第23章　定点数和浮点数

日常生活中，有各种各样的数，整数、分数、百分数等等，我们无时无刻不与这些数打交道。如：用加班2.75小时获得的1倍半的钱来买半匣鸡蛋需支付8.25%的销售税。许多人对诸如此类的数都感到很适应，并不需要怎么在行，即使在听到“平均每个美国家庭有　2.6人”这样的统计数字的时候，也不会联想到2.6这个数字对人来说是不是要把人肢解了这样可怕的问题。

在计算机内存里，整数和分数的换算是常见的。存在计算机内存里的东西都是二进制位的形式，也就是说，都是二进制数。但有些数用位来表示比其他数用位来表示要容易一些。

我们使用位来表示数学上称为自然数而计算机编程人员称为正整型数的数，并介绍如何用2的补码来表示负整数，而这种方法很容易实现正数、负数的加法。下表列出了8位、16位、32位的正整数及它们的2的补码的范围：

数的位数	正整数范围	2的补码范围
8	0～255	－128～127
16	0～65 535	-32 768～32 767
32	0～4 294 967 295	-2 147 483 648～2 147 483 647

要介绍的就是这些。除了整数以外，数学上还定义了有理数，它们可表示成两个整数的比，这个比也叫分数。例如，3/4是一个有理数，因为它是3与4的比。可以把这个数写成小数形式0.75，当写成小数时，它真正表示了分数，在此为75/100。

回忆一下第7章里的小数系统，在小数点左边的数字与10的整数次幂相关联；同样，在小数点右边的数字与10的负整数次幂相关联。第7章用42 705.684作为例子，该数可以表示成与下面与之相等的形式：

4×10 000＋
2×1000＋
7×100＋
0×10＋
5×1＋
6÷10＋
8÷100＋
4÷1000

注意一下除号，可以把这个序列写成没有除号的形式：

4×10 000＋
2×1000＋
7×100＋
0×10＋

$$5\times1+$$
$$6\times0.1+$$
$$8\times0.01+$$
$$4\times0.001$$

最后，可以用10的幂的形式表示如下：

$$4\times10^{4}+$$
$$2\times10^{3}+$$
$$7\times10^{2}+$$
$$0\times10^{1}+$$
$$5\times10^{0}+$$
$$6\times10^{-1}+$$
$$8\times10^{-2}+$$
$$4\times10^{-3}$$

有些分数并不容易用小数表示，常见的如1/3。如果用3去除1，可以得到：

0.333333333333333333333.....

而永无止境。我们通常写成简洁形式，在3上面加一道横线来表示无限循环：

$$0.\overline{3}$$

即使这样，把1/3写成小数也是有些笨拙的。它还是一个分数，因为它是两个整数的比。同样，1/7是：

0.142857142857142857..... 或$0.\overline{142857}$

无理数则更不同，如2的平方根。无理数不能表示成两个整数的比，也就是说，小数部分是无穷的，没有重复规律或固定模式：

$\sqrt{2}$=1.414213 562373095048801688724209698078569671875376 95...

2的平方根是下面这个代数方程的根：

$$x^{2} - 2 = 0$$

如果一个数不是以整数为系数的代数方程的根，则称为超越数（所有的超越数为无理数，但并不是所有的无理数都是超越数）。超越数包括π，它是圆的周长与直径的比，近似值为：

3.14159265358979328462643383279502884197169399375 11.....

另一个超越数是e，它是下面表达式：

$$\left(1+\frac{1}{n}\right)^{n}$$

当n趋近于无穷大时的近似值：

2.71828182845904523536028747135266249775724709369996...

到现在为止，谈到的所有数——有理数和无理数——统称为实数。这种定义用来与虚数相区分。虚数是负数的平方根，复数是由虚数和实数组成的。不管名称如何，虚数揭示了现实世界的奥秘，可以用来（例如）解决电子学的一些高级问题。

习惯上，我们把数看成是连续的。如果给出两个有理数，则可以找出一个数在这两个数中间。实际上，只需取平均值即可。但是，数字计算机不能处理连续事件。位不是 0就是1，

没有中间值。由于这一特性，数字计算机必须处理离散值。可以表示的离散值的个数直接与可达到的二进制位数相关。例如：如果用32位来存放正整数，则可以存放0～4 294 967 295个整数。如果需要存放4.5这个数，则必须重新考虑一种方法并做一些改动。

小数可以表示成二进制吗？是的，可以。最容易的方法可能是二进制编码的十进制(BCD)。前面第19章讲到BCD是十进制数的二进制编码，每一个十进制数字（0、1、2、3、4、5、6、7、8和9）需要4位，如下表所示：

