深入理解浮点数

0、几个问题

开始前请思考如下问题：

- 二进制 0.1，用十进制表示是多少？十进制的 0.1，用二进制表示是多少？
- 为什么 0.1 + 0.2 = 0.30000000000000004？
- 单精度和双精度浮点数的有效小数位分别是多少？
- 单精度浮点数能表示的范围是什么？
- 浮点数为什么会存在-0？infinity 和 NaN 是怎么表示？

1、什么是浮点数

数学中无浮点数概念，虽然小数象浮点数，但从不这么叫。为什么计算机中不叫小数而叫浮点数呢？因为资源的限制，数学中的小数无法直接在计算机中准确表示。为更好表示它，计算机科学家们发明了浮点数，这是对小数的近似表示。维基百科中关于浮点数的概念说明：“The term floating point refers to the fact that a number's radix point (decimal point, or, more commonly in computers, binary point) can float; that is, it can be placed anywhere relative to the significant digits of the number.” 也就是说，浮点数是相对于定点数而言的，表示小数点位置是浮动的。如 7.5×10、0.75×10²等表示法，值一样，但小数点位置不一样。具体来说，浮点数是指用符号、尾数、基数和指数等四部分表示的小数。

2、IEEE754 又是什么

知道了浮点数的概念，但需要确定一套具体的表示、运算标准。其中最有名的就是 IEEE754 标准。William Kahan 因浮点数标准化的工作获得了图灵奖。

The **IEEE Standard for Floating-Point Arithmetic** (**IEEE 754**) is a technical standard for floating-point arithmetic established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE). The standard addressed many problems found in the diverse floating-point implementations that made them difficult to use reliably and portably. Many hardware floating-point units use the IEEE 754 standard.

以下讨论基于 IEEE754 标准，这是目前多数处理器用的标准。根据上面浮点数的组成，因为是在计算机中表示浮点数，基数自然是 2，因此 IEEE754 浮点数只关注符号、尾数和指数三部分。

3、小数的二进制和十进制转换

为方便理解后面内容，先看二进制和十进制的转换，其中涉及小数转换。

二进制转十进制

和整数转换一样，用各位数值和位权相乘。如：$(0.101)_2 = 1\times2^{-1} + 0\times2^{-2} + 1\times2^{-3} = (0.625)_{10}$，记住小数点后第一位指数是从-1 开始。

十进制转二进制

十进制整数转二进制采用“除 2 取余，逆序排列”法。如十进制数 11 转为二进制：11/2=5 余 1；5/2=2 余 1；2/2=1 余 0；1/2=0 余 1。所以 $(11)_{10}$ 的二进制是 $(1011)_2$。

但若十进制是小数，转为二进制小数如何做？采用“乘 2 取整，顺序排列”。如十进制小数 0.625 转为二进制小数：0.625*2=1.25 … 取整数部分 1；0.25*2=0.5 … 取整数部分 0；0.5*2=1 … 取整数部分 1；顺序排列，所以 $(0.625)_{10} = (0.101)_2$。

为快速转换，网上有很多工具。推荐 https://baseconvert.com/，支持各进制的转换，还支持浮点数。

4、经典问题：0.1 + 0.2 = 0.30000000000000004

这个问题网上相关的讨论很多，甚至有专门的一个网站：https://0.30000000000000004.com/，这个网站上有各门语言的 0.1+0.2 的结果。如 C 语言：

```
#include <stdio.h>

void main(void)

{

  printf("%.17f\n", .1 + .2);

}
```

结果是 0.30000000000000004。为什么会这样？这要回到 IEEE754 标准关于浮点数的规定。

5、浮点数的 IEEE754 表示

浮点数由四个部分构成，那 IEEE754 标准是如何规定它们的存储方式的呢？一般，IEEE754 浮点数有两种类型：单精度浮点数(float)和双精度浮点数(double)，还有其它不常用的。单精度浮点数用 4 字节表示；双精度浮点数用 8 字节表示。

符号位，0 表示正数；1 表示负数。着重看指数部分和尾数部分。基数固定是 2，因此没必要存。

尾数部分（精度）

“浮点数”叫法的由来在于小数点浮动。具体存储时，需固定一种形式，叫尾数的标准化。IEEE754 规定，**在二进制数中，通过移位，将小数点前面的值固定为 1。IEEE754 称这种形式的浮点数为规范化浮点数(normal number)。**

如十进制数 0.15625，转为二进制是 0.00101。为让第 1 位为 1，执行逻辑右移 3 位，尾数部分成为 1.01，因为右移 3 位，所以指数部分是-3。因为规定第 1 位永远为 1，因此没必要存，这样尾数部分多了 1 位，只需存 0100(要记住，这是的数字是小数点后的数字，因此实际是 0.01，转为十进制是 0.25——没算未存的小数点前面的 1)。

对于规范化浮点数，尾数其实比实际的多 1 位，也就是说，单精度的是 24 位，双精度是 53 位。为区分，IEEE754 称这种尾数为 significand。有规范化浮点数，自然会有**非规范化浮点数(**denormal number)，后面会解释。

尾数决定精度，对于单精度浮点数，因为只有 23 位，而 1<<23 对应十进制是 8388608，因此不能完整表示全部的 7 个十进制位，所以说，单精度浮点数**有效小数位**最多 7 位；双精度的**有效小数位**是 15 位！

指数部分（范围）

因为指数有正、有负，为避免用符号位，同时方便比较、排序，指数部分用 **The Biased exponent(**有偏指数)。IEEE754 规定，$2^{e-1}-1$ 的值是 0，其中 e 表示指数部分的位数，小于此值表示负数，大于此值表示正数。因此，对于单精度浮点数而言， $2^{8-1}-1 = 127$ 是 0；双精度浮点数，$2^{11-1}-1 = 1023$ 是 0。**（定了一个二分中间值，大于该值为正，小于该值为负）**

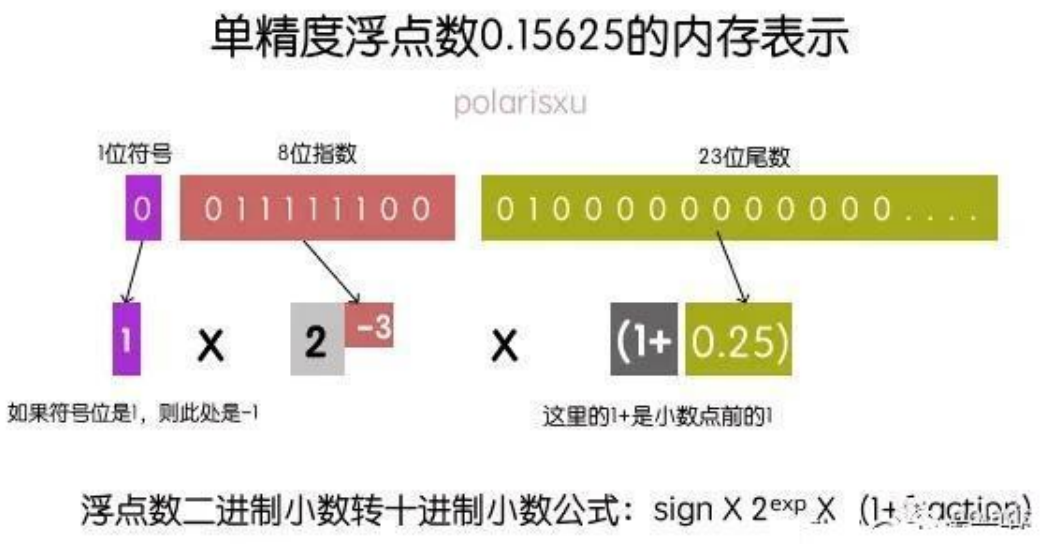
还用十进制 0.15625 举例。因为右移 3 位，所以指数是-3。根据 IEEE754 的定义，单精度浮点数情况下，-3 的实际值是 127-3 = 124。127 表示 0，124 就表示-3。而十进制 124 转为二进制是 1111100。

