的最低有效位 LSB，第 51 位存放整个小数 f 的最高有效位 MSB。

第 52 到 62 位，即 e[62:52]，存放 11 位的偏置指数 e，其中第 52 位存放偏置指数的最低有效位 LSB，第 62 位存放最高有效位 MSB。

最高位，第 63 位存放符号位 s，s 为 0 表示正数，为 1 则表示负数。

在 Intel x86 结构中，数据存放采用小端法 (little endian)，故较低地址的 32 位的字中存放小数 f 的 32 位最低有效位 f[31:0] 位。而在在 SPARC 结构计算机中，因其数据存放采用大端法 (big endian)，故较高地址的 32 位字中存放小数 f 的 32 位最低有效位 f[63:32] 位，其内存存储结构如图 3所示。

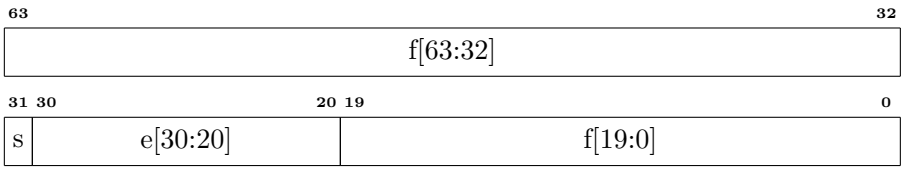

图 3　SPARC 结构双精度浮点数的内存结构

3、双精度扩展格式 (SPARC 结构)

SPARC 结构的 4 倍精度浮点环境符合 IEEE 754 标准关于扩展双精度格式的定义，共 128 位，占 16 个连续字节，仍由三部分构成，分别是 1 位符号 s、15 位偏置指数 e 和 112 位小数 f。将这 16 个连续的字节整体看作一个 128 位的字，进行重新编号，其内存存储结构如图 4所示。

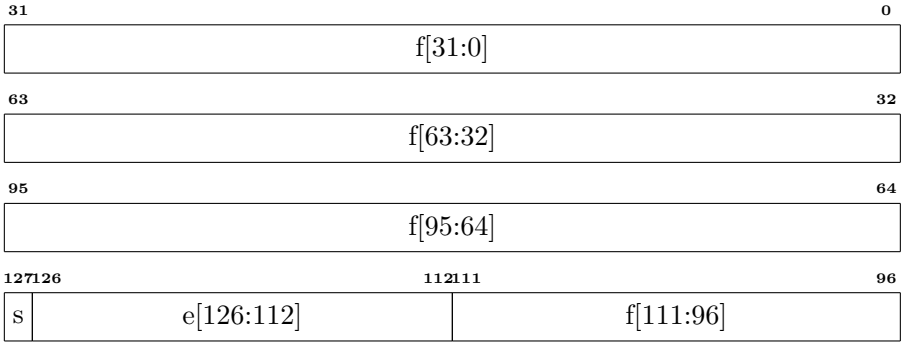

图 4　SPARC 结构双精度扩展浮点数的内存结构

地址最高的 32 位字包含小数的 32 位最低有效位，用 f[31:0] 表示。紧邻的两个 32 位字分别包含 f[63:32] 和 f[95:64]。下面的 96:111 位包含小数的 16 位最高有效位 f[111:96]，其中第 96 位是这 16 位的最低有效位，而第 111 位是整个小数的最高有效位。112:126 位包含 15 位偏置指数 e，其中第 112 位是该偏置指数的最低有效位，而第 126 位是最高有效位；第 127 位包含符号位 s。

4、双精度扩展格式 (x86)

x86 结构的双精度扩展格式符合 IEEE 754 标准关于扩展双精度格式的定义，共 80 位，占 12 个连续字节，由四部分构成，分别是 1 位符号 s、15 位偏置指数 e、1 位显式前导有效位 (explicit leading significand bit)j 和 63 位小数 f。将这 12 个连续的字节整体看作一个 96 位的字，进行重新编号，其内存存储结构如图 5所示。

