关于补码原码反码的一些简单理解

计算机有三种编码方式表示一个数. 对于正数因为三种编码方式的结果都相同:

`[+1] = [00000001]原 = [00000001]反 = [00000001]补`

但是对于负数:

`[-1] = [10000001]原 = [11111110]反 = [11111111]补`

可见原码, 反码和补码是完全不同的. 既然原码才是被人脑直接识别并用于计算表示方式, 为何还会有反码和补码呢?

首先, 因为人脑可以知道第一位是符号位, 在计算的时候我们会根据符号位, 选择对真值区域的加减. 但是对于计算机, 加减乘数已经是最基础的运算, 要设计的尽量简单. 计算机辨别"符号位"显然会让计算机的基础电路设计变得十分复杂! 于是人们想出了将符号位也参与运算的方法. 我们知道, 根据运算法则减去一个正数等于加上一个负数, 即: 1-1 = 1 + (-1) = 0 , 所以机器可以只有加法而没有减法, 这样计算机运算的设计就更简单了.

于是人们开始探索 将符号位参与运算, 并且只保留加法的方法. 首先来看原码:

计算十进制的表达式: 1-1=0

`1 - 1 = 1 + (-1) = [00000001]原 + [10000001]原 = [10000010]原 = -2`

如果用原码表示, 让符号位也参与计算, 显然对于减法来说, 结果是不正确的.这也就是为何计算机内部不使用原码表示一个数.

为了解决原码做减法的问题, 出现了反码:

计算十进制的表达式: 1-1=0

`1 - 1 = 1 + (-1) = [0000 0001]原 + [1000 0001]原= [0000 0001]反 + [1111 1110]反 = [1111 1111]反 = [1000 0000]原 = -0`

发现用反码计算减法, 结果的真值部分是正确的. 而唯一的问题其实就出现在"0"这个特殊的数值上. 虽然人们理解上+0和-0是一样的, 但是0带符号是没有任何意义的. 而且会有`[0000 0000]原`和`[1000 0000]原`两个编码表示0.

于是补码的出现, 解决了0的二义性以及两个编码的问题:

`1-1 = 1 + (-1) = [0000 0001]原 + [1000 0001]原 = [0000 0001]补 + [1111 1111]补 = [0000 0000]补=[0000 0000]原`

这样0用`[0000 0000]`表示, 而以前出现问题的-0则不存在了.而且可以用[1000 0000]表示-128:

`(-1) + (-127) = [1000 0001]原 + [1111 1111]原 = [1111 1111]补 + [1000 0001]补 = [1000 0000]补`

-1-127的结果应该是-128, 在用补码运算的结果中,`[1000 0000]补` 就是-128. 但是注意因为实际上是使用以前的-0的补码来表示-128, 所以-128并没有原码和反码表示.(对-128的补码表示`[1000 0000]补`算出来的原码是`[0000 0000]原`, 这是不正确的)

使用补码, 不仅仅修复了0的符号以及存在两个编码的问题, 而且还能够多表示一个最低数. 这就是为什么8位二进制, 使用原码或反码表示的范围为[-127, +127], 而使用补码表示的范围为[-128, 127].

ieee754

浮点数表达法采用了科学计数法来表达实数，即用一个有效数字。一个基数（Base）、一个指数（Exponent）以及一个表示正负的符号来表达实数。比如，666.66 用十进制科学计数法可以表达为 6.6666×10^2（其中，6.6666 为有效数字，10 为基数，2 为指数）。浮点数利用指数达到了浮动小数点的效果，从而可以灵活地表达更大范围的实数。

当然，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还不是唯一的。例如，上面例子中的 666.66 可以表达为 0.66666×10^3、6.6666×10^2或者66.666×10^1 三种方式。因为这种表达的多样性，因此有必要对其加以规范化以达到统一表达的目标。规范的浮点数表达方式具有如下形式：

$$\pm\, d.dd\cdots d \times \beta^{e} \quad (0 \leqslant d_i < \beta)$$

其中，`d.dd...d` 为有效数字，`β` 为基数，`e`为指数。

有效数字中数字的个数称为精度，我们可以用 p 来表示，即可称为 p 位有效数字精度。每个数字 `d`介于 `0` 和基数`β` 之间，包括 `0`。更精确地说，可以如下表示：

$$\pm\,(d_0 + d_1 \beta^{-1} + \cdots + d_{p-1} \beta^{-(p-1)})\beta^{e} \quad (0 \leqslant d_i < \beta)$$

