深入剖析浮点存储及其运算规则

Email：vincent040.lin@gmail.com

blog：http://blog.csdn.net/vincent040

浮点数是用计算机表示的实数，用来表示那些介于整数之间的数值，比如 1.2 或者 3.14 等。在书写浮点常量时，可以使用小数点来与整型加以区分，比如 3.0 是浮点常量而 3 则是整型常量，也可以用科学计数法来表示浮点常量，比如 1.23e4，这个数代表了 1.23×10^5，其中字母 e 不区分大小写，紧跟其后的幂可正可负。科学计数法可以更方便地表示有效位数很多的数值。

我们还可以用以下关键字来定义浮点变量：

类型	Macintosh	Linux	Windows	ANSI C 规定的最小值
float	6 位	6 位	6 位	6 位
	-37 到 38	-37 到 38	-37 到 38	-37 到 37
double	18 位	15 位	15 位	10 位
	-4931 到 4932	-307 到 308	-307 到 308	-37 到 37
long double	18 位	18 位	18 位	10 位
	-4931 到 4932	-4931 到 4932	-4931 到 4932	-37 到 37

表中每种数据类型中，上面的位数代表有效数字位数，下面的数字范围代表指数（以 10 为基数）的范围。

以下语句定义了一个单精度浮点数和一个双精度浮点数并进行了初始化：

```
float  f;
double  lf = 1.23;
```

对于浮点类型数据，首先我们需要明白的一点是：浮点数和整型数的编码方式是很不一样的，IEEE 浮点标准采用 $V = (-1)^s\times M\times 2^E$ 的形式来表示一个数，其中符号 s 决定是负数(s=1)还是正数(s=0)，由 1 位符号位表示。有效数 M 是一个二进制小数，它的范围在 1~2-ε 之间（当指数域 E 既不全为 0 也不全为 1，即浮点数为规格化值时。ε 为有效数 M 的精度误差，比如当有效数为 23 位时，ε 为 2^{-24}），或者在 0~1-ε 之间（当指数域全为 0，即浮点数为非规格化值时），由 23 位或 52 位的小数域表示。指数 E 是 2 的幂，可正可负，它的作用是对浮点数加权，由 8 位或者 11 位的指数域表示。

下面我们来详细地剖析 IEEE 浮点数据表示，相信在认真研读完这一章节之后，你对浮点数的存储将会有非常清晰的认识。不用担心文中会充满数学家才会考虑的算式和符号，虽然大多数人认为 IEEE 浮点格式晦涩难懂，但理解了其小而一致的定义原则之后，相信你能感觉到它的优雅和温顺，以下的叙述将尽量用浅显易懂的语言和例子让大家心情舒畅地了解浮点存储的内幕。

以 32 位浮点数为例，其存储器的内部情况是这样的：

从上述公式 $V = (-1)^s \times M \times 2^E$ 可以计算出一个浮点数具体的数值，要理解这三个数据域是如何被解释的，我们需要知道：浮点数的编码根据指数域的不同取值表示被分成三种情形：

1、规格化值。

当指数域不全为 0 且不全为 1 的时候即为这种情形，这是最常见的情况。这时有效数域被解释为小数值 f，且 $0 \leqslant f < 1$，其二进制表示为 $0.f_{n-1}f_{n-2}\ldots f_1f_0$，而有效数 M 被解释为 M=1+f。同时指数域被解释为偏置形式的有符号数，指数 E 被定义为 E=e-Bias，其中 e 就是指数域的二进制表示，Bias 是一个等于 $2^{k-1}-1$ 的偏置值(k 为指数域位数，对于 32 位浮点数而言是 8，对于 64 位浮点数而言是 11)。我们可以调整指数域 E 来使得有效数 M 的范围在 $1 \leqslant M < 2$ 之间，也就可以始终使得 M 的第一位是 1，从而也没有必要在有效数域中显式地表示它了。

我们来做一个透视浮点数存储的练习，以此来更好地理解以上内容。例如浮点数 12345.0，其二进制表示为 $(1.1000000111001)_2 \times 2^{13}$，显然符号位 S 应该为 0，而指数域部分 E 应该根据公式 E= e-Bias 计算，e 就是我们想要的内部存储表示，Bias 在 32 位单精度浮点中的值为 127，因此 e=E+Bias=13+127=140，用二进制表示即为 $(100011001)_2$，而小数域就是在其二进制表示的基础上减去最高位的 1 即可，即 0.1000000111001，补满 23 位小数位，即可得到 0.10000001110010000000000，存储时略去前面的的整数部分和小数点 ，因此浮点数 12345.0 的 IEEE 浮点表示为：