小结

结合尾数和指数的规定，IEEE754 单精度浮点数，十进制 0.15625 对应的二进制内存表示是：0 01111100 01000000000000000000000。**（尾数用规范化浮点数 + 指数用有偏指数）**

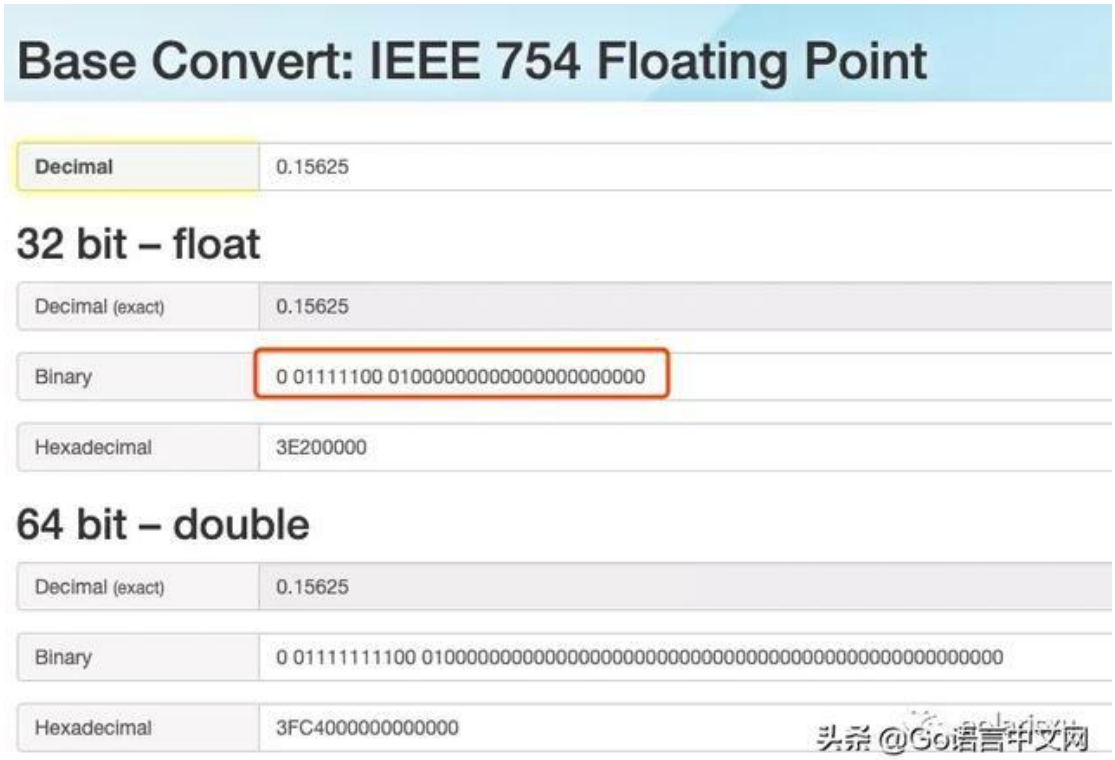
6、程序确认下 IEEE754 的如上规定

这张图是单精度浮点数 0.15625 的内存存储表示。根据三部分的二进制表示，可反推出该数的十进制表示。作为练习，十进制的 2.75，用图表示的话，各个位置分别是什么值？



程序确认单精度浮点数的内存表示

使用 C 语言编程，能得到一个单精度浮点数的二进制内存表示。如提供单精度浮点数 0.15625，该程序能够输出：0-01111100-01000000000000000000000。提供一个在线可运行版本：**https://play.studygolang.com/p/pg0QNQtBHYx**。其实，推荐的那个工具能得到十进制浮点数的二进制内存表示，地址：**https://baseconvert.com/ieee-754-floating-point。**



6、再看 0.1+0.2 = 0.30000000000000004

出错的原因

出现这种情况的根本原因是，有些十进制小数无法转换为二进制数。如下图：

二进制数	对应的十进制数
0.0000	0
0.0001	0.0625
0.0010	0.125
0.0011	0.1875
0.0100	0.25
0.0101	0.3125
0.0110	0.375
0.0111	0.4375
0.1000	0.5
0.1001	0.5625
0.1010	0.625
0.1011	0.6875
0.1100	0.75
0.1101	0.8125
0.1110	0.875
0.1111	0.9

在小数点后 4 位时，连续的二进制数，对应的十进制数不连续，因此只能增加位数来尽可能近似的表示。

0.1 和 0.2 是如何表示的？

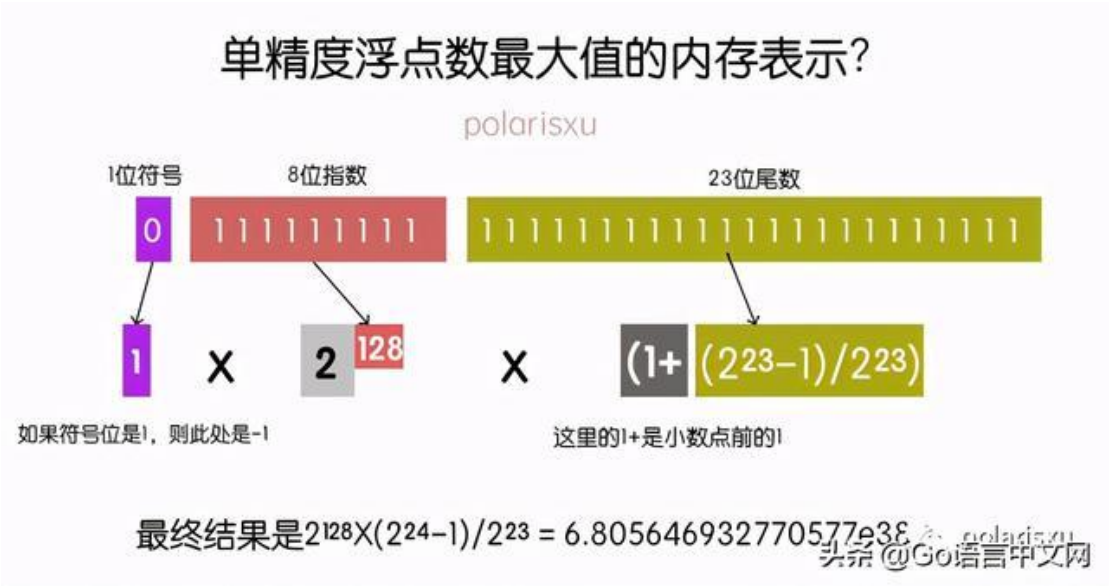
根据前面解释，十进制 0.1 转为二进制小数，得到 0.0001100…（重复 1100）这样一个循环二进制小数，用 IEEE754 表示如下图：

