浮点数运算

南京大学
计算机科学与技术系
袁春风
email: cfyuan@nju.edu.cn
2015.6

浮点数运算及结果

设两个规格化浮点数分别为 $A=M_a \cdot 2^{Ea}$　$B=M_b \cdot 2^{Eb}$,则:

$A \pm B = (M_a \pm M_b \cdot 2^{-(Ea-Eb)}) \cdot 2^{Ea}$　(假设Ea>=Eb)

$A*B = (M_a * M_b) \cdot 2^{Ea+Eb}$

$A/B = (M_a / M_b) \cdot 2^{Ea-Eb}$

1.5+1.5=?

1.5-1.0=?

上述运算结果可能出现以下几种情况:

SP最大指数为多少? 127

阶码上溢: 一个正指数超过了最大允许值 =〉+∞/-∞/溢出

阶码下溢: 一个负指数超过了最小允许值 =〉+0/-0

SP最小指数呢? -126

尾数溢出: 最高有效位有进位 =〉右规

尾数溢出，结果不一定溢出

非规格化尾数: 数值部分高位为0 =〉左规

右规或对阶时，右段有效位丢失 =〉尾数舍入

运算过程中添加保护位

IEEE建议实现时为每种异常情况提供一个自陷允许位。若某异常对应的位为1，则发生相应异常时，就调用一个特定的异常处理程序执行。

IEEE 754 标准规定的五种异常情况

① 无效运算（无意义）

- 运算时有一个数是非有限数，如：

  加 / 减∞、0 x ∞、 ∞/∞等

- 结果无效，如：

  源操作数是NaN、0/0、x REM 0、 ∞ REM y 等

② 除以0（即：无穷大）

③ 数太大（阶上溢）：对于SP，指阶码 E >1111 1110 （指数大于127）

④ 数太小（阶下溢）：对于SP，指阶码 E < 0000 0001（指数小于-126 ）

⑤ 结果不精确（舍入时引起），例如1/3、1/10等不能精确表示成浮点数

上述情况硬件可以捕捉到，因此这些异常可设定让硬件处理，也可设定让软件处理。让硬件处理时，称为硬件陷阱。

浮点数除0的问题

```
#include <conio.h>
#include <stdio.h>
int    main()
{
        int    a=1,b=0;
        printf( "Division    by    zero:%d\n ",a/b);
        getchar();
        return    0;
}
int    main()
{
        double    x=1.0,y=-1.0,z=0.0;
        printf( "division    by    zero:%f    %f\n ",x/z,y/z);
        getchar();
        return    0;
}
```

这是网上的一个帖子

为什么整数除0会发生异常?

为什么浮点数除0不会出现异常?

浮点运算中，一个有限数除以0，
结果为正无穷大（负无穷大）

问题一:为什么整除int型会产生错误? 是什么错误?

二:用double型的时候结果为1.#INF00和-1.#INF00，作何解释???

浮点数加/减运算

- 十进制科学计数法的加法例子

$1.123 \times 10^5 + 2.560 \times 10^2$

其计算过程为：

$1.123 \times 10^5 + 2.560 \times 10^2 = 1.123 \times 10^5 + 0.002560 \times 10^5$

$=(1.123 + 0.00256) \times 10^5 = 1.12556 \times 10^5$

$=1.126 \times 10^5$

进行尾数加减运算前，必须“对阶”！最后还要考虑舍入

计算机内部的二进制运算也一样！

- “对阶”操作：目的是使两数阶码相等
  - 小阶向大阶看齐，阶小的那个数的尾数右移，右移位数等于两个阶码差的绝对值
  - IEEE 754尾数右移时，要将隐含的“1”移到小数部分，高位补0，移出的低位保留到特定的“附加位”上

浮点数加减法基本要点

（假定：Xm、Ym分别是X和Y的尾数， Xe和Ye 分别是X和Y的阶码 ）

(1) 求阶差：Δe=Xe – Ye （若Ye > Xe，则结果的阶码为Ye)

(2) 对阶：将Xm右移Δe位，尾数变为 $Xm*2^{Xe-Ye}$ （保留右移部分附加位）

(3) 尾数加减： $Xm*2^{Xe-Ye} \pm Ym$

(4) 规格化：

当尾数高位为0，则需左规：尾数左移一次，阶码减1，直到MSB为1

每次阶码减1后要判断阶码是否下溢（比最小可表示的阶码还要小）

当尾数最高位有进位，需右规：尾数右移一次，阶码加1，直到MSB为1

每次阶码加1后要判断阶码是否上溢（比最大可表示的阶码还要大）

阶码溢出异常处理：阶码上溢，则结果溢出；阶码下溢，则结果为0

(5) 如果尾数比规定位数长（有附加位），则需考虑舍入（有多种舍入方式）

(6) 若运算结果尾数是0，则需要将阶码也置0。为什么?