其中，对十进制的浮点数，即基数 β 等于 10 的浮点数而言，上面的表达式非常容易理解。如 12.34，我们可以根据上面的表达式表达为：$1 \times 10^1 + 2 \times 10^0 + 3 \times 10^{-1} + 4 \times 10^{-2}$，其规范浮点数表达为$1.234 \times 10^1$。

但对二进制来说，上面的表达式同样可以简单地表达。唯一不同之处在于：二进制的 β 等于 2，而每个数字 d 只能在 0 和 1 之间取值。如二进制数 1001.101，我们可以根据上面的表达式表达为：$1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$，其规范浮点数表达为 1.001101×2^3。

现在，我们就可以这样简单地把二进制转换为十进制，如二进制数 1001.101 转换成十进制为：

$$\begin{aligned} & 1001.101 \\ &= 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} \\ &= 8 + 0 + 0 + 1 + \frac{1}{2} + 0 + \frac{1}{8} \\ &= 9\frac{5}{8} \\ &= 9.625 \end{aligned}$$

由上面的等式，我们可以得出：向左移动二进制小数点一位相当于这个数除以 2，而向右移动二进制小数点一位相当于这个数乘以 2。如

101.11=3/4

而

10.111=7/8

除此之外，我们还可以得到这样一个基本规律：一个十进制小数要能用浮点数精确地表示，最后一位必须是 5（当然这是必要条件，并非充分条件）。规律推演如下面的示例所示：

$$0.1=2^{-1}=0.5$$
$$0.01=2^{-2}=0.25$$
$$0.001=2^{-3}=0.125$$
$$0.0001=2^{-4}=0.0625$$
$$0.00001=2^{-5}=0.03125$$
$$0.000001=2^{-6}=0.015625$$
$$0.0000001=2^{-7}=0.0078125$$
$$0.00000001=2^{-8}=0.00390625$$

我们也可以使用一段 C ++程序来验证：

```
#include <iostream>
using namespace std;
int main(void)
{
    float f1=34.6;
    float f2=34.5;
    float f3=34.0;
    cout<<"34.6-34.0="<<(f1-f3)<<'\n';
    cout<<"34.5-34.0="<<(f2-f3)<<'\n';
    return 0;
}
```

运行结果为：

```
34.6-34.0=0.599998
34.5-34.0=0.500000
```

之所以`“34.6-34.0=0.599998”`，产生这个误差的原因是`34.6` 无法精确地表达为相应的浮点数，而只能保存为经过舍入的近似值。而这个近似值与 `34.0` 之间的运算自然无法产生精确的结果。

上面阐述了二进制数转换十进制数，如果你要将十进制数转换成二进制数，则需要把整数部分和小数部分分别转换。其中，整数部分除以`2`，取余数；小数部分乘以 `2`，取整数位。如将 `13.125` 转换成二进制数如下：

1、首先转换整数部分（13），除以 2，取余数，所得结果为 1101。

2、其次转换小数部分（0.125），乘以 2，取整数位。转换过程如下：
`0.125×2=0.25` 取整数位0
`0.25×2=0.5`取整数位0
`0.5×2=1` 取整数位1

- 注意：可结合高程第一周作业理解

3、小数部分所得结果为 001，即 13.125=1101.001，用规范浮点数表达为 $1.101001×2^3$。

浮点数表示法

IEEE 浮点数标准是从逻辑上用三元组{S，E，M}来表示一个数 V 的，即 V=（-1）S×M×2^E

其中：

符号位`s（Sign）`决定数是正数`（s=0）`还是负数`（s=1）`，而对于数值`0`的符号位解释则作为特殊情况处理。

尾数位`M（Significand）`是二进制小数，它的取值范围为 1 ~ $2^{-ε}$，或者为 0 ~ $1^{-ε}$。它也被称为尾数位（Mantissa）、系数位（Coefficient），甚至还被称作“小数”。