同样，0.2 用单精度浮点数表示是：0.200000002980232238769531 25。所以，0.1 + 0.2 的结果是：0.300000004470348358154296875。

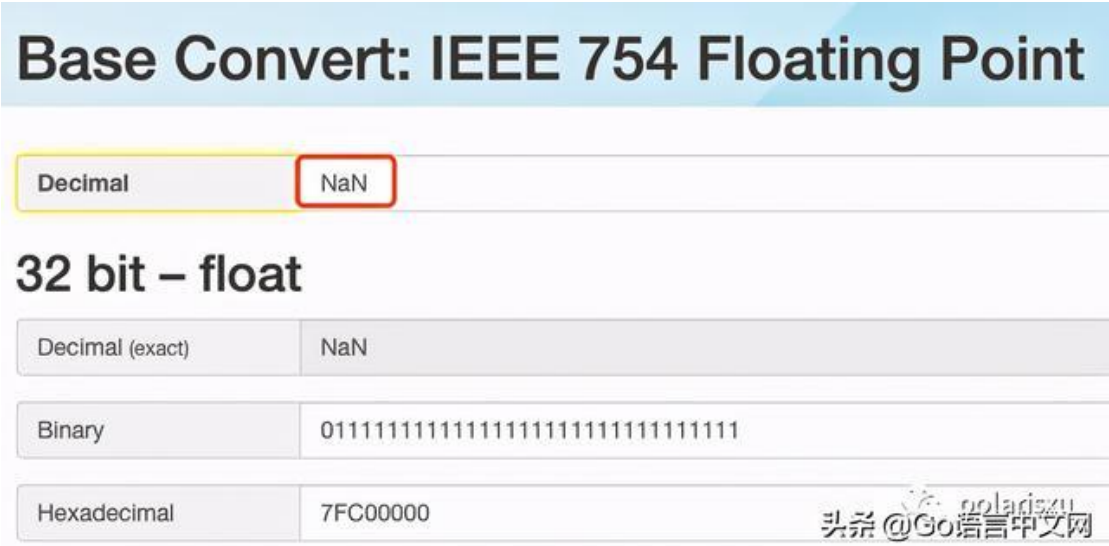
7、特殊值

单精度浮点数的最大值

根据前面知识，易想到它的最大值的内存应为



即：01111111111111111111111111111111。把此值填入 https://baseconvert.com/ieee-754-floating-point 中，发现结果是：



别急，因为凡是都有特殊。现在就讲讲浮点数中的特殊值。

特殊值 infinity（无穷）

当指数位全 1，尾数位全 0 时，这样的浮点数表示无穷。根据符号位，有正无穷和负无穷(+infinity 和-infinity)。为什么需无穷？因为计算机资源的限制，没法表示所有数，当一个数超过了浮点数的表示范围时，可用 infinity 来表示。而数学中也有无穷的概念。具体表示可定义一个常量，如：

正无穷：0x7FF0000000000000，负无穷：0xFFF0000000000000

和上面浮点数内存位模型强转 int 类似，这个执行相反操作，就得到该特殊浮点值。

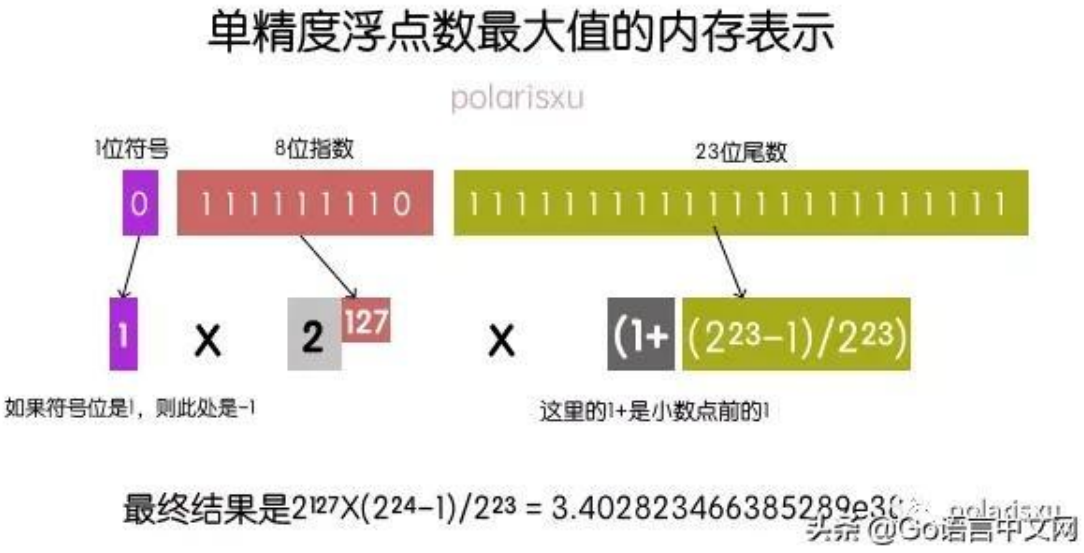
特殊值 NaN

NaN 是 not-a-number 的缩写，即不是一个数。为什么需要它？例如，当对 -1 进行开根号时，浮点数不知道如何进行计算，就会使用 NaN，表示不是一个数。NaN 的具体内存表示是：指数位全是 1，尾数位不全是 0。

小结

现在清楚上面单精度浮点数最大值不对，它是 NaN。画一张图，方便更清楚这些特殊值。

单精度浮点数的最大值应能确认，即：0 11111110 11111111111111111111111。



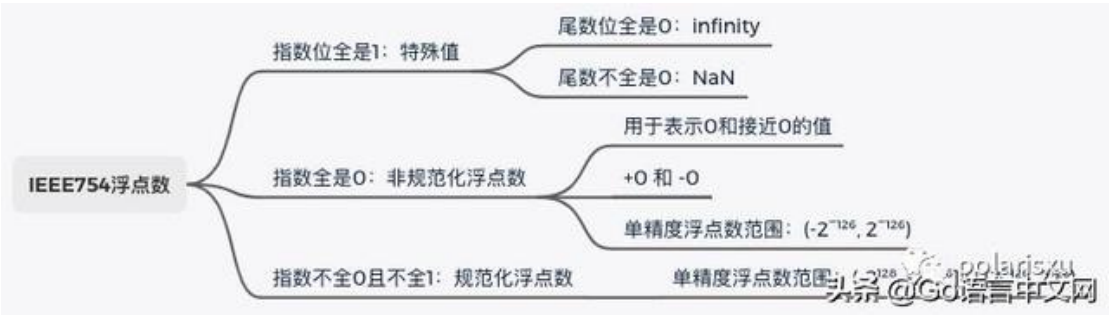
8、非规范化浮点数

单精度浮点数的最小值是多少（正数）？根据前面的知识，得这样的最小值 0 00000000 00000000000000000000001。根据前面规范化浮点数的规定，该值是 $2^{-127}\times(1+2^{-23})$。然而，最小值的内存表示没错，但算出来的结果错。为避免两个小浮点数相减结果是 0，也就是规范化浮点数无法表示，这样情况出现，同时根据规范化浮点数的定义，因为尾数部分有一个省略的前导 1，因此无法表示 0。所以，**IEEE754 规定另外一种浮点数：当指数位全是 0，尾数部分不全为 0，尾数部分没有省略的前导 1，同时指数部分的偏移值比规范形式的偏移值小 1，即单精度是-126，双精度是-2046。**这种形式的浮点数叫非规范化浮点数（denormal number）。因此单精度浮点数的最小值（正数）如下图：