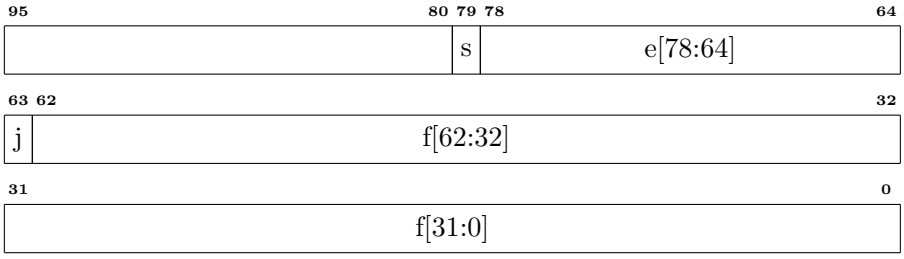

图 5　x86 结构双精度扩展浮点数的内存结构

在 Intel 结构系计算机中，这些字段依次存放在十个连续的字节中。但是，由于 Intel ABI 的规定，存储 x86 结构双精度扩展浮点数需要 12 个连续的字节，其中，最高 2 个字节的 16 位未被使用。

五、使用浮点数时的注意事项

由以上分析可知,浮点数在计算机内部总是一个近似值,无论采用`float`、`double`还是`long double`类型，都只能在一定的精度下保存浮点数。因此，有限的精度就引发了浮点数值使用时的**陷阱**。在使用浮点数时，要相当小心。

1、交换定律不适用浮点数

例如，有如下代码：

```c
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    float x = 1.0 / 3.0;
    float y = 1.0 / 6.0;
    float z = 1.0 / 7.0;

    if (x * y / z != y * x / z)
    {
        printf("Not equal!\n");
    }

    return 0;
}
```

其结果输出为："Not equal!"，也就是说 `x*y/z` 不等于 `y*x/z`。而对于整数来说，如果不发生溢出的情况下，`x*y/z` 是等于 `x*(y/z)`。

2、计算顺序会影响结果

浮点数的计算顺序对结果也会有影响，例如，有如下代码：

```c
#include<stdio.h>

int main(void)
{
    float  score1 = 90.5;
    float  score2 = 80;
    float  score3 = 70;
    float  score4 = 89;
    float  score5 = 84.6;

    float  sumsco = 0.0, avesco = 0.0;

    sumsco = score1 + score2 + score3 + score4 + score5; /* 累加总成绩 */
    avesco =  sumsco / 5;           /* 计算平均成绩 */
    printf("%f\n", avesco); /* 输出成绩 */

    avesco = score1 / 5 + score2 / 5 + score3 / 5 + score4 / 5 + score5 / 5; /* 累加总成绩 */
    printf("%f\n", avesco); /* 输出成绩 */

```

```
    return 0;
}
```

其结果输出分别为：“82.820000” 和 “82.819992”，显然，这是由于计算顺序造成的误差。

3、避免对两个实数做是否相等的判断

由于实数在内存中的存储误差，因此，可能出现在理论上应该**相等**的两个数，用计算机却判断它们为 **不相等**，例如 `x = 0.1`，而 `x * 9`却不等于 0.9，关系表达式 `x * 9 == 0.9`的值为假，例如代码：

```c
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    float x = 0.1;

    if (x * 9 == 0.9)
    {
        printf("x * 9 is 0.9!\n");
    }

    return 0;
}
```

的结果是不进入判断分支。

如果想判断 x 是否等于 y, 应改写为：

```
fabs(x - y) < epsilon //epsilon 被赋为一个选定的值来控制接近度
```

只要小于 epsilon，例如10e-5，就认为 x 和 y 足够地接近，可以近似地认为相等。

例如代码：

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

int main(void)
{
    float x = 0.1;
    float epsilon = 10e-5;

    //epsilon 被赋为一个选定的值来控制接近度
    if(fabs(x * 9 - 0.9) < epsilon)
    {
        printf("x * 9 is 0.9!\n");
```

```
    }

    return 0;
}
```

的输出结果是：x * 9 is 0.9!。

如果 x 或 y 的值比较大（如约等于 10^{30}），则 x - y 的差可能大于 10^{-5}，因此，可用相对误差，即 fabs((x - y) / x) < 10e-5，当此关系表达式的值为真时，x 和 y 的相对误差小于百万分之一。

4、慎用浮点数作为循环变量

C 语言中的循环变量可以用浮点数，但是使用浮点数作为循环变量一定要慎重对待，由于浮点数的误差，可能会使循环次数达不到预定的次数而导致程序出现**逻辑错误**。例如，有如下代码：