尾数为0说明结果应该为0（阶码和尾数为全0）。

浮点数加法运算举例

例：用二进制浮点数形式计算 0.5 +(– 0.4375) =?

0.4375=0.25+0.125+0.0625=0.0111B

解：$0.5 = 1.000 \times 2^{-1}$，$-0.4375 = -1.110 \times 2^{-2}$

对　阶：$-1.110 \times 2^{-2} \rightarrow -0.111 \times 2^{-1}$

加　减：$1.000 \times 2^{-1} + (-0.111 \times 2^{-1}) = 0.001 \times 2^{-1}$

左　规：$0.001 \times 2^{-1} \rightarrow 1.000 \times 2^{-4}$

判溢出：无

结果为：$1.000 \times 2^{-4} = 0.0001000 = 1/16 = 0.0625$

问题：为何IEEE 754 加减运算右规时最多只需一次?

因为即使是两个最大的尾数相加，得到的和的尾数也不会达到4，故尾数的整数部分最多有两位，保留一个隐含的“1”后，最多只有一位被右移到小数部分。

Extra Bits(附加位)

"Floating Point numbers are like piles of sand; every time you move one you lose a little sand, but you pick up a little dirt. "

"浮点数就像一堆沙，每动一次就会失去一点‘沙’，并捡回一点‘脏’"

如何才能使失去的“沙”和捡回的“脏”都尽量少呢? 在后面加附加位!

加多少附加位才合适?

无法给出准确的答案!

Add/Sub:

	1.xxxxx	1.xxxxx	1.xxxxx	1.xxxxxxxx
+	1.xxxxx	0.001xxxxx	0.01xxxxx	-1.xxxxxxxx
	1x.xxxxy	1.xxxxxyyy	1x.xxxxyyy	0.0…0xxxx

IEEE754规定: 中间结果须在右边加2个附加位 (guard & round)

Guard (保护位): 在significand右边的位

Round (舍入位): 在保护位右边的位

附加位的作用: 用以保护对阶时右移的位或运算的中间结果。

附加位的处理: ①左规时被移到significand中; ② 作为舍入的依据。

Rounding Digits(舍入位)

举例：若十进制数最终有效位数为 3，采用两位附加位（G、R）。

问题：若没有舍入位，采用就近舍入到偶数，则结果是什么?

结果为2.36！精度没有2.37高！

$$2.3400 * 10^2$$

$$0.0253 * 10^2$$

$$2.3653 * 10^2$$

IEEE Standard: four rounding modes（用图说明）

round to nearest (default)
round towards plus infinity (always round up)
round towards minus infinity (always round down)
round towards 0

round to nearest: 称为就近舍入到偶数

round digit < 1/2 then truncate (截断、丢弃)

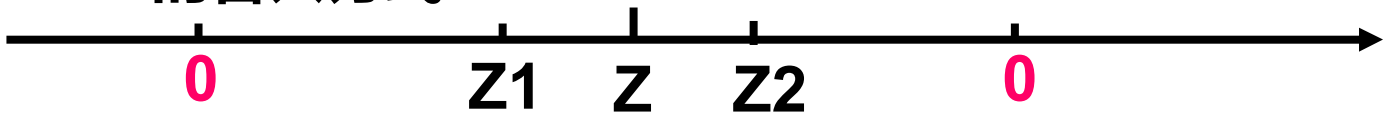
> 1/2 then round up (末位加1)

= 1/2 then round to nearest even digit（最近偶数）

可以证明默认方式得到的平均误差最小。

IEEE 754的舍入方式的说明

IEEE 754的舍入方式



（Z1和Z2分别是结果Z的最近的可表示的左、右两个数）

(1) 就近舍入：舍入为最近可表示的数

非中间值：0舍1入；

中间值：强迫结果为偶数-慢

例如：附加位为
01：舍
11：入
10：(强迫结果为偶数)

