C 语言浮点数运算

有些 C 语言书上说 float 型的有效位数是 6~7 位，为什么不是 6 位或者 7 位？而是一个变化的 6~7 位？

浮点数在内存中是如何存放的？

float 浮点数要比同为 4 字节的 int 定点数表示的范围大的多，那么是否可以使用浮点数替代定点数？

为什么 float 型浮点数 9.87654321 > 9.87654322 不成立？为何 10.2 - 9 的结果不是 1.2，而是 1.1999998？为何 987654321 + 987.654322 的结果不是 987655308.654322？

如何才能精确比较浮点数真实的大小？

看完本文档，你将会得到答案!

在论坛上或 QQ 群中有时会看到新同学问一些有关浮点数运算的问题，经常是走了错误的方向，但苦于交流方式不方便，无法为其详细说明，在此，我将我所掌握的一些知识加以整理写出来，希望对大家能有所帮助。更多案例请访问我的博客 blog.sina.com.cn/ifreecoding。

我主要是从事底层软件开发的，最开始写驱动程序，后来也做一些简单的业务层软件，在我所涉及的工作范围内，我使用的都是定点数，而且 90%以上都是无符号定点数，在我印象中并没有使用过浮点数，即使做过一个专门使用 DSP 来处理信号的项目，也只是使用了无符号定点数，我将在另一篇案例《C 语言使用定点数代替浮点数计算》里介绍定点数处理简单的浮点数的方法，这也是在底层驱动中常使用的方法。

C 语言浮点数

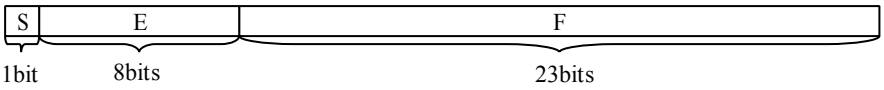
C 语言标准 C89 里规定了 3 种浮点数，float 型、double 型和 long double 型，其中 float 型占 4 个字节，double 型占 8 个字节，long double 型长度要大于等于 double 型，本文档将以 float 型为例进行介绍，double 型和 long double 型只是比 float 型位数长，原理都是一样的。

float 型可以表示的范围是$-3.402823466e^{38}$~$3.402823466e^{38}$，而作为同为 4 个字节的定点数却只能表示-2147483648~2147483647 的范围，使用同样的内存空间，浮点数却能比定点数表示大得多的范围，这是不是太神奇了？既然浮点数能表示这么大的范围，那么我们为何不使用浮点数来代替定点数呢？

先不说浮点数实现起来比较复杂，有些处理器还专门配置了硬件浮点运算单元用于浮点运算，主要原因是浮点数根本就无法取代定点数，因为精度问题。鱼和熊掌不可兼得，浮点数表示了非常大的范围，但它失去了非常准的精度。在说明精度问题前，我们先了解一下浮点数的格式。

ANSI/IEEE Std 754-1985 标准

IEEE 754 是最广泛使用的二进制浮点数算术标准，被许多 CPU 与浮点运算器所采用。IEEE 754 规定了多种表示浮点数值的方式，在本文档里只介绍 32bits 的 float 浮点类型。它被分为 3 个部分，分别是符号位 S（sign bit）、指数偏差 E（exponent bias）和小数部分 F（fraction）。



其中 S 位占 1bit，为 bit31。S 位为 0 代表浮点数是正数，S 位为 1 代表浮点数是负数，比如说 0x449A522C 的 S 位为 0，表示这是一个正数，0x849A522C 的 S 位为 1，表示这是一个负数。

E 位占 8bits，为 bit23~bit30。E 位代表 2 的 N 次方，但需要减去 127，比如说 E 位为 87，那么 E 位的值为 $2^{(87-127)}$ =9.094947017729282379150390625e-13。