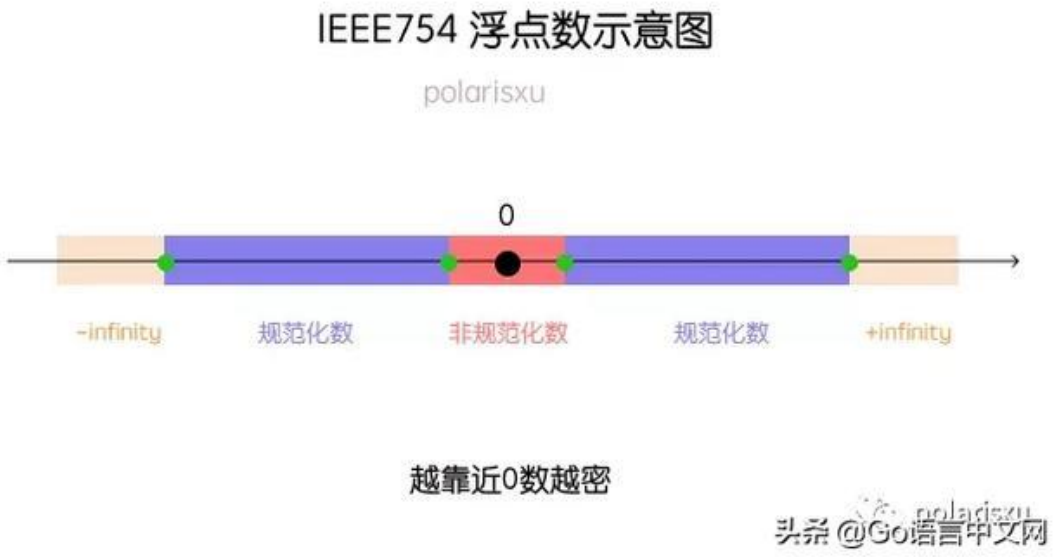
有非规范化浮点数，IEEE754 可表示 0，但会存在+0 和-0：即所有位全是 0 时是+0；符号位是 1，其它位是 0 时是-0。

9、IEEE754 浮点数分类小结

IEEE754 浮点数，指数是关键，根据指数，将其分为：特殊值、非规范化浮点数和规范化浮点数。



从上图规范化和非规范化浮点数的表示范围可以看出，两种类型的表示是具有连续性的。这也就是为什么非规范化浮点数指数规定为比规范形式的偏移值小1，即单精度为-126，双精度为-2046。在数轴上，浮点数的分布：



10、总结

其它一些知识点，如浮点数的运算、不满足结合律、四舍但五不一定入等，有兴趣的可查相关资料。

参考资料或相关链接

https://floating-point-gui.de/

https://www.geeksforgeeks.org/ieee-standard-754-floating-point-numbers/

https://baseconvert.com/

这个交互式工具，很不错：http://evanw.github.io/float-toy/

https://bartaz.github.io/ieee754-visualization/

柴大：https://mp.weixin.qq.com/s/0lCte3UD5qYcaBnebwnYrQ

左神：https://mp.weixin.qq.com/s/QsEe34pcimNdqCb99h44cQ

图书《程序是怎样跑起来的》

程序定点化

8.1 数的定标

在定点处理器中，用定点数进行数值运算，其操作数一般用整型数表示。一个整型数的最大表示范围取决于处理器所给定的字长，一般为 16 位或 32 位。显然，字长越长，所能表示的数的范围越大，精度也越高。如无特别说明，以下以 16 位字长为例。

处理器的数以 2 的补码形式表示。每个 16 位数用一个符号位来表示数的正负，0 表示数值为正，1 则表示数值为负。其余 15 位表示数值的大小。因此二进制数 0010000000000011 表示 8195；二进制数 1111111111111100 表示-4。

对处理器而言，参与数值运算的数就是 16 位的整型数。但在许多情况下，数学运算过程中的数不一定都是整数。那么，处理器是如何处理小数的呢？应该说，处理器本身无能为力。那么是不是说处理器就不能处理各种小数呢？当然不是。这其中的关键就是由程序员来确定一个数的小数点处于 16 位中的哪一位。这就是数的定标。

表 1　Q 表示、S 表示及数值范围

Q 表示	S 表示	十进制数表示范围	Q 表示	S 表示	十进制数表示范围
Q15	S0.15	-1≤x≤0.9999695	Q7	S8.7	-256≤x≤255.9921875
Q14	S1.14	-2≤x≤1.9999390	Q6	S9.6	-512≤x≤511.9804375
Q13	S2.13	-4≤x≤3.9998779	Q5	S10.5	-1024≤x≤1023.96875
Q12	S3.12	-8≤x≤7.9997559	Q4	S11.4	-2048≤x≤2047.9375
Q11	S4.11	-16≤x≤15.9995117	Q3	S12.3	-4096≤x≤4095.875
Q10	S5.10	-32≤x≤31.9990234	Q2	S13.2	-8192≤x≤8191.75
Q9	S6.9	-64≤x≤63.9980469	Q1	S14.1	-16384≤x≤16383.5
Q8	S7.8	-128≤x≤127.9960938	Q0	S15.0	-32768≤x≤32767

通过设定小数点在 16 位数中的不同位置，就可以表示不同大小和不同精度的小数了。数的定标有 Q 表示法和 S 表示法两种。表 1 列出了一个 16 位数的 16 种 Q 表示、S 表示及它们所能表示的十进制数值范围。从表 1 可以看出，同样一个 16 位数，若小数点设定的位置不同，它所表示的数也就不同。例如：16 进制数 2000 H 用 Q0 表示代表 8192， 16 进制数用 Q15 表示代表 0.25， 但对于处理器来说，处理方法是完全相同的。从表 1 还可以看出，不同的 Q 所表示的数不仅范围不同，而且精度也不相同。Q 越大，数值范围越小，但精度越高；相反，Q 越小，数值范围越大，但精度就越低。例如，**Q0 的数值范围是-32768 到+32767，其精度为 1，而 Q15 的数值范围为-1 到 0.9999695，精度为 1/32768=0.00003051。**因此，对定点数而言，数值范围与精度是一对矛盾，一个变量要想能够表示比较

大的数值范围，必须以牺牲精度为代价；而想提高精度，则数的表示范围就相应地减小。在实际的定点算法中，为了达到最佳的性能，必须充分考虑到这一点。浮点数与定点数的转换关系可表示为：浮点数(x)转换为定点数(x_q)：x_q = (int)x*2^Q；定点数(x_q)转换为浮点数(x)：x = (float) x_q*2^{-Q}。例如，浮点数 x=0.5，定标 Q=15，则定点数 $x_q=\lfloor 0.5\times 32768\rfloor=16384$，式中⌊ ⌋表示下取整。反之，一个用 Q=15 表示的定点数 16384，其浮点数为 16384×2^{-15}=16384/32768=0.5。

8.2 从浮点到定点

编算法时，为方便，一般多用 C 语言。程序中所用的变量一般既有整型数，又有浮点数。如例 1 程序中的变量 i 是整型数，而 pi 是浮点数，hamwindow 则是浮点数组。