指数位`E（Exponent）`是 2 的幂（可能是负数），它的作用是对浮点数加权

1. 格式化值

当指数段 exp 的位模式既不全为 0（即数值 0），也不全为 1（即单精度数值为 255，以单精度数为例， 8 位的指数为可以表达 0~255 的 255 个指数值)的时候，就属于这类情况。

我们知道，指数可以为正数，也可以为负数。为了处理负指数的情况，实际的指数值按要求需要加上一个偏置（Bias）值作为保存在指数段中的值。因此，这种情况下的指数段被解释为以偏置形式表示的有符号整数。即指数的值为：指数段对应的无符号整数减去`偏置值`（32位浮点数中为127）

对小数段 frac，可解释为描述小数值 f，其中 0≤f<1，其二进制表示为 $0.f_{n-1}…f_1f_0$，也就是二进制小数点在最高有效位的左边。有效数字定义为 M=1+f。有时候，这种方式也叫作隐含的以 1 开头的表示法，因为我们可以把 M 看成一个二进制表达式为 $1.f_{n-1}f_{n-2}…f_0$ 的数字。既然我们总是能够调整指数 E，使得有效数字 M 的范围为 1≤M<2（假设没有溢出），那么这种表示方法是一种轻松获得一个额外精度位的技巧。同时，由于第一位总是等于 1，因此我们就不需要显式地表示它。拿单精度数为例，按照上面所介绍的知识，实际上可以用 23 位长的有效数字来表达 24 位的有效数字。比如，对单精度数而言，二进制的 1001.101（即十进制的 9.625）可以表达为 $1.001101×2^3$，所以实际保存在有效数字位中的值为：

```
00110100000000000000000
```

即去掉小数点左侧的 1，并用 0 在右侧补齐。

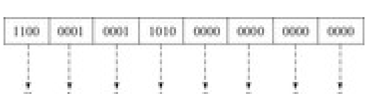
根据上面所阐述的规则，下面以实数 -9.625 为例，来看看如何将其表达为单精度的浮点数格式。具体转换步骤如下：

1、首先，需要将 -9.625 用二进制浮点数表达出来，然后变换为相应的浮点数格式。即 -9.625 的二进制为 1001.101，用规范的浮点数表达应为 $1.001101×2^3$。

2、其次，因为 -9.625 是负数，所以符号段为 1。而这里的指数为 3，所以指数段为 3+127=130，即二进制的 10000010。有效数字省略掉小数点左侧的 1 之后为 001101，然后在右侧用零补齐。因此所得的最终结果为：

1	10000010	00110100000000000000000

3、最后，我们还可以将浮点数形式表示为十六进制的数据，如下所示：



即最终的十六进制结果为 0xC11A0000。

2. 特殊数值

IEEE 标准指定了以下特殊值：±0、反向规格化的数、±∞ 和 NaN（如下表所示）。这些特殊值都是使用 e_{max+1} 或 e_{min-1} 的指数进行编码的。

[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]
[illegible]	[illegible]	[illegible]

NaN：当指数段 exp 全为 1 时，小数段为非零时，结果值就被称为“NaN”（Not a Number）。

一般情况下，我们将 0/0 或:

$$\sqrt{-1}$$

视为导致计算终止的不可恢复错误。但是，一些示例表明在这样的情况下继续进行计算是有意义的。这时候就可以通过引入特殊值 NaN，并指定诸如 0/0 或

$$\sqrt{-1}$$

之类的表达式计算来生成 NaN 而不是停止计算，从而避免此问题。下表中列出了一些可以导致 NaN 的情况。

操作	产生 NaN 的表达式
+	$\infty + (-\infty)$
×	$0 \times \infty$
/	$0/0$, ∞ / ∞
REM	x REM 0, ∞ REM y
$\sqrt{}$	$\sqrt{x}$（当 $x < 0$ 时）