F 位占 23bits，为 bit0~bit22。F 位是小数点后面的位数，其中 bit22 是 2^{-1}=0.5，bit21 是 2^{-2}=0.25，以此类推，bit0 为 2^{-23}=0.00000011920928955078125。但 F 位里隐藏了一个 1，也就是说 F 位所表示的值是 1+（F 位 bit22~bit0 所表示的数值），比如说 F 位是 0b10100000000000000000001，只有 bit22、bit20 和 bit0 为 1，那么 F 位的值为 $1+(2^{-1}+2^{-3}+2^{-23})$，为 1.625000119209289550781 25。

综上所述，从二进制数换算到浮点数的公式为：$(-1)^S \times 2^{E-127} \times (1+F)$。但还有几个特殊的情形：

- 若 E 位为 0 并且 F 位也为 0 时表示浮点数 0，此时浮点数受 S 位影响，表现出+0 和-0 两种 0，但数值是相等的。比如二进制数 0x00000000 表示+0，二进制数 0x80000000 表示-0。
- 若 E 位为 0 并且 F 位不为 0 时浮点数为$(-1)^S \times 2^{-126} \times F$，注意，E 位的指数是-126，而不是 0-127=-127，而且 F 位是 0.xx 格式而不是 1.xx 格式，比如 0x00000001 的浮点数为 $2^{-126} \times 2^{-23}$=1.401298464324817070923729583289 9e-45，而不是 $20^{-121} \times (1+2^{-23})$。一旦 E 为不为 0，从 0 变为 1，不是增加 2 倍的关系，因为公式改变了。
- 若 E 位为 255 并且 F 位不为 0 时表示非数值，也就是说是非法数，例如 0x7F800001。
- 若 E 位为 255 并且 F 位为 0 时表示无穷大的数，此时浮点数受 S 位影响，例如 0x7F800000 表示正无穷大，0xFF800000 表示负无穷大。当我们使用 1 个数除以 0 时，结果将被记作 0x7F800000。

浮点型在多个处理器间通信时，传递的数值是它的二进制数，比如说 1234.5678 这个浮点数的二进制数是 0x449A522B，如果使用串口发送的话，就会发现串口里发送的是 0x44、0x9A、0x52 和 0x2B 这 4 个数（发送的顺序也可能是逆序，这与约定的字节序有关，与浮点格式无关），接收端接收到这 4 个数字后再组合成 0x449A522B，按照 IEEE 754 的定义被解析成 1234.5678，这样就实现浮点数通信了。如果两个处理器所使用的浮点数规则不同，则无法传递浮点数。

浮点数的换算

下面来看看浮点数与二进制数如何转换。

例 1，二进制数换算成浮点数：

假如在内存中有一个二进制数为 0x449A522C，先将十六进制转换成二进制，如下：

0100 0100 1001 1010 0101 0010 0010 1100

按照 SEF 的格式分段，如下：

0 10001001 00110100101001000101100

这个数值不是特殊的情形，可以按照公式$(-1)^S\times2^{E-127}\times(1+F)$转换。S 位的值为$(-1)^0=1$，E 位的值为$213^{7-127}=1024$。F 位的值为$1+2^{-3}+2^{-4}+2^{-6}+2^{-9}+2^{-11}+2^{-14}+2^{-18}+2^{-20}+2^{-21}=1.20563268661499023437$5。最终结果为$1\times1024\times1.20563268661499023437$5=1234.56787109375。

其中 F 位比较长，使用二进制方式转换比较麻烦，也可以先转换成十六进制再计算，转换为十六进制如下：

0011 0100 1010 0100 0101 1000

0x3 0x4 0xA 0x4 0x5 0x8

F 位为 23bits，需要在最后补一个 0 凑成 24bits，共 6 个十六进制数。F 位的值为$1+3\times16^{-1}+4\times16^{-2}+10\times16^{-3}+4\times16^{-4}+5\times16^{-5}+8\times16^{-6}=1.20563268661499023437$5，与上面使用二进制方法得到的结果相同。