例 1：256 点汉明窗计算

```
int  i;
float     pi=3.14159，hamwindow[256];
for(i=0;i<256;i++)    hamwindow[i]=0.54-0.46*cos(2.0*pi*i/255);
```

若将上述程序用定点处理器实现，则需将 C 语言浮点算法改为定点算法。下面讨论基本算术运算的定点实现方法。

1、加法/减法运算的 C 语言定点模拟

设浮点加法运算的表达式为：

```
float x,y,z;
z=x+y;
```

将浮点加法/减法转化为定点加法/减法时最重要的一点就是必须保证两个操作数的定标值一样。若两者不一样，则在做加法/减法运算前先进行小数点的调整。**为保证运算精度，需使 Q 值小的数调整为与另一个数的 Q 值一样大//【大 Q 值 精度高 因此尽量向大 Q 靠拢】。**此外，在做加法/减法运算时，必须注意结果可能会超过 16 位表示。若加法/减法的结果超出 16 位的表示范围，则必须保留 32 位结果，以保证运算的精度。

(1)结果不超过 16 位表示范围

设 x 的 Q 值为 Q_x，y 的 Q 值为 Q_y，且 Q_x >Q_y，加法/减法结果 z 的定标值为 Q_z，则 z=x+y 可写为 $z_q2^{-Qz}=x_q2^{-Qx}+y_q2^{-Qy}=x_q2^{-Qx}+y_q2^{(Qx-Qy)}2^{-Qx}=[x_q+y_q2^{(Qx-Qy)}]2^{-Qx}$，因此，$z_q=[x_q+y_q2^{(Qx-Qy)}]\,2^{(Qz-Qx)}$。

所以定点加法可以描述为：

```
int x,y,z;
```

```
long temp;  /*临时变量*/
temp＝y<<(Qx－Qy);
temp＝x＋temp;
if(Qx≥Qz)   z＝(int)(temp>>(Qx－Qz));
if(Qx<Qz)   z＝(int)(temp<<(Qz－Qx));
```

例　定点加法中，设 x＝0.5，y＝3.1，则浮点运算结果为 z=x+y=0.5+3.1=3.6; Q_x=15，Q_y=13，Q_z=13，则定点加法为：

```
x=16384；y=25395;
temp=25395<<2=101580;
temp=x+temp=16384+101580=117964;
z=(int)(117964L>>2)=29491;
```

因为 z 的 Q 值为 13，所以定点值 z=29491 即为浮点值 z=29491/8192=3.6。

例 2：定点减法中，设 x=3.0，y=3.1，则浮点运算结果为 z=x-y=3.0-3.1=-0.1; Qx=13，Qy=13，Qz=15，则定点减法为：

```
x=24576；y=25295；
temp=25395;
temp=x-temp=24576-25395=-819;
```

因为 Qx < Qz，故 z=(int)(-819<<2)=-3276。由于 z 的 Q 值为 15，所以定点值 z=-3276 即为浮点值 z=-3276/32768≈-0.1。

(2)结果超过 16 位表示范围

设 x 的 Q 值为 Q_x，y 的 Q 值为 Q_y，且 Q_x >Q_y，加法结果 z 的定标值为 Q_z，则定点加法为：

```
int x，y；
long temp，z；
temp=y<<(Qx-Qy)；
temp=x＋temp;
if(Qx≥Qz)   z=temp>>(Qx-Qz);
if(Qx<Qz)    z=temp<<(Qz-Qx);
```

例：结果超过 16 位的定点加法中：设 x=15000，y=20000，则浮点运算值为 z=x＋y=35000，显然 z>32767，因此 Q_x=1，Q_y=0，Q_z=0，则定点加法为：

```
x=30000；y=20000；
temp=20000<<1=40000;
temp=temp+x=40000+30000=70000;
z=70000L>>1=35000;
```

因为 z 的 Q 值为 0，所以定点值 z=35000 就是浮点值，这里 z 是一个长整型数。

当加法或加法的结果超过 16 位表示范围时，若程序员事先能了解到这种情况，且需保证运算精度时，则须保持 32 位结果。若程序中是按照 16 位数运算，则超过 16 位实际上是出现溢出。若不采取适当的措施，则数据溢出会导致运算精度的严重恶化。**一般的定点处理器都设有溢出保护功能，当溢出保护功能有效时，一旦出现溢出，则累加器的结果为最大的饱和值(上溢为 7FFFH，下溢为 8000H)，从而达到防止溢出引起精度严重恶化的目的。**

2、乘法运算的 C 语言定点模拟

设浮点乘法运算的表达式为：

```
float x,y,z;
z = xy;
```

设经分析后，x 的定标值为 Q_x，y 的定标值为 Q_y，乘积 z 的定标值为 Q_z，则由 $z= xy$，$z_q2^{-Qz} =x_qy_q2^{-(Qx+Qy)}$，因此，$z_q=(x_qy_q)2^{Qz-(Qx-Qy)}$。所以定点表示的乘法为：

```
int x,y,z;
long temp;
temp = (long)x;
z = (temp*y) >> (Qx+Qy-Qz);
```

例 3：定点乘法中，设 x = 18.4，y = 36.8，则浮点运算值为 z =18.4×36.8 = 677.12; 根据上节，得 $Q_x = 10$，$Q_y = 9$，$Q_z = 5$，所以

```
x = 18841；y = 18841；
temp = 18841L;
z = (18841L*18841)>>(10+9-5) = 354983281L>>14 = 21666;
```

因为 z 的定标值为 5，故定点 z = 21666 即为浮点的 z = 21666/32 = 677.08。

3、除法运算的 C 语言定点模拟

设浮点除法运算的表达式为：

```
float x,y,z;
z = x/y;
```

设经分析后，被除数 x 的定标值为 Q_x，除数 y 的定标值为 Q_y，商 z 的定标值为 Q_z，则 $z = x/y$，$z_q2^{-Qz} =(x_q2^{-Qx})/(y_q2^{-Qy})$，因此，$z_q=(x_q2^{(Qz-Qx+Qy)})/y_q$ 。所以定点表示的除法为：

```
int x,y,z;
long temp;
```

temp = (long)x;
z = (temp<<(Qz-Qx+Qy))/y;　　//**【大 Q 值 精度高 因此尽量向大 Q 靠拢】**

例 4：定点除法中，设 x = 18.4，y = 36.8，浮点运算值为 z = x/y = 18.4/36.8 = 0.5; 根据上节，得 Q_x = 10，Q_y = 9，Q_z = 15；所以有

x = 18841, y = 18841;
temp = (long)18841;
z = (18841L<<(15-10+9))/18841 = 308690944L/18841 = 16384;

因为商 z 的定标值为 15，所以定点 z = 16384 即为浮点 z = $16384/2^{15}$= 0.5。

4、程序变量的 Q 值确定

在前面介绍中，由于 x、y、z 的值已知，因此从浮点变为定点时 Q 值很好确定。在实际应用中，程序中参与运算的都是变量，那么如何确定浮点程序中变量的 Q 值？从前面的分析可以知道，确定变量的 Q 值实际上就是确定变量的动态范围，动态范围确定了，则 Q 值就确定了。

设变量的绝对值的最大值为|max|，注意|max|必须小于或等于 32767。取一个整数 *n*，使它满足 2^{n-1} <|max|<2^n，则有 2^{-Q}=$2^{-15}\times 2^n$ =$2^{-(15-n)}$，Q = 15-*n*。//**【前面的位数大于等于 n，因此 Q 值是 15-n　　15 表示前面有 0 位】**