十进制数字	二进制数字
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

BCD码特别适用于用美元和美分表示的与钱数有关的计算机程序。银行和保险公司是两个典型的多与钱打交道的行业，对这类公司的计算机程序来说，许多分数只需要两个十进制数位。

通常1个字节存储两个BCD数字，有时将这称为压缩BCD码。2的补码不与BCD一起使用，因此，压缩BCD码通常要有额外的一位(称作符号位)来标明是正数还是负数。一个BCD数存入整个字节比较方便，所以，小小的符号位通常需要牺牲4位或8位的存储空间。

来看一个例子。假定计算机要处理的钱数不会超过正、负1000万，换句话说，需要表示的钱数的范围从－9 999 999.99～9 999 999.99，则存储在内存的每一笔钱数需要用5个字节来表示。例如，－4 325 120.35用5个字节表示为：

00010100 00110010 01010001 00100000 00100101

或用十六进制表示为：

14h 32h 51h 20h 25h

注意最左边的1用来表示负数，即符号位。如果是正数，则该位为0。每一个数字需要4位，从十六进制值中可以直接看到。

如果需要表示的数的范围从－99 999 999.99～99 999 999.99，则需要6个字节——10个数字占5个字节，另一个字节仅用来表示符号位。

这类存储和标记方法也称作定点格式，因为小数点通常固定在特定的位置——本例中，小数点在两个小数位之前。注意，实际上并没有什么东西与数一起存放用来标明小数点的位置。处理定点格式数的程序应该知道小数点在哪里。定点数可以有任意个小数位数，在同一计算机程序里可以混用这些数字，但是对这些数进行算术运算的那部分程序必须知道小数点的位置。

定点格式只在知道这些数不会超过预先确定的内存单元，且没有太多小数位的场合比较适用。在数可能很大或可能很小的场合定点格式完全不适用。假设保留一个内存区域用来存储以英尺为单位的距离，则存在的问题是距离可能超出范围。从地球到太阳的距离是490 000 000 000

英尺，氢原子的半径为0.00000000026英尺，则你需要12字节的定点存储空间来容纳这些可能很大也可能很小的数值。

如果你还记得科学家和工程师们喜欢用称为“科学记数法”的系统来表示数的话，你也许已找到更好的存储此类数的方法。科学记数法特别适用于表示很大和很小的数，因为它采用10的幂方法从而不用写很长的一串0。采用科学记数法后，数字

490 000 000 000 写成 4.9×10^{11}

数字

0.00000000026 写成2.6×10^{-10}

在这两个例子里，数字4.9和2.6称作小数部分或首数，有时也称作有效数（尽管这个词更适用于对数运算）。为了与计算机术语相协调，在这儿把科学记数法的这一部分称作有效数。

指数部分是10的幂。在第一个例子中，指数是11；在第二个例子中，指数是－10。指数用来指明有效数的小数点要移动的位数。

为方便起见，有效数通常大于或等于1而小于10。尽管下面的数字是相等的：

$$4.9\times10^{11}=49\times10^{10}=490\times10^{9}=0.49\times10^{12}=0.049\times10^{13}$$

但我们选用第一种格式。这种格式也称作科学记数法的规格化格式。

注意，指数符号只是标明数的大小而并不表示数本身是正的还是负的。下面是用科学记数法表示的两个负数的例子：

-5.8125×10^{7} 等于－58 125 000

和

-5.8125×10^{-7} 等于－0.00000058125

在计算机中，对应于定点表示法的是浮点表示法。浮点格式用来存储较小或较大的数比较理想，因为它是以科学记数法为基础的。但是，计算机中采用的浮点格式是用科学记数法表示的二进制数。这里首先要提到的是如何用二进制表示小数数字。

实际上，这比设想的要容易，在十进制表示中，小数点右边的数字具有10的负整数次幂；在二进制表示中，二进制小数点（也仅是一个点，看起来与十进制小数点一样）右边的数具有2的负整数次幂。例如，一个二制数：

101.1101

可以用以下表达式转换成十进制：

1×4＋
0×2＋
1×1＋
1÷2＋
1÷4＋
0÷8＋
1÷16

除号可以用2的负整数次幂替换：

$$1\times2^{2}+$$
$$0\times2^{1}+$$
$$1\times2^{0}+$$
$$1\times2^{-1}+$$
$$1\times2^{-2}+$$
$$0\times2^{-3}+$$
$$1\times2^{-4}$$