无穷：当指数段 exp 全为 1，小数段全为 0 时，得到的值表示无穷。当 s=0 时是 +∞，或者当 s=1 时是 -∞。

无穷用于表达计算中产生的上溢问题。比如两个极大的数相乘时，尽管两个操作数本身可以保存为浮点数，但其结果可能大到无法保存为浮点数，必须进行舍入操作。根据IEEE标准，此时不能将结果舍入为可以保存的最大浮点数（因为这个数可能与实际的结果相差太远而毫无意义），而应将其舍入为无穷。对于结果为负数的情况也是如此，只不过此时会舍入为负无穷，也就是说符号域为1的无穷。

3. 非格式化值(了解即可)

当指数段 exp 全为 0 时，所表示的数就是非规格化形式。

在这种情况下，指数值 E=1-Bias，而有效数字的值 M=f，也就是说它是小数段的值，不包含隐含的开头的 1。

非规格化值有两个用途：

第一，它提供了一种表示数值 0 的方法。因为规格化数必须得使有效数字 M 在范围 1≤M<2 之中，即 M≥1，因此它就不能表示 0。实际上，+0.0 的浮点表示的位模式为全 0（即符号位是 0，指数段全为 0，而小数段也全为 0），这就得到 M=f=0。令人奇怪的是，当符号位为 1，而其他段全为 0 时，就会得到值 -0.0。根据 IEEE 的浮点格式来看，值 +0.0 和 -0.0 在某些方面是不同的。

第二，它表示那些非常接近于 0.0 的数。它们提供了一种属性，称为逐渐下溢出。其中，可能的数值分布均匀地接近于 0.0。

一些精度问题的简单讨论：

我们先来思考这样一件事: 现在的计算机能存储[1,2]之间的所有小数吗?

稍想一下就知道: 不可以. 因为计算机的内存或硬盘容量是有限的(可以把内存看成一个盒子), 而1到2之间小数的个数是无限的.

极端一点, 计算机甚至无法存储1到2之间的某一个小数, 比如对于小数 **1.00000.....一万亿个零.....00001,** 恐怕很难用计算机去存储它...

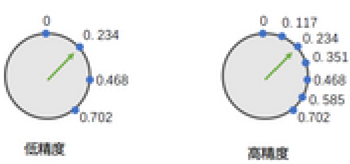
不过计算机却能存储[1, 10000]之间的所有整数. 因为整数是`"离散"`的, [1, 10000]之间的整数只有10000个. 这么多种状态是很容易就能存储到计算机中, 而且还能进行运算, 比如计算10000 + 10000, 也只是要求你的计算机能存储20000种状态而已...

这样来看的话: 计算机可以进行数学概念中的整数运算的, 但却难以进行数学概念中的小数运算. 小数这种"连续"的东西, 当前的计算机很难应对。

所以, 计算机为了进行小数运算, 不得不将小数也当成`"离散"`的值, 一个一个的, 就像整数那样:

数学中的整数是一个一个的, 想象绿色指针必须一次走一格

- 数学中的小数是连续的, 想象一下上图的绿色指针可以随意调节, 想走到哪儿走到哪儿
- 而计算机中存储的小数是一个一个的, 绿色指针必须一次走指定一格, 就像整数那样
- 这就引发了精度的问题, 比如上图中, 我们无法在计算机中存储0.3, 因为绿色指针只能一次走指定一格, 要么在0.234, 要么就到了0.468...
- 当然, 我们也可以增加计算机存储小数的精度, 或者说缩小点与点之间的间隔（更加密集）:



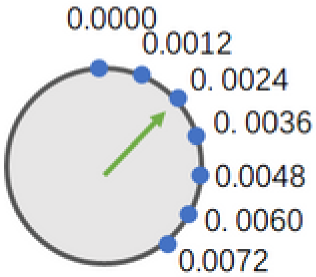
1.从"间隔"的角度理解来"精度", 其实是这样一种思路:

想象一个类似于上图的圆形表盘, 表盘上有一些蓝点作为刻度, 有一个绿色的指针用于指向蓝点, 绿色指针只能一次走一格: 即只能从当前蓝点移动到下一个蓝点. 不能指向两个蓝点之间的位置.