例如，某变量的值在-1 至＋1 之间，即|max|<1，因此 *n* = 0，Q＝15-*n* = 15。

确定了变量的|max|就可以确定其 Q 值，那么变量的|max|又是如何确定的呢？一般来说，确定变量的|max|有两种方法：一种是理论分析法，另一种是统计分析法。

(1)理论分析法

有些变量的动态范围通过理论分析是可以确定的。例如：

三角函数，*y* = sin(*x*)或 *y* = cos(*x*)，由三角函数知识可知，|*y*|≤1；

汉明窗，*y*(*n*) = 0.54-0.46cos [2π*n*/(*N*-1)] ，0≤*n*≤*N*-1。因为-1≤cos [2π*n*/(*N*-1)] ≤1，所以 0.08≤*y*(*n*)≤1.0；

FIR 卷积。$y(n)=\sum_{k=0}^{N-1}h(k)x(n-k)$，设 $\sum_{k=0}^{N-1}|h(k)|=1.0$，且 *x*(*n*)是模拟信号 12 位量化值，即有 $|x(n)|\leqslant 2^{11}$，则$|y(n)|\leqslant 2^{11}$；

理论已经证明，在自相关线性预测编码(LPC)的程序设计中，反射系数 k_i 满足下列不等式：|k_i|<1，*i* = 1, 2, …, *p*,　*p* 为 LPC 的阶数。

(2)统计分析法

对于理论上无法确定范围的变量，一般采用统计分析的方法来确定其动态范围。所谓统计分析，是用足够多的输入信号样值来确定程序中变量的动态范围，这里输入信号一方面要有一定的数量，另一方面必须尽可能地涉及各种情况。例如，在语音信号分析中，统计分析时就必须采集足够多的语音信号样值，并且在所采集的语音样值中，应尽可能地包含各种情况，如音量的大小、声音的种类（男声、女声等）。只有这样，统计出来的结果才能具有典型性。当然，统计分析毕竟不可能涉及所有可能发生的情况，因此，对统计得出的结果在程序设计时可采取一些保护措施，如适当牺牲一些精度，Q 值取比统计值稍大些，用处理器提供的溢出保护功能等。

5、浮点至定点变换的 C 程序举例

本节通过例子说明 C 程序从浮点变换至定点的方法。这是一个对语音信号(0.3kHz～3.4kHz)进行低通滤波的 C 语言程序，低通滤波的截止频率为 800Hz，滤波器采用 19 点的有限冲击响应 FIR 滤波。语音信号的采样频率为 8kHz，每个语音样值按 16 位整型数存放在 insp.dat 文件中。

例　语音信号 800Hz 19 点 FIR 低通滤波 C 语言浮点程序

```
#include <stdio.h>
int length = 180  /*语音帧长为 180 点＝22.5ms@8kHz 采样*/
void    filter(int xin[ ],int xout[ ],int n,float h[ ]); /*滤波子程序说明*/

/*19 点滤波器系数*/
float h[19]={0.01218354，-0.009012882，-0.02881839，-0.04743239，-0.04584568，
-0.008692503，0.06446265，0.1544655,0.2289794，0.257883，0.2289794，0.1544655，
0.06446265，-0.008692503，-0.04584568，-0.04743239，-0.02881839，-0.009012882，
0.01218354};
int x1[length+20];

/*低通滤波浮点子程序*/
void    filter(int xin[ ],int xout[ ],int n,float h[ ])
{
      int i，j;
      float sum;
      for(i=0;i<length;i++) x1[n+i-1]=xin[i];
      for (i=0;i<length;i++)
      {
            sum=0.0;
            for(j=0;j<n;j++)    sum+=h[j]*x1[i-j+n-1];
            xout[i]=(int)sum;
      }
      for(i=0;i<(n-1);i++) x1[n-i-2]=xin[length-1-i];
}
```

```
/*主程序*/
void    main( )
{
      FILE    *fp1,*fp2;
      int       frame,indata[length],outdata[length];
      fp1=fopen(insp.dat,"rb");            /*输入语音文件*/
      fp2=fopen(outsp.dat,"wb");          /*滤波后语音文件*/

      frame=0;
      while(feof(fp1)==0)
      {
            frame++;
            printf("frame=%d\n",frame);
            for(i=0;i<length;i++)    indata[i]=getw(fp1);    /*取一帧语音数据*/
            filter(indata,outdata,19,h);          /*调用低通滤波子程序*/
            for(i=0;i<length;i++)    putw(outdata[i],fp2);    /*将滤波后的样值写入文
件*/
      }
      fcloseall(fp1);    fcloseall(fp2);          /*关闭文件*/
}
```

例　语音信号 800Hz 19 点 FIR 低通滤波 C 语言定点程序

```
#include <stdio.h>
int length=180;
void    filter(int xin[ ],int xout[ ],int n,int h[ ]);
int    h[19]={399,-296,-945,-1555,-1503,-285,2112,5061,7503,8450,
          7503,5061,2112,-285,-1503,-1555,-945,-296,399};          /*Q15*/
int    x1[length+20];
/*低通滤波定点子程序*/
void    filter(int xin[ ],int xout[ ],int n,int h[ ])
{
      int i,j;
      long sum;
      for(i=0;i<length;i++) x1[n+i-1]=xin[i];
      for (i=0;i<length;i++)
      {
            sum=0;
            for(j=0;j<n;j++)    sum+=(long)h[j]*x1[i-j+n-1];
            xout[i]=sum>>15;
      }
      for(i=0;i<(n-1);i++) x1[n-i-2]=xin[length-i-1];
}
```

主程序与浮点的完全一样。

8.3 DSP 定点算术运算

定点处理器的数值表示是基于 2 的补码表示形式。每个 16 位数用 1 个符号位、i 个整数位和 15-i 个小数位来表示。因此数 00000010.10100000 表示的值为 $2^1+2^{-1}+2^{-3}=2.625$，这个数可用 Q8 格式（ 8 个小数位）来表示，它表示的数值范围为-128～+127.996，一个 Q8 定点数的小数精度为 1/256=0.004。

虽然特殊情况（如动态范围和精度要求）必须使用混合表示法，但是，常见的是全以 Q15 格式表示的小数或以 Q0 格式表示的整数来工作。这一点对于主要是乘法和累加的信号处理算法特别现实，小数乘以小数得小数，整数乘以整数得整数。当然，乘积累加时可能会出现溢出现象，在这种情况下，程序员应当了解数学里面的物理过程以注意可能的溢出情况。下面讨论乘法、加法和除法的 DSP 定点运算，汇编程序以 TMS320C5410 为例。

1、定点乘法

2 个定点数相乘时可以分为下列 3 种情况：

(1)小数乘小数

Q15×Q15＝Q30； 例 0.5*0.5 = 0.25

```
    0.100000000000000       Q15
×   0.100000000000000       Q15
00.010000000000000000000000000000=0.25     Q30
```

2 个 Q15 的小数相乘后得到 1 个 Q30 的小数，即有 2 个符号位。一般情况下相乘后得到的满精度数不必全部保留，而只需保留 16 位单精度数。由于相乘后得到的高 16 位不满 15 位的小数精度，为了达到 15 位精度，可将乘积左移 1 位，下面是上述乘法的 TMS320C5410 程序：

```
st   #x,ar0
st   #y,ar1
ld   *ar0,t
mpy *ar1,a
```