例：1.1101**11** → 1.1110;　1.1101**01** → 1.1101;
1.1101**10** → 1.1110;　1.1111**10** → 10.0000;

(2) 朝+∞方向舍入:舍入为Z2(正向舍入)

(3) 朝-∞方向舍入:舍入为Z1(负向舍入)

(4) 朝0方向舍入：截去。正数：取Z1；负数：取Z2

浮点数舍入举例

例：将同一实数分别赋值给单精度和双精度类型变量，然后打印输出。

```
#include <stdio.h>
main()
{
        float a;
        double b;
        a = 123456.789e4;
        b = 123456.789e4;
        printf( “%f/n%f/n” ,a,b);
}
```

运行结果如下：

```
        1234567936.000000
        1234567890.000000
```

为什么float情况下输出的结果会比原来的大？这到底有没有根本性原因还是随机发生的？为什么会出现这样的情况？

float可精确表示7个十进制有效数位，后面的数位是舍入后的结果，舍入后的值可能会更大，也可能更小

问题：为什么同一个实数赋值给float型变量和double型变量，输出结果会有所不同呢？

C语言中的浮点数类型

- C语言中有float和double类型，分别对应IEEE 754单精度浮点数格式和双精度浮点数格式
- long double类型的长度和格式随编译器和处理器类型的不同而有所不同，IA-32中是80位扩展精度格式
- 从int转换为float时，不会发生溢出，但可能有数据被舍入
- 从int或 float转换为double时，因为double的有效位数更多，故能保留精确值
- 从double转换为float和int时，可能发生溢出，此外，由于有效位数变少，故可能被舍入
- 从float 或double转换为int时，因为int没有小数部分，所以数据可能会向0方向被截断

浮点数比较运算举例

- 对于以下给定的关系表达式，判断是否永真。

```
int x ;
float f ;
double d ;
```

Assume neither d nor f is NaN

x == (int)(float) x　　否

x == (int)(double) x　　是

f == (float)(double) f　　是

d == (float) d　　否

f == -(-f);　　是

2/3 == 2/3.0　　否

d < 0.0　⇒　((d*2) < 0.0)　　是

d > f　⇒　-f > -d　　是

d * d >= 0.0　　是

x*x>=0　　否

(d+f)-d == f　　否

自己写程序测试一下!

IEEE 754 的范围和精度

- 单精度浮点数（float型）的表示范围多大?

  最大的数据: $+1.11\ldots1X\ 2^{127}$　　约 $+3.4 \times 10^{38}$

  双精度浮点数（double型）呢?　　约 $+1.8 \times 10^{308}$

- 以下关系表达式是否永真?

```
if ( i == (int) ((float) i) ) {
      printf ( "true" );
}
```

Not always true!

How about double?　True!

```
if ( f == (float) ((int) f) ) {
      printf ( "true" );
}
```

Not always true!

How about double?　同 float

- 浮点数加法结合律是否正确?　FALSE!

  $x = -1.5 \times 10^{38}$,　$y = 1.5 \times 10^{38}$,　$z = 1.0$

  $(x+y)+z = (-1.5\times10^{38}+1.5\times10^{38}) +1.0 = 1.0$

  $x+(y+z) = -1.5\times10^{38}+ (1.5\times10^{38}+1.0) = 0.0$

举例：Ariana火箭爆炸

- 1996年6月4日，Ariana 5火箭初次航行，在发射仅仅37秒钟后，偏离了飞行路线，然后解体爆炸，火箭上载有价值5亿美元的通信卫星。
- 原因是在将一个64位浮点数转换为16位带符号整数时，产生了溢出异常。溢出的值是火箭的水平速率，这比原来的Ariana 4火箭所能达到的速率高出了5倍。在设计Ariana 4火箭软件时，设计者确认水平速率决不会超出一个16位的整数，但在设计Ariana 5时，他们没有重新检查这部分，而是直接使用了原来的设计。
- 在不同数据类型之间转换时，往往隐藏着一些不容易被察觉的错误，这种错误有时会带来重大损失，因此，编程时要非常小心。