假如表盘上用于表示刻度的蓝点如下所示:

0.0000、0.0012、0.0024、0.0036、0.0048、0.0060、0.0072、0.0084、0.0096、0.0108、0.0120 、0.0132、0.0144、...

即, 这是一组十进制数, 这组数以 0.0012 的步长逐渐递增... 假设这个表盘就就是你的计算机所能表示的所有小数.



此时可以考虑一下我们能说这个表盘, 或者说这组数的精度达到了 4 位十进制数吗(比如, 可以精确到1位整数 + 3位小数)?

如果说可以精确点1位整数 + 3位小数, 那我们就应该可以说出下面这样的话:

我们可以说, 当前指针正位于0.001x: 而指针确实可以位于0.0012, 属于0.001x (x表示这一位是任意数, 或说这对该位的精度不做限制)

我们可以说, 当前指针位于0.002x: 而指针确实可以位于0.0024, 属于0.002x

我们可以说, 当前指针位于0.003x: 而指针确实可以位于0.0036, 属于0.003x

...

我们可以说, 当前指针位于0.009x: 而指针确实可以位于0.0096, 属于0.009x

我们可以说, 当前指针位于0.010x: 而指针确实可以位于0.0108, 属于0.010x

我们可以说: 当前指针位于0.011x...但, 注意, 指针始终无法指向0.011x...在我们的表盘中, 指针可以指向0.0108, 或指向0.0120, 但始终无法指向0.011x

...

这就意味着: 对于当前表盘 (或者说对于这组数) 来说, 4位精度太高了...4位精度所能描述的状态中, 有一些是无法用这个表盘表示的.

那, 把精度降低一些.

我们能说这个表盘, 或者说这组数的精度达到了 3 位十进制数吗(比如, 可以精确到1位整数 + 2位小数)?

再来分析一下: 如果说可以精确点1位整数 + 2位小数, 那我们就应该可以说出下面这样的话:

我们可以说, 当前指针位于0.00xx: 而指针确实可以位于0.0012, 0.0024, 0.0036...0.0098, 这些都属于0.00xx

我们可以说, 当前指针位于0.01xx: 而指针确实可以位于0.0108, 0.0120...这些都属于0.01xx

...

可以看出, 对于当前这个表盘 (或者说对于这组数) 来说, 它完全能"hold住"3位精度. 或者说3位精度所能描述的所有状态在该表盘中都可以得到表示.

如果我们的机器使用这个表盘作为浮点数的取值表盘的话, 那我们就可以说:

我们机器的浮点数精度 (或者说这个表盘的浮点数精度), 能精确到3位十进制数(无法精确到4位十进制数).

而这个精度, 本质上是由表盘间隔决定的, 本例中的表盘间隔是0.0012, 如果把表盘间隔缩小到0.00000012, 那相应的表盘能表示的精度就会提升(能提升到 7 位十进制数, 无法达到 8 位十进制数)

通过这个例子, 希望大家能够直观的认识到 "表盘的间隔" 和 "表盘的精度" 之间, 存在着密切的关系. 这将是后文进行讨论的基础.

事实上: ieee754标准中的32位浮点数, 也可以被想象为一个 "蓝点十分密集的浮点数表盘", 如果我们能分析出这个表盘中蓝点之间的间隔, 那我们就能分析出这个表盘的精度.

2.32位浮点数的间隔

那怎么分析32位浮点数的间隔与精度呢, 有一个很笨的方法: 把32位浮点数能表示的所有小数都罗列出来, 计算间隔. 然后分析精度...

注: 此处只分析规格数(normal number), 且先不考虑负数情况, 也就是说不考虑符号位为 1 的情况

32位浮点数能表示的最小规格数是 :

0 00000001 00000000000000000000000 (二进制)

(注意, 规格数的指数位最小为 00000001 , 不能为00000000. 这个在本系列的第二章中已经讨论过了, 以下不再赘述)

紧邻的下一个数是:

0 00000001 00000000000000000000001 (二进制)

紧邻的下一个数是:

0 00000001 00000000000000000000010 (二进制)

紧邻的下一个数是:

0 00000001 00000000000000000000011(二进制)

...