例 2，浮点数换算成二进制数：

下面我们将$-987.654e^{30}$换算成二进制数。我们先不看符号位，将$987.654e^{30}$归一化为整数部分为 1 的形式，也就是写作$987.654e^{30}=2^{E-127}\times$（1+F）的标准形式，其中$E=\log(987.654e^{30})/\log2+127=109.6+127$，取 E 位的整数值为 109+127=236，再求$F=987.654e^{30}/2^{236-127}-1=0.52172193$，这个小数位数保留 8 位就够了，已经超出了 7 位的精度。然后我们求小数部分的二进制数，这个转换就没啥好说的了，依次减去 2 的幂，从2^{-1}一直到2^{-23}，够减的位置 1，不够减的位置 0，例如，2^{-1}为 0.5，0.52172193-0.5=0.02172193，F 位的 bit22 置 1，2^{-2}为 0.25，0.02172193 不够减，F 位的 bit21 置 0，2^{-3}为 0.125，0.02172193 不够减，F 位的 bit20 置 0，2^{-4}为 0.0625，0.02172193 不够减，F 位的 bit19 置 0……，一直算到 F 位的 bit0，这样就得到 F 位的数值。

如果觉得使用二进制方式转换太麻烦的话也可以使用十六进制进行转换。16^{-1}为 0.0625，0.52172193/0.0625=8.3，说明够减 8 个，记做 0x8，0.52172193-0.0625×8=0.02172193，16^{-2}为 0.00390625，0.02172193/0.00390625=5.6，说明够减 5 个，加上刚才的 0x8 记做 0x85，以此类推：

	16 的-N 次幂	被减数	十六进制数	减后的数
1	0.0625	0.52172193	0x8	0.02172193
2	0.00390625	0.02172193	0x85	0.00219068
3	0.000244140625	0.00219068	0x858	0.000237555
4	0.0000152587890625	0.000237555	0x858F	0.0000086731640625
5	0.00000095367431640625	0.0000086731640625	0x858F9	0.000000090095214843 75
6	0.0000000596046447753906 25	0.000000090095214843 75	0x858F91	

一直凑够 23bits，也就是 6 个十六进制，得到 0x858F91，换算成二进制如下所示：

1000 0101 1000 1111 1001 0001

由于只有 23bits 有效，因此需要去掉最后一个 bit，二进制本着 0 舍 1 入的原则，变成

1000 0101 1000 1111 1001 001

最后需要再补上前面的 S 位和 E 位。由于是负数，S 位为 1。E 位为 236，二进制形式

为 1110 1100，将 S、E、F 位组合在一起就形成了：

1 1110 1100 1000 0101 1000 1111 1001 001

从左边最高位开始，4 个一组合并成十六进制：

1111 0110 0100 0010 1100 0111 1100 1001

换算成十六进制为：

0xF 0x6 0x4 0x2 0xC 0x7 0xC 0x9

综上所述，$-987.654e^{30}$ 换算成二进制数为 0xF642C7C9。

浮点数的精度

在前面的讲解中可以看到 1.xx 这个数量级的最小数是 2^{-23}，对应的十进制数值为 1.000000119209289550781 25，可以精确表示到小数点后 23 位，但有些 C 语言书上却说 float 型的有效位只有 6~7 位，这是为什么？

这是因为二进制小数与十进制小数没有完全一一对应的关系，二进制小数对于十进制小数来说相当于是离散的而不是连续的，我们来看看下面这些数字：

二进制小数	十进制小数
2^{-23}	1.00000011920928955078125
2^{-22}	1.00000023841857910156 25
2^{-21}	1.00000047683715820312 5
2^{-20}	1.00000095367431640625
2^{-19}	1.0000019073486328125
2^{-18}	1.000003814697265625

不看 S 位和 E 位，只看 F 位，上表列出了 1.xx 这个数量级的 6 个最小幂的二进制小数，对应的十进制在上表的右边，可以看到使用二进制所能表示的最小小数是 1.000000119209289550781 25，接下来是 1.000000238418579101562 5，这两个数之间是有间隔的，如果想用二进制小数来表示 8 位有效数（只算小数部分，小数点前面的 1 是隐藏的默认值）1.00000002、1.00000003、1.00000004...这些数是无法办到的，而 7 位有效数 1.0000001 可以用 2^{-23} 来表示，1.0000002 可以用 2^{-22} 来表示，1.0000003 可以用 $2^{-23}+2^{-22}$ 来表示。从这个角度来看，float 型所能精确表示的位数只有 7 位，7 位之后的数虽然也是精确表示的，但却无法表示任意一个想表示的数值。