```c
#include <stdlib.h>
#include <stdio.h>

int main()
{
    float i, sum;
    int count = 0;

    for(i = 0.1, sum = 0.0; i <= 0.9; i += 0.1)
    {
        sum += i;
        count++;
    }

    printf("%d\n", count);
    printf("sum= %lf\n", sum);

    return 0;
}
```

其循环次数统计结果为 8 次，计算结果为 3.600000，这显然是错误的结果。

5、避免数量级相差很大的数直接加减

由于float、double或是long double类型变量是用有限的存储单元存储的，因此能提供的有效数字总是有限的，在有效位以外的数字将被舍去，因此，将一个很大的数和一个很小的数直接相加或相减，否则会丢失小的数（**淹没了小的数**），如程序：

```c
#include <stdio.h>

int main(void)
{
    float a = 987654321;
```

```
    float b = 987.654322;
    float c;

    c = a + b;

    printf("%f\n", c);

    return 0;
}
```

其运行结果为：987655296.000000。

另外，在对符号相同的两个数做减法或者符号不同的两个数做加法时，结果的精度可能比所用的浮点格式所能支持的精度要小。

建议对数量级接近的浮点数进行加减运算。

6、浮点数的乘除运算

在进行一系列涉及到加法，减法，乘法，除法的计算时，**尽量先做乘法与除法**。例如，对于 `x * (y + z)` 运算，应该使用 `x * y + x * z` 进行运算。

在对一组数做乘法或除法时，尽量对数量级相对一样的数做乘法或除法。

7、尽量使用double型以提高精度

为提高精度，建议使用double类型的数据，默认情况下，浮点数常量都是被看作是double类型的常量。例如，将循环变量及结果改为double类型，则代码：

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

int main()
{
    double i, sum;
    int count = 0;

    for(i = 0.1, sum = 0.0; i <= 0.9; i += 0.1)
    {
        sum += i;
        count++;
    }

    printf("%d\n", count);
    printf("sum= %lf\n", sum);

    return 0;
}
```

的循环次数统计结果为 9 次，计算结果为 4.500000，结果正确。

虽然本例结果是正确的，但仍然要明确的是浮点数不可以用作循环控制变量！

8、浮点数的特殊数

在 IEEE 754 标准浮点数中，有两个特殊数：

1）infinite 无限

2）NaN 即 Not a Number

如代码：

```c
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    float x = 1/0.0;

    printf("x is %f\n", x);

    x = 0/0.0;

    printf("x is %f\n", x);


    return 0;
}
```

的输出结果如图 6所示。

图 6　程序输出 inf 和-nan

当 1 除以 0.0 时，我们得到的是 infinite，而是用 0 除以 0.0 时，得到的就是 NaN。

在下面的代码中，调用 `scanf()` 接收用户输入的浮点数，值得注意的是，可以输入 `inf` 和 `nan`，`scanf()` 能够接收这两种浮点数的特殊数。

```c
#include <stdlib.h>
#include <stdio.h>

int main(void)
```

```
{
    float x;

    scanf("%f", &x);
    printf("x is %f\n", x);

    return 0;
}
```

可以使用 `isinf()` 或 `isnan()`[1]对浮点数进行判断，以确定其是不是特殊数。

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main()
{
    float x;

    scanf("%f", &x);

    if (isinf(x))
    {
        printf("It's infinite\n");
    }
    if (isnan(x))
    {
        printf("It's NaN\n");
    }

    return 0;
}
```

六、结论

现实中，不可避免的要进行各种各样的浮点数运算，但由于 C 语言中采用的 IEEE 754 标准浮点数类型，无论是哪种方式，其计算机中的表达总是不精确的，这种有限的精度一定会引发了浮点数值使用时的**陷阱**。因此，在使用浮点数时，要相当小心。这一结论同样也适用于后续 Java、C# 等其它计算机语言中浮点数的处理。

编程无小事，细节决定成败，期望大家“静心”学习和理解计算机世界的“精”与“简”。

[1]这两个函数属于 **C99** 以后的 **math.h** 中的函数。