这样一步一步的往下走, [公式] 步之后, 我们将指向这个数:

0 00000001 11111111111111111111111(二进制)

再走一步, 也就是 2^{23}步之后, 我们将指向这个数:

0 00000010 00000000000000000000000(二进制)

总结一下: 2^{23}次移动之后:

我们从起点: 0 00000001 00000000000000000000000,

移动到了终点: 0 00000010 00000000000000000000000

现在可以求间隔了, 间隔 = 差值 / 移动次数 = (终点对应的值 - 起点对应的值) / 2^{23},

但是, 先别急着计算. 我们先仔细观察一下, 可以发现, 和起点相比, 终点的符号位和尾数位都没变, 仅仅是指数位变了: 起点指数位00000001 → 终点指数位00000010, 终点的指数位, 比起点的指数位变大了1

而ieee754中浮点数的求值公式是:

$尾数 * 2^{指数}$ $尾数 * 2^{指数}$

(先不考虑符号位)
这样的话: 假如说起点对应的值是

$$1.0 * 2^{-8}\ 1.0 * 2^{-8}$$

那终点对应的值就应该是

$$1.0 * 2^{-7}\ 1.0 * 2^{-7}$$

即, 仅仅是指数位变大了1

把指数展开会看的更清晰一些:

假如说起点对应的值是 0.0000 0001 (8位小数)

那终点对应的值就应该是 0.0000 001 (7位小数)

那起点和终点的差值就是: (0.0000 001 - 0.0000 0001), 是一个非常小的数

那间隔就是: 差值 / 2^{23}

注意: 其实上面我们并没有计算出真正的间隔, 只是假设了起点和终点的值分别是

$$1.0 * 2^{-8}\ 1.0 * 2^{-8}$$

和

$$1.0 * 2^{-7}\ 1.0 * 2^{-7}$$

然后算出了一个假设的间隔. 但这个假设格外重要, 下文我们会继续沿用这个假设进行分析

废话不多说, 现在我们继续前进.

现在起点变成了: 0 00000010 00000000000000000000000

再走2^23步, 来到了: 0 00000011 00000000000000000000000

同样: 符号位, 尾数位都没有变, 指数位又变大了1

沿用上面的假设, 此时起点对应的值是

$$1.0 * 2^{-7}\ 1.0 * 2^{-7}$$

, 则终点对应的值应该是

$$1.0 * 2^{-6}\ 1.0 * 2^{-6}$$

, 即, 还是指数位变大了1

再次计算差值: 0.0000 01(6位小数) - 0.0000 001(7位小数)

再次计算间隔: 等于 差值 / 2^23(移动次数)

不知道同学们有没有体会到不对劲的地方, 没有的话, 我们计算往前走:

现在起点变成了: 0 00000011 00000000000000000000000

再走2^23步, 来到了: 0 00000100 00000000000000000000000

同理, 终点相对起点, 还只是指数位变大了1

再次计算差值: (0.00001(5位小数) - 0.000001(6位小数))...

再次计算间隔: 等于 差值 / 2^23(移动次数)

感受到不对劲了吗? 继续往前走...

现在起点变成了: 0 00000100 00000000000000000000000

再走2^23步, 来到了: 0 00000101 00000000000000000000000

再次计算差值: (0.0001(4位小数) - 0.00001(5位小数))...

再次计算间隔: 等于 差值 / 2^23(移动次数)

...一路走到这儿, 感受到不对劲了吗?

不对劲的地方在于: 终点和起点的差值! 差值在越变越大! 同理间隔也在越变越大!