但还是有一些例外的，比如说 7 位有效数 1.0000006 这个数就无法使用 F 位表示，二进制小数对于十进制小数来说相当于是离散的，刚好凑不出 1.0000006 这个数，从这点来看 float 型所能精确表示的位数只有 6 位。至于 5 位有效值的任何数都是可以使用 F 位相加组合出来的，即便是乘以 E 位的指数后也是可以准确表示出来的。

因此 float 型的有效位数是 6~7 位，但这个说法应该不是非常准确，准确来说应该是 6 位，C 语言的头文件中规定也是 6 位。

对于一个很大的数，比如说 1234567890，它是 F 位乘上 E 位的系数被放大了的，但它的有效位仍然是 F 位所能表示的 6 位有效数字。1234567890 对应的二进制数是 0x4E932C06，其中 F 位的数值为 1.149780986932373046875，E 位的数值为 2^{30}=1073741824，1073741824

×1.149780988693237304687 5=1234567936，对比 1234567890，也只有高 7 位是有效位，后 3 位是无效的。int 型定点数可以准确的表示 1234567890，而 float 浮点数则只能近似的表示 1234567890，精度问题决定了 float 型根本无法取代 int 型。

浮点数的比较

从上面的讨论可以看出，float 型的有效位数是 6 位，那么我们在用 float 型运算时就要注意了，来看下面这段程序：

```
#include <stdio.h>

int main(void)
{
    float a = 9.87654321;
    float b = 9.87654322;

    if(a > b)
    {
        printf("a > b\n");
    }
    else if(a == b)
    {
        printf("a == b\n");
    }
    else
    {
        printf("a < b\n");
    }

    return 0;
}
```

按照我们平时的经验来说这段程序应该走 a < b 的分支，但程序运行的结果却走了 a == b 的分支，原因就是 float 型的精度问题，float 型无法区分出小数点后的第 8 位数，在内存中，a 和 b 的二进制数都是 0x411E0652，因此就走了 a == b 的分支。

某些编译器在编译时会发现 a 和 b 的值超出了浮点数的精度，会产生一个告警，提示数据超过精度，但有些编译器则不会做任何提示。最可怕的是有一类编译器，调试窗口里显示的长度超出 float 型的精度，比如说 a 的值显示为 9.87654321，b 的值显示为 9.87654322，但在运行时，硬件可不管这套，硬件认为这 2 个数都是 0x411E0652，因此实际运行结果是 a == b 的分支。以前就遇到过一个同学在 QQ 群里问一个类似的问题，在调试窗口里明明写着 a 是 9.87654321，小于 b 的 9.87654322，但运行结果却是 a ==b。当时我给他说了半天也没能让他明白这个问题，希望他能有机会看到这个文档，希望他这次能够明白^_^。

由于精度这个问题的限制，我们在浮点数比较时就需要加一个可接受的精度条件来做判决，比如说上面的这个问题，如果我们认为精度在 0.00001 就足够了，那么 a - b 之差的绝对值只要小于 0.00001，我们就认为 a 和 b 的值是相等的，大于 0.00001 则认为不等，还要考虑到 a - b 正负等情况，因此可以将上面的程序改写为：

```
#include <stdio.h>

int main(void)
{
```

```
    float a = 9.87654321;
    float b = 9.87654322;

    if(a - b < -0.00001)
    {
        printf("a < b\n");
    }
    else if(a - b > 0.00001)
    {
        printf("a > b\n");
    }
    else
    {
        printf("a == b\n");
    }

    return 0;
}
```