(2)整数乘整数

Q0×Q0 = Q0；例 17×(-5)=-85

```
    0000000000010001          =17
×   1111111111111011          =-5
11111111111111111111111110101011          =-85
```

(3)混合表示法

许多情况下，运算过程中为了既满足数值的动态范围又保证一定的精度，就必须采用 Q0 与 Q15 之间的表示法。比如，数值 1.2345，显然 Q15 无法表示，而若用 Q0 表示，则最接近的数是 1，精度无法保证。因此，数 1.2345 最佳的表示法是 Q14。

例 1.5×0.75 = 1.125

```
    01.10000000000000          =1.5    Q14
×   00.11000000000000          =0.75 Q14
0001.00100000000000000000000000000      = 1.125    Q28
```

Q14 的最大值不大于 2，因此，2 个 Q14 数相乘得到的乘积不大于 4。

一般的，若一个数的整数位为 i 位，小数位为 j 位，另一个数的整数位为 m 位，小数位为 n 位，则这两个数的乘积为(i+m)位整数位和(j+n)位小数位。这个乘积的最高 16 位可能的精度为(i+m)整数位和(15-i-m)小数位。但是，若事先了解数的动态范围，就可以增加数的精度。例如，程序员了解到上述乘积不会大于 1.8，就可以用 Q14 数表示乘积，而不是理论上的最佳情况 Q13。

例 5：1.5×0.75 的 TMS320C5410 程序如下：

```
st   #x,ar0          6000H(1.5/Q14)
st   #y,ar1          3000H(0.75/Q14)
ld   *ar0,t
mpy *ar1,a
ld   #1,asm
ld   a,asm,a         2400H(1.125/Q13)
```

上述方法为了保证精度均对乘的结果舍位，结果所产生的误差相当于减去 1 个 LSB(最低位)。采用简单的舍入方法，可使误差减少二分之一。

2、定点加法

乘的过程中，程序员可不考虑溢出而只需调整运算中的小数点。而加法则是一个更加复杂的过程。首先，加法运算必须用相同的 Q 点表示；其次，程序员或者允许其结果有足够的高位以适应位的增长，或者必须准备解决溢出问题。若操作数仅为 16 位长，其结果可用双精度数表示。下面举例说明 16 位数相加的两种途径。

(1)保留 32 位结果

(2)调整小数点保留 16 位结果

加法运算最可能出现的问题是运算结果溢出。TMS320C5410 提供了溢出的标志位 OVM，此外，使用溢出保护功能可使累加结果溢出时累加器饱和为最大

的整数或负数。当然，即使如此，运算精度还是大大降低。因此，最好的方法是完全理解基本的物理过程并注意选择数的表达方式。

3、定点除法

在通用处理器中，一般不提供单周期的除法指令，为此必须采用除法子程序来实现。二进制除法是乘法的逆运算。乘法包括一系列的移位和加法，而除法可分解为一系列的减法和移位。下面来说明除法的实现过程。设累加器为 8 位，且除法运算为 10 除以 3。除的过程就是除数逐步移位并与被除数比较的过程，在每一步进行减法运算，若能减则将位插入商中。

(1) 除数的最低有效位对齐被除数的最高有效位。

00001010
－ 00011000
11110010

(2) 由于减法结果为负，放弃减法结果，将被除数左移一位再减。

00010100
－ 00011000
11111000

(3) 结果仍为负，放弃减法结果，被除数左移一位再减。

00101000
－ 00011000
00010000

(4) 结果为正，将减法结果左移一位后加 1 ，作最后一次减。

00100001
－ 00011000
00001001

(5) 结果为正，将结果左移一位加 1 得最后结果。高 4 位代表余数，低 4 位表示商。

00010011

即商为 0011=3，余数为 0001=1。

很多处理器无专门除法指令，但用条件减指令 SUBC 可完成有效灵活的除法功能。用这一指令的唯一限制是两个操作数均为正。程序员须事先了解其可能的运算数的特性，如其商是否可用小数表示及商的精度是否可算出来。

8.4 非线性运算的定点快速实现

在数值运算中，除基本的加减乘除运算外，还有其他许多非线性运算，如对数运算、开方运算、指数运算、三角函数运算等，**实现这些非线性运算的方法一般有：(1)调用 DSP 编译系统的库函数；(2)查表法；(3)混合法。**下面分别介绍这三种方法。

1、调用 DSP 编译系统的库函数

TMS320C5X 的 C 编译器提供了比较丰富的运行支持库函数。在这些库函数中，包含了诸如对数、开方、三角函数、指数等常用的非线性函数。在 C 程序中（也可在汇编程序中）只要采用与库函数相同的变量定义，就可以直接调用。例如，在库函数中，定义了以 10 为底的常用对数 log10( )：

```
＃include <math.h>
double    log10(double x);
```

在 C 程序中按如下方式调用：

```
float x,y;
x = 10.0;
y = log10(x);
```

从上例可以看出，库函数中的常用对数 log10( )要求的输入值为浮点数，返回值也为浮点数，运算的精度完全可以保证。直接调用库函数非常方便，但由于运算量大，很难在实时 DSP 中得到应用。

2、查表法

在实时 DSP 应用中实现非线性运算，一般都采取适当降低运算精度来提高程序的运算速度。查表法是快速实现非线性运算最常用的方法。采用这种方法必须根据自变量的范围和精度要求制作一张表格。显然输入的范围越大，精度要求越高，则所需的表格就越大，即存储量也越大。查表法求值所需的计算就是根据输入值确定表的地址，根据地址就可得到相应的值，因而运算量较小。**查表法比较适合于非线性函数是周期函数或已知非线性函数输入值范围这两种情况**，下面两例分别说明这两种情况。

例　已知正弦函数 y=cos(x)，制作一个 512 点表格，并说明查表方法。

由于正弦函数是周期函数，函数值在－1 至＋1 之间，用查表法比较合适。由于 Q15 的表示范围为－1 至 32767/32768 之间，原则上讲－1 至＋1 的范围必须用 Q14 表示。但一般从方便和总体精度考虑，类似情况仍用 Q15 表示，此时＋1 用 32767 来表示。

(1) 产生 512 点值的 C 语言程序如下所示:

```
#define    N    512
#define    pi    3.14159
int    sin_tab[512];
void    main( )
{
      int i;
      for(i=0;i<N;i++) sin_tab[i]=(int)(32767*sin(2*pi*i/N));
}
```

(2)查表

查表实际上就是根据输入值确定表的地址。设输入 x 在 0~2π之间，则 x 对应于 512 点表的地址为：index = (int)(512*x/2π)，则 y = sin(x) = sin_tab[index]。若 x 用 Q12 定点数表示，将 512/2π用 Q8 表示为 20861，则计算正弦表的地址的公式为：index = (x*20861L)>>20。

例　用查表法求以 2 为底的对数，已知自变量取值范围为 0.5~1,要求将自变量范围均匀划分为 10 等分。试制作这个表格并说明查表方法。

(1)做表：

y = log2(x)，由于 x 在 0.5 到 1 之间，因此 y 在-1 到 0 之间，x 和 y 均可用 Q15 表示。由于对 x 均匀划分为 10 段，因此，10 段对应于输入 x 的范围如表 3.2 所示。若每一段的对数值都取第 1 点的对数值，则表中第 1 段的对数值为 y0(Q15) = (int)(log2(0.5)×32768)，第 2 段的对数值为 y1(Q15) = (int)(log2(0.55)×32768)，依次类推。