举例：爱国者导弹定位错误

- 1991年2月25日，海湾战争中，美国在沙特阿拉伯达摩地区设置的爱国者导弹拦截伊拉克的飞毛腿导弹失败，致使飞毛腿导弹击中了一个美军军营，杀死了美军28名士兵。其原因是由于爱国者导弹系统时钟内的一个软件错误造成的，引起这个软件错误的原因是浮点数的精度问题。
- 爱国者导弹系统中有一内置时钟，用计数器实现，每隔0.1秒计数一次。程序用0.1的一个24位定点二进制小数x来乘以计数值作为以秒为单位的时间
- 这个x的机器数是多少呢?
- 0.1的二进制表示是一个无限循环序列：0.00011[0011]...，x=0.000 1100 1100 1100 1100 1100B。显然，x是0.1的近似表示，0.1-x

  = 0.000 1100 1100 1100 1100 1100 [1100]... -

  0.000 1100 1100 1100 1100 1100B，即为：

  =0.000 0000 0000 0000 0000 0000 1100 [1100]...B

  $=2^{-20}\times 0.1 \approx 9.54\times 10^{-8}$ 这就是机器值与真值之间的误差!

举例：爱国者导弹定位错误

已知在爱国者导弹准备拦截飞毛腿导弹之前，已经连续工作了100小时，飞毛腿的速度大约为2000米/秒，则由于时钟计算误差而导致的距离误差是多少?

100小时相当于计数了$100\times60\times60\times10=36\times10^5$次，因而导弹的时钟已经偏差了$9.54\times10^{-8}\times36\times10^5\approx0.343$秒

因此，距离误差是2000×0.343秒 $\approx$ 687米

小故事：实际上，以色列方面已经发现了这个问题并于1991年2月11日知会了美国陆军及爱国者计划办公室（软件制造商）。以色列方面建议重新启动爱国者系统的电脑作为暂时解决方案，可是美国陆军方面却不知道每次需要间隔多少时间重新启动系统一次。1991年2月16日，制造商向美国陆军提供了更新软件，但这个软件最终却在飞毛腿导弹击中军营后的一天才运抵部队。

举例：爱国者导弹定位错误

- 若x用float型表示，则x的机器数是什么？0.1与x的偏差是多少？系统运行100小时后的时钟偏差是多少？在飞毛腿速度为2000米/秒的情况下，预测的距离偏差为多少？
  - 0.1= 0.0 0011[0011]B=+1.1 0011 0011 0011 0011 0011 00B×2^{-4}，故x的机器数为0 011 1101 1 100 1100 1100 1100 1100 1100
  - Float型仅24位有效位数，后面的有效位全被截断，故x与0.1之间的误差为：|x–0.1|=0.000 0000 0000 0000 0000 0000 0000 1100 [1100]...B。这个值等于$2^{-24}\times0.1 \approx 5.96\times10^{-9}$。100小时后时钟偏差$5.96\times10^{-9}\times36\times10^{5}\approx$0.0215秒。距离偏差0.0215×2000≈43米。比爱国者导弹系统精确约16倍。
- 若用32位二进制定点小数x=0.000 1100 1100 1100 1100 1100 1100 1101 B表示0.1，则误差比用float表示误差更大还是更小？
  - 当x=0.000 1100 1100 1100 1100 1100 1100 1101 B时，与0.1之间的误差约为：|x–0.1|=0.000 0000 0000 0000 0000 0000 0000 0000 00 1100 [1100]...B。这个值等于$2^{-30}\times0.1 \approx 9.31\times10^{-11}$。100小时后时钟偏差$9.31\times10^{-11}\times36\times10^{5}\approx 0.000335$秒。预测的距离偏差仅为$0.000335\times2000 \approx 0.67$米。

举例：浮点数运算的精度问题

- 从上述结果可以看出：
  - 用32位定点小数表示0.1，比采用float精度高64倍
  - 用float表示在计算速度上更慢，必须先把计数值转换为IEEE 754格式浮点数，然后再对两个IEEE 754格式的数相乘，故采用float比直接将两个二进制数相乘要慢
- Ariana 5火箭和爱国者导弹的例子带来的启示

✓ 程序员应对底层机器级数据的表示和运算有深刻理解

✓ 计算机世界里，经常是“差之毫厘，失之千里”，需要细心再细心，精确再精确

✓ 不能遇到小数就用浮点数表示，有些情况下（如需要将一个整数变量乘以一个确定的小数常量），可先用一个确定的定点整数常量与整数变量相乘，然后再通过移位运算来确定小数点