例子中 a 和 b 之差的绝对值小于 0.00001，因此认为 a 和 b 是相等的，运行程序，也正确的打印了 a == b。

也许你会觉得费了这么大的劲，最后 2 个程序运行的结果还是一样的，这不是画蛇添足么？硬件已经自动考虑到精度问题了，为什么我们还要去再做一个精度的限定？这是因为我们在应用中的精度往往要低于硬件的 6 位精度。比如说我们使用 2 个 AD 采集 2V 的电压，并对这 2 个电压值做比较，一般要求精确到 0.1V 即可。实际情况下 AD 采集出来的数值都会围绕真实值有上下的波动，比如说 AD 的精度是 0.001V，我们采集出的电压数值就可能是 2.003V、2.001V、1.999V 等围绕 2V 波动的数值，如果我们在程序里不对精度加以限制就对这些数值做比较，就会发现几乎每次采集到的电压值都是不同的。在这种情况下我们就需要将精度设定为 0.1V，将上面例子中的 0.00001 改为 0.1，就可以正确的判断出每次采集的电压值都是相同的了。

在实际使用 AD 采样时可能并不需要使用浮点数，我一般都是使用定点数来代替浮点数进行处理的，请参考另一篇案例《C 语言使用定点数代替浮点数计算》。

下面我们再看一个例子：

```
#include <stdio.h>

int main(void)
{
    float a = 987654321;
    float b = 987654322;

    if(a - b < -0.00001)
    {
        printf("a < b\n");
    }
    else if(a - b > 0.00001)
    {
        printf("a > b\n");
    }
    else
    {
        printf("a == b\n");
```

```
    }

    return 0;
}
```

这个例子中的两个数都是很大的数，已经远远超过了 0.00001 的精度，运行结果是不是应该是 a < b？但程序运行的结果依然是 a == b。这是因为这个例子里的 a 和 b 并不是 1.xx 的数量级，我们将 a 和 b 进行归一化，都除以 900000000 就会发现 a = 1.09739369，b = 1.097393691，只是在第 9 位才出现不同，因此在 0.00001 这个精度下，这 2 个数还是相等的。换个角度来看，a 和 b 虽然是很大的数了，但 F 位仅能表示 23bits，有效值仅有 6 位，a 和 b 的大是因为 E 位的指数放大 F 位表现出来的，但有效值依然是 6 位。在内存中 a 和 b 的二进制数都是 0x4E6B79A3。其中 E 位为 156，$2^{156-127}$=536870912，F 位为 0xD6F346，F 位 1.xx 数量级的 1.83964955806732177734375 被 E 位放大了 536870912 倍，E 位如果算 7 位有效精度的话能精确到 0.0000001，乘以 536870912 已经被放大了 53 倍多，这就说明 a 和 b 的个位与十位已经不是有效位数了，所以程序运行的结果表现出 a == b 也是正常的。

由此可见，设定一个合理的精度是需要结合浮点数的数量级的，这看起来似乎比较难，毕竟在程序运行时十分精确的跟踪每个浮点数的数量级并不容易实现，但实际应用中需要比较的浮点数往往都会有其物理含义，例如上面电压的例子，因此，根据浮点数的物理含义，浮点数的精度还是比较好确定的。当然在一些复杂的数值运算过程中可能会存在非常复杂的情况，这时浮点数的精度问题就比较棘手了，所幸我所做的都是比较简单的东西，那些复杂的情况就不讨论了，我也没能力讨论^_^。

上面所说的都是同等数量级下的浮点数进行比较的情况，不同数量级的浮点数比较则没有这个限制，比如说 1.23456789 与 12.23456789 的比较，在 E 位已经足以区分大小了，因此 F 位的精度就没有必要再比较了。

浮点数的加减

二进制小数与十进制小数之间不存在一一对应的关系，因此某些十进制很整的加减法小数运算由二进制小数来实现就表现出了不整的情况，来看下面的例子：

```
#include <stdio.h>

int main(void)
{
    float a = 10.2;
    float b = 9;
    float c;

    c = a - b;

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

    return 0;
}
```