表　logtab0 10 点对数表(输入 0.5~1)

地址	输入值	对数值(Q15)	地址	输入值	对数值(Q15)
0	0.50~0.55	-32768	5	0.75~0.80	-13600
1	0.55~0.60	-28262	6	0.80~0.85	-10549
2	0.60~0.65	-24149	7	0.85~0.90	-7683
3	0.65~0.70	-20365	8	0.90~0.95	-4981
4	0.70~0.75	-16862	9	0.95~1.00	-2425

(2)查表

查表时，先根据输入值计算表的地址，计算方法为：index=((x-16384)×20)>>15。式中，index 就是查表用的地址。例如，已知输入 x= 26869，则 index=6，因此 y=-10549。

3、混合法

(1)提高查表法的精度

上述方法查表所得结果的精度随表的大小而变化，表越大，则精度越高，但存储量也越大。当系统的存储量有限而精度要求较高时，查表法不太适合。能否在适当增加运算量的情况下提高非线性运算的精度呢？下面介绍一种查表结合少量运算来计算非线性函数的混合法，这种方法适用于在输入变量的范围内函数呈单调变化的情形。

混合法是在查表的基础上采用计算的方法以提高当输入值处于表格两点之间时的精度。提高精度的一个简便方法是采用折线近似法，如图 1 所示。以求以 2 为底的对数为例。设输入值为 x，则精确的对数值为 y，在表格值的两点之间作一直线，用 y′ 作为 y 的近似值，则有：

$$y' = y_0 + \Delta y$$

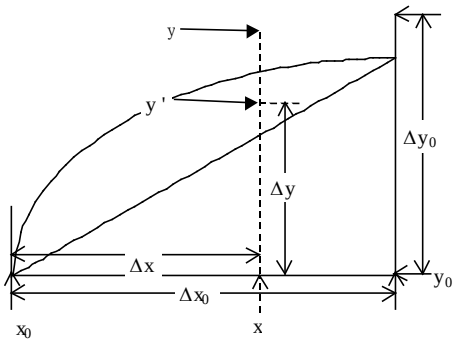


图 1　提高精度的折线近似法

其中 y_0 由查表求得。现在只需在查表求得 y_0 的基础上增加 Δy 即可。Δy 的计算方法如下：$\Delta y=(\Delta x/\Delta x_0)\Delta y=\Delta x(\Delta y_0/\Delta x_0)$。式中 $\Delta y_0/\Delta x_0$ 对每一段来说是一个恒定值，可作一个表格直接查得。此外计算 Δx 时需用到每段横坐标的起始值，这个值也可作一个表格。这样共有三个大小均为 10 的表格，分别为存储每段起点对数值的表 logtab0、存储每段 $\Delta y_0/\Delta x_0$ 值的表 logtab1 和存储每段输入起始值 x_0 的表 logtab2，表 logtab1 和表 logtab2 可用下列两个数组表示：

```
int logtab1[10]={22529, 20567, 18920, 17517, 16308, 15255, 14330, 13511, 12780, 12124};     /*Δy0/Δx0 : Q13*/
int logtab2[10]={16384, 18022, 19660, 21299, 22938, 24576, 26214, 27853, 29491, 31130};     /* x0: Q15*/
```

综上所述，采用混合法计算对数值的方法可归纳为：

①根据输入值，计算查表地址：index=((x-16384)×20)>>15;
②查表得 y_0=logtab0[index];
③计算Δx=x-logtab2[index];
④计算Δy＝(Δx×logtab1[index])>>13;
⑤计算得结果 $y=y_0+\Delta y$。

例　已知 x=0.54，求 log2(x)。

0.54 的精确对数值为 y=log2(0.54)=-0.889。

混合法求对数值的过程为：

①定标 Q15，定标值 x=0.54*32768=17694；
②表地址 index=((x-16384)×20)>>15=0;
③查表得 y0=logtab0[0]=-32768;
④计算Δx=x-logtab2[0]=17694-16384=1310;
⑤计算Δy＝(Δxlogtab1[0])>>13=(1310*22529L)>>13=3602;
⑥计算结果 y=y0+Δy=-32768+3602=-29166。

结果 y 为 Q15 定标，折算成浮点数为-29166/32768=-0.89，可见精度较高。

(2)扩大自变量范围

如上所述，查表法比较适用于周期函数或自变量的动态范围不是太大的情形。对于像对数这样的非线性函数，输入值和函数值的变化范围都很大。若输入值的变化范围很大，则作表就比较困难。那么能否比较好地解决这个问题，既不使表格太大，又能得到比较高的精度呢？下面讨论一种切实可行的方法。

设 x 是一个大于 0.5 的数，则 x 可以表示为下列形式：$x = m \cdot 2^e$。式中，$0.5 \leqslant m \leqslant 1.0$，e 为整数。则求 x 的对数可以表示为：

$$\log_2(x) = \log_2(m \cdot 2^e) = \log_2(m) + \log_2(2^e) = e + \log_2(m)$$

也就是说，求 x 的对数实际上只要求 m 的对数就可以了，而由于 m 的数值在 0.5~1.0 之间，用上面介绍的方法是完全可以实现的。例如：

$$\log_2(10000) = \log_2(0.61035 \times 2^{14}) = \log_2(0.61035) + 14 = 13.2877$$

可见，若一个数可以用比较简便的方法表示为上面的形式，则求任意大小数的对数也是比较方便的。TMS320C5X 指令集提供了一条用于对累加器 A 中的数进行规格化的指令 exp 和 norm，exp 指令的作用就是求出左移位数，即对累加器 A 中的数左移，直至数的最高位被移至累加器的第 30 位所需移的位数，norm 完成移位操作。例如，对数值 10000 进行规格化的 TMS320C5410 程序为：

```
ld   #10000,a
exp a
nop          //流水线延迟等待
norm a       //a=00 4E20 0000
```

例　实现以 2 为底的对数的 C 定点模拟程序

```
int logtab0[10]＝{-32768, -28262, -24149, -20365, -16862, -13600, -10549, -7683, -4981, -2425}; /*Q15*/
int logtab1[10]＝{22529, 20567, 18920, 17517, 16308, 15255, 14330, 13511, 12780, 12124};     /*Q13*/
int logtab2[10]＝{16384, 18022, 19660, 21299, 22938, 24576, 26214, 27853, 29491, 31130};     /*Q15*/
int log2_fast(int Am)
{
    int        point,point1;
    int        index,x0,dx,dy,y;
    point＝0;
    while(Am<16384)
    {
        point++;
        Am＝Am<<1;
```

```
    }   /*对 Am 进行规格化*/
    point1＝(15-point-4)*512;          /*输入为 Q4，输出为 Q9*/
    index＝((Am-16384)*20L)>>15;       /*求查表地址*/
    dx＝Am-logtab2[index];
    dy＝((long)dx*logtab1[index])>>13;
    y＝(dy+logtab0[index])>>6;         /*Q9*/
    y＝point1+y;
    return (y);
}
```

上述程序中，输入值 Am 采用 Q4 表示，输出采用 Q9 表示，若输入输出的 Q 值与上面程序中的不同，则应做相应的修改。