不信的话我们来罗列一下之前的差值:

`那差值就是: 0.0000 001 ( 7位小数) - 0.0000 0001( 8位小数), 差值等于0.0000 0009`

`...`

`那差值就是: (0.000001 ( 6位小数) - 0.0000001( 7位小数)), 差值等于0.0000 009`

`...`

`那差值就是: (0.00001 ( 5位小数) - 0.000001( 6位小数)), 等于 0.0000 09`

`...`

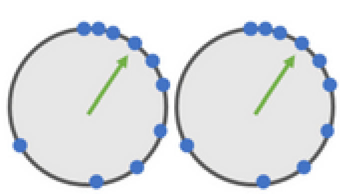
`那差值就是: (0.0001 ( 4位小数) - 0.00001( 5位小数)), 等于 0.0000 9`

差值的小数点在不断向右移动, 这样走下次, 总有一天, 差值会变成9, 变成90, 变成90000...

而 移动次数始终 = 2^23, 间隔始终 = 差值/2^23....差值在越变越大, 间隔也会跟着越变越大...

到这里, 你发现了ieee754标准的一个重要特性: 如果把ieee754所表示的浮点数想象成一个表盘的话, 那表盘上的蓝点不是均匀分布的, 而是越来间隔越大, 越来越稀疏:

大概就像这样:



也因此，浮点数只能存储蓝点位置对应的值（即一些固定值，或者说近似值）
正如前文所说, 32位浮点数会形成一个表盘, 表盘上的蓝点逐渐稀疏. 绿色指针只能指向某个蓝点, 不能指向两个蓝点之间的位置. 或者换句话说: 32位浮点数只能保存蓝点对应的值.
如果你要保存的值不是蓝点对应的值, 就会被自动舍入到离该数最近的蓝点对应的值.

也可以参考阅读以下内容

计算机内部通常使用浮点数进行实数的运算, 计算机的浮点数是仅有有限字长的二进制 数, 大部分实数存入计算机时需要做四舍五入, 由此引起的误差称为舍入误差. 一个浮点数 的表示由正负号、小数形式的尾数以及为确定小数点位置的阶三部分组成. 例如单精度实数 用 32 位的二进制表示, 其中符号占 1 位. 尾数占 23 位, 阶数占 8 位. 这样一个规范化的计 算机单精度数 (零除外) 可以写成如下形式:

$$\pm 2^{p} \times (0.\alpha_1\alpha_2\alpha_3\cdots\alpha_{23})_2, \quad |p| \leqslant 2^7 - 1, \quad p \in Z. a, \in \{0,1\}.$$

上面记号中, Z 表示整数集. 二进制的非零数字只有 1, 所以 $\alpha_1 = 1$. 阶数的 8 位中须有 1 位表示阶数的符号, 所以阶数的值占 7 位. 凡是能帕写成上述形式的数称为机器数. 设机器数 a 有上述形式, 则与之相邻的机器数为 $b = a + 2^{p-23}$ 和 $c = a - 2^{p-23}$. 这样, 区间 (c.a) 和 (a,b) 中的数无法准确表示, 计算机通常按规定用与之最近的机器数表示.
设实数 x 在机器中的浮点 (float) 表示为 $fl(x)$, 我们把 $x - fl(x)$ 称为舍入误差. 如当 $x \in \left[\frac{c+a}{2}, \frac{a+b}{2}\right) = [a - 2^{p-1-23}, a + 2^{p-1-23})$ 时, 用 a 表示 x. 记为 $fl(x) = a$. 其相对误差 满足

$$|\varepsilon_r| = \left|\frac{x - fl(x)}{fl(x)}\right| \leqslant \frac{2^{p-1-23}}{2^{p-1}} = 2^{-23} \approx 10^{-6.9}.$$

上式表明单精度实数有 $6 \sim 7$ 位有效数字.
二进制阶数最高为 $2^7 - 1$, 相应于十进制的阶数 38 . 即 $(2^7 - 1)\lg 2$. 因此单精度实数 (零 除外) 的数量级不大于 10^{38} 且不小于 10^{-38}. 当输入数技、输出数据或中间数据太大而无法表示时, 计算过程将会非正常停止, 此现象称为上溢(overflow): 当数据太小而只能用零表示时, 计算机将此数置零, 精度损失, 此现象称为下溢(underflew). 下溢并不总是有害的. 在做浮点运算时, 我们需要考虑数据运算可能产生的上溢及有害的下溢.