如果用十进制计算的话变量 c 应该为 1.2，在 Visual C++ 2010 环境下实验输出为 1.200000，但实际上 c 变量的值是 1.1999998，只不过是在输出时被四舍五入为 1.200000 罢了。在内存中 c 变量的二进制数是 0x3F999998，它对应的浮点数是 0.19999980926513671875。如果我们将 printf 函数%f 的格式改为%.7f 格式，就会看到 c 变量输出的值是 1.1999998。

两个数量级相差很大的数做加减运算时，数值小的浮点数会受精度限制而被忽略，看下面的例子：

```
#include <stdio.h>

int main(void)
{
    float a = 987654321;
    float b = 987.654322;
    float c;

    c = a + b;

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

    return 0;
}
```

Visual C++ 2010 上的计算结果为 987655296.000000，而实际的真实值为 987655308.654322，二进制值为 0x4E6B79B2 对应 987655296，就是 987655296.000000。可以看出有效值是 6 位，如果按四舍五入的话可以精确到 8 位，其中变量 b 贡献的有效数值只有 2 位。

```
987654321
+      987.654322
----------------
987655308.654322     真实值
987655296.000000     计算值
987655296            二进制值, 0x4E6B79B2
```

对于这种数量级相差很大的计算，计算结果会保证高位数有效，数量级小的数相对计算结果显的太小了，不能按自身 6 位的精度保持，而是需要按照计算结果的 6 位精度保持。

使用二进制数比较浮点数

下面我们从另一个方向探索一下浮点数的比较问题。

我们可以使用$(-1)^S \times 2^{E-127} \times (1+F)$这个公式来计算 IEEE 754 标准规定的浮点数，先抛开 S 位和那 4 种特殊的规定，只看 E 位和 F 位 $2^{E-127} \times (1+F)$，我们会发现 E 位和 F 位组成的数值具有单调递增性，也就是说任意一个浮点数 A 掩掉 S 位的数值 B = (A & 0x7FFFFFFF)是单调递增的，如果 A1 大于 A2，那么 B1 一定大于 B2，反之亦然，如果 B1 大于 B2，那么 A1 也一定大于 A2，这样的话我们就可以使用浮点数的二进制数来比较大小了。

看下面程序，使用联合体将浮点数转换成二进制数再做比较：

```
#include <stdio.h>

typedef union float_int
{
    float f;
    int i;
}FLOAT_INT;

int main(void)
{
    FLOAT_INT a;
    FLOAT_INT b;
```

```
    int ca;
    int cb;

    a.f = 9.87654321;
    b.f = 9.87654322;

    ca = a.i & 0x7FFFFFFF;
    cb = b.i & 0x7FFFFFFF;

    if(ca > cb)
    {
        printf("a > b\n");
    }
    else if(ca == cb)
    {
        printf("a == b\n");
    }
    else
    {
        printf("a < b\n");
    }

    return 0;
}
```

上面的程序使用联合体使浮点型和整型共享同一个内存空间，浮点型变量.f 输入浮点数，使用整型变量.i 就可以获取到.f 的二进制数，比较时利用.i 的 E 位和 F 位就可以判断浮点数的绝对大小了，这个判决的精度为硬件所支持的精度。