0	1000 1100	100 0000 1110 0100 0000 0000

2、非规格化值。

当指数域全为 0 时属于这种情形。非规格化编码用于表示非常接近 0.0 的数值以及 0 本身（因为规格化编码时有效数 M 始终大于等于 1），我们会看到，由于在非规格化编码时指数域 E 被解释为一个定值：E = 1-Bias = -126（而不是规格化时 E = e-Bias），使得规格化数值和非规格化数值之间实现了平滑过渡。另外，此时有效数 M 解释为 M=f，对比上面所讲的规格化编码，此时的有效数 M 就是小数域的值，不包含开头的 1。

举个例子，编码为如下情形的浮点数就是一个非规格化的样本：

0	0000 0000	000 0000 0000 0000 0000 0001

这个数值的大小，就应该被解释为 $(-1)^0 \times (0.00000000000000000000001)_2 \times 2^{1-Bias}$，即 $2^{-23} \times 2^{-126} = 2^{-149}$，转成十进制表示大约等于 1.4×10^{-45}，实际上这就是单精度浮点数所能表达的最小正数了。

以此类推，规格化值和非规格化值所能表达的非负数值范围如下所示：

	指数域	小数域	32 位单精度浮点数	
			值	十进制
0	0000 0000	000 0000 0000 0000 0000 0000	0	0.0
最小非规格化数	0000 0000	000 0000 0000 0000 0000 0001	$2^{-23} \times 2^{-126}$	$\approx 1.4 \times 10^{-45}$
最大非规格化数	0000 0000	111 1111 1111 1111 1111 1111	$(1-\varepsilon) \times 2^{-126}$	$\approx 1.2 \times 10^{-38}$
最小规格化数	0000 0001	000 0000 0000 0000 0000 0000	1×2^{-126}	$\approx 1.2 \times 10^{-38}$
最大规格化数	1111 1110	111 1111 1111 1111 1111 1111	$(2-\varepsilon) \times 2^{127}$	$\approx 3.4 \times 10^{38}$

1	0111 1111	000 0000 0000 0000 0000 0000	1×2^0	1.0

表 2.1.3　规格化和非规格化值非负数值范围

从上表中可以看出，由于在非规格化中将指数域数值 E 定义为 E = 1-Bias，实现了其最大值与规格化的最小值平滑的过渡（最大的非规格化数为$(1-\varepsilon)\times2^{-126}$，只比最小的规格化数 2^{-126} 小一点点，ε 为 2^{-24}）。

2、特殊数值。

当指数域全为 1 时属于这种情形。此时，如果小数域全为 0 且符号域 S=0，则表示正无穷+∞，如果小数域全为 0 且符号域 S=1，则表示负无穷-∞。如果小数域不全为 0 时，浮点数将被解释为 NaN，即不是一个数(Not a Number)。比如计算负数平方根或者处理未初始化数据时。

以下是理清各种数据之间关系的总结：

1. 浮点数值 $V = (-1)^s\times M\times2^E$。
2. 在 32 位和 64 位浮点数中，符号域 s 均为 1 位，小数域位数 n 分别为 23 位和 52 位，指数域位数 k 分别为 8 位和 11 位。
3. 对于规格化编码，有效数 M = 1+f，指数 E = e-Bias（e 即为 k 位的指数域二进制数据，对于 32 位浮点数而言 e 的范围是 1~254，此时 E 的范围是-126~127）。
4. 对于非规格化编码，有效数 M = f，指数 E = 1-Bias（这是一个常量）。
5. Bias 为偏置值，Bias = 2^{k-1}-1，k 即为指数域位数，在 32 位和 64 位浮点数中 k 分别为 8 和 11。
6. f 为小数域的二进制表示值，即 n 位的小数域 $f_{n-1}f_{n-2}...f_1f_0$ 将被解释为 $f=(0.f_{n-1}f_{n-2}...f_1f_0)_2$。