如果考虑到 S 位，情况会有些变化。S 位是符号位，0 正 1 负，与 int 型的符号位有一样的作用，并且都在 bit31。从这点来看，不对浮点数的二进制数进行(& 0x7FFFFFFF)的操作而是直接使用浮点数的二进制数来当做 int 型数做比较，那么浮点数的 S 位则正好可以充当 int 型数的符号位。两个比较的浮点数都是正数的情况就不用说了，上面的例子(& 0x7FFFFFFF)已经验证了。正浮点数与负浮点数比较的情况也没有问题，浮点数和 int 型数的符号位是兼容的，符号位就可以直接比较出大小，比如说-9.87654321 和 9.87654322 之间做比较，-9.87654321 的 bit31 是 1，9.87654322 的 bit31 是 0，从二进制 int 型数的角度来看，bit31 为 0 是正数，bit31 为 1 是负数，通过符号位就可以直接判断出大小。最后剩下两个负浮点数比较的情况了，这种情况存在问题，如果采用二进制 int 型数来比较浮点数的话，结果则正好相反，比如说-1.5 和-1.25 做比较，int 型数是用补码表示的，对于两个负数来说，补码的二进制数值越大则补码值也越大。-1.5 的补码是 0xBFC00000，-1.25 的补码是 0xBFA00000，从二进制角度来看 0xBFC00000>0xBFA00000，因此 int 的补码是 0xBFC00000>0xBFA00000，也就是-1077936128>-1080033280，如果使用 int 型来判断，就会得出-1.5＞-1.25 的结论，正好相反了。这样的话我们就需要对两个浮点数的符号位做一个判断，如果同为负数的话则需要将比较结果反一下，如下面程序：

```
#include <stdio.h>

typedef union float_int
{
    float f;
    int i;
}FLOAT_INT;
```

```
int main(void)
{
    FLOAT_INT a;
    FLOAT_INT b;

    a.f = -9.876543;
    b.f = -9.876542;

    if((a.i < 0) && (b.i < 0))
    {
        if(a.i < b.i)
        {
            printf("a > b\n");
        }
        else if(a.i == b.i)
        {
            printf("a == b\n");
        }
        else
        {
            printf("a < b\n");
        }
    }
    else
    {
        if(a.i > b.i)
        {
            printf("a > b\n");
        }
        else if(a.i == b.i)
        {
            printf("a == b\n");
        }
        else
        {
            printf("a < b\n");
        }
    }
}
```

如果再考虑 IEEE 754 标准定义的那几种特殊情况，问题变得又会复杂一些，比如说在运算过程中有±x / 0 的情况出现，那么结果就是一个±无穷大的数，还有可能遇到±0 等情况，这些问题在这里就不讨论，只要增加相应的条件分支就可以做出判断的。

使用二进制数比较浮点数的方法可以依据硬件精度判断出浮点数的真正大小，但实际使用过程中往往不是根据硬件精度做判断的，因此最好还是使用上面所介绍的加入精度的判断方法。

C 语言中有关浮点数的定义

C 语言对浮点数做了一些规定，下面是摘自 Visual C++ 2010 头文件 float.h 中有关 float 型的定义，如下：

```
#define FLT_DIG         6                   /* # of decimal digits of precision */
#define FLT_EPSILON     1.192092896e-07F    /* smallest such that 1.0+FLT_EPSILON != 1.0 */
```

```
#define FLT_GUARD       0
#define FLT_MANT_DIG    24                      /* # of bits in mantissa */
#define FLT_MAX         3.402823466e+38F        /* max value */
#define FLT_MAX_10_EXP  38                      /* max decimal exponent */
#define FLT_MAX_EXP     128                     /* max binary exponent */
#define FLT_MIN         1.175494351e-38F        /* min positive value */
#define FLT_MIN_10_EXP  (-37)                   /* min decimal exponent */
#define FLT_MIN_EXP     (-125)                  /* min binary exponent */
```

其中 FLT_DIG 定义了 float 型的十进制精度，是 6 位，与我们上面的讨论是一致的。

FLT_EPSILON 定义了 float 型在 1.xx 数量级下的最小精度，1.xx 数量级下判断浮点数是否为 0 可以使用这个精度。

FLT_GUARD 不知道是啥意思--!

FLT_MANT_DIG 定义了 float 型 F 位的长度。

FLT_MAX 定义了 float 型可表示的最大数值。

FLT_MAX_10_EXP 定义了 float 型十进制的最大幂。

FLT_MAX_EXP 定义了 float 型二进制的最大幂。

FLT_MIN 定义了 float 型所能表示的最小正数。

FLT_MIN_10_EXP 定义了 float 型十进制的最小幂。

FLT_MIN_EXP 定义了 float 型二进制的最小幂。

float.h 文件里对其它的浮点数也做了规定，本文不再做介绍了。