有了以上的背景知识之后，我们就可以更从容地分析浮点运算了。毕竟我们不是数学家，学习浮点数不是为了科学研究，更多地是从实用主义的角度出发，是为了要写出更好更可靠的代码。

首先要说明的是，浮点运算是不遵循结合性的，也就是说(a+b)-c 和 a+(b-c)可能会得出不一样的结果（比如(1.23+4.56e20)-4.56e20 = 0，但 1.23+(4.56e20-4.56e20) = 1.23）。这是由浮点数的表示方法限制的，浮点数的范围和精度有限，因此它只能近似地表示实数运算。在两个浮点数进行运算的时候，它们之间会产生一种称之为“舍入”的行为，要理解清楚这种游戏规则，请仔细研读以下叙述。

假设有两个浮点数 f1 和 f2，其 IEEE 存储格式分别为如下：

s	$e_{k-1}e_{k-2}e_{k-3}...e_1e_0$	$f_{n-1}f_{n-2}f_{n-3}.....f_1f_0$

s’	$e'_{k-1}e'_{k-2}e'_{k-3}...e'_1e'_0$	$f'_{n-1}f'_{n-2}f'_{n-3}.....f'_1f'_0$

对于单精度浮点数而言 k=8，n=23，对于双精度浮点数而言 k=11，n=52。这个对于我们讨论舍入问题不是关键，权且就把 f 当成是 32 位的单精度浮点数即可。

如果现在要执行 f1 + f2，则有可能会发生舍入操作，具体情况如下：

1、取两数中指数域较大的一个 e，再取小数域为最小值 0000…0001。

2、令 $\delta = M\times2^E$，即$(0.0000...0001)_2\times2^{e-127}$，注意：这里 M 取的是非规格化值。

3、所有小于 $\delta/2$ 的数值都会被舍入。

下面的代码验证了以上推论：

```
#include <stdio.h>

int main(void)
{
    float   x = 0.1;
```

```
    float   a = x + 0.37252e-8;
    float   b = x + 0.37253e-8;

    if( x == a )
        printf("x == a|n");
    if( x == b )
        printf("x == b|n");

    return 0;
}
```

代码 2.1.3　float_accuracy.c

程序的运行结果是打印出了 x==a，但是不会打印 x==b。也就是说系统辨识不出来比 0.372529e-8 还小的值，区分不出来 x 与 a 的差别。其内幕如下：

单精度数据 x=0.1 的存储细节如下：

0	011 1101 1	100 1100 1100 1100 1100 1101

因此 δ/2 = (M×2^E) /2 = (2^{-23}×$2^{123-127}$)/2 = 2^{-28} ≈ 0.372529e-8。变量 a 与 b 都分别在 x 的基础上加了一个常量，但是由于加在 a 上的增量小于 δ/2，在此精度范围内该增量无法被辨识。由此可见，从数学角度上看明明是三个不相等的实数，但是由于计算机本身特性的限制会导致程序运行结果跟预料的不符，这就要求我们必须对浮点数运算的内部实现非常了解，不能想当然地写出似是而非的代码。

同样地，我们可以自己再做一个练习，比如有一个单精度浮点数 f = 10e18，用代码 2.1.2 所示的方法查看其二进制存储细节，然后计算出 δ/2 ≈ 0.549755e12，也就是说对于 f 而言精度小于 0.549755e12 的数据都将会被舍入，所有跟 f 的差值小于此精度的数据都将会被认为等于 f。

从上述叙述中我们得出一个重要的结论：对于每个不同的浮点数，都有相应的最小可辨识精度（即 δ/2），此最小可辨识精度随着该浮点数的数值变化而变化，具体究竟是多少要像上面那样分析该浮点数的二进制存储内部细节，找到其指数域之后才能确定，我们根据这个最小可辨识精度才能明确判定代码中所有对此浮点数的运算是否有效，否则可能会由于舍入的问题存在而在逻辑上存在歧义。网上有文章简单地用一个固定的最小精度 EPISION 来判断两个浮点数是否相等的方法是不严谨的。我们以后在程序中需要进行高精度浮点比较（记住，精度是个相对的概念）的时候，需要先计算出其相应的最小可辨识精度 δ/2，在此基础上再进行操作和判断。如果当前浮点数精度不能满足工程的需要，你或许需要考虑使用更高精度的浮点数据类型。