或者，2的负整数次幂可以从1开始重复除以2来计算：

$$1\times4+$$
$$0\times2+$$
$$1\times1+$$
$$1\times0.5+$$
$$1\times0.25+$$
$$0\times0.125+$$
$$1\times0.0625$$

通过这些计算得到101.1101等效的十进制数5.8125。

在十进制科学记数法中，规格化有效数通常大于或等于 1而小于10。同样，二进制科学记数法的规格化有效数也通常大于或等于 1而小于10（即十进制中的2）。所以，按二进制科学记数法，数

101.1101 表示成 1.011101×2^{2}

这个规则隐含了一件有趣的事实：通常二进制浮点数在二进制小数点的左边除了 1以外再没有别的了。

现代计算机和计算机程序按照 IEEE在1985年制定的标准来处理浮点数，这个标准也为ANSI(the American national standards institute，美国国家标准局)所认可。ANSI/IEEE Std754-1985称作IEEE二进制浮点数算术运算标准。它并不像一般标准那样长，只有 18页，但却奠定了以方便的方式编码二进制浮点数的基础。

IEEE浮点数标准定义了两个基本格式：单精度格式，需要 4个字节；双精度格式，需要 8个字节。

首先看一下单精度格式，它有三部分： 1位个符号位（0表示正，1表示负）、8位的指数位和23位的有效数位。如下所示，最低有效数在最右边：

s＝1位符号	e＝8位指数	f＝23位有效数

总共有32位，4个字节。因为规格化二进制浮点数的有效数通常在二进制小数点左边为 1，所以在IEEE格式中这一位不包含在浮点数的存储空间中。有效数的 23位小数部分是反被存储的部分，所以，即使只有 23位用来存储有效数，精度仍然认为是 24位的。过一会儿将要看到24位精度的意义。

8位指数范围从0～255，称为移码指数，意思是必须从指数中减去一个数（称为偏移量）才能确定有符号指数的实际值。对单精度浮点数，偏移量为 127 。

指数0和255用于特殊用途，在此简单描述一下。如果指数从1变化到254，则由s（符号位）、e（指数）和f（有效数）来表示的数为：

$$(-1)^{s}\times 1.f\times 2^{e-127}$$

－1的s次幂是数学上的一种方法，意思是“如果s为0，则数是正的（因为任何数的0次幂等于1）；如果s为1，则数是负的（因为－1的1次幂为－1）”。

表达式的另一部分是1.f，意思是1后面为二进制小数点，再后面为23位的有效小数部分。它乘以2的幂，其中指数为内存中的8位移码指数减去127。

注意，到现在还没有提到如何表示一个很常见的数字，那就是 0。这是一种特殊情况，即：

- 如果e等于0，且f等于0，则数为0。通常，所有32位均为0则表示0。但是符号位可以是1，在这种情况下，数被解释为－0。－0可以表示一个很小的数，小到在单精度格式中不能用数字和指数来表示。尽管如此，它们然小于0。
- 如果e等于0，且f不等于0，则数是有效的。但是，它不是规格化的数，它等于

$$(-1)^{s}\times 0.f\times 2^{-127}$$

注意，二进制小数点左边的有效数为0。

- 如果e等于255，且f等于0，则数为正或负无穷大，这取决于符号s。
- 如果e等于255，且f不等于0，该值被认为“不是一个数”，简写为NaN。NaN可以表示一个不知道的数或者一个无效操作的结果。

通常，单精度浮点格式中可以表示的最小规格化的正或负二进制数为：

$$1.00000000000000000000000_{TWO}\times 2^{-126}$$

在二进制小数点之后有23个0。在单精度浮点格式中可以表示的最大规格化的正或负二进制数为：

$$1.11111111111111111111111_{TWO}\times 2^{127}$$

换算或十进制，这两个数近似为$1.175494351\times 10^{-38}$和$3.402823466\times 10^{38}$。这就是单精度浮点数表示法的有效范围。

前面讲过，10位二进制数近似等于3位十进制数。也就是说，若10位都置1(即十六进制为3FFh，十进制为1023)，则它近似等于3位十进制都设置为9，即999。或者

$$2^{10}\approx 10^{3}$$

这种关系表明按单精度浮点格式存放的24位二进制数大约与7位十进制数等效。因此，也可以说单精度浮点格式提供24位二进制精度，或大约7位十进制精度。它的含义是什么呢？

当观察定点数的时候，数的精度是很显然的。例如，对于钱数，用两位十进制小数的定点数就可精确到分。但是，对浮点数来说，就不能这么肯定了。根据指数值的不同，有时浮点数可以精确到比分还小的单位，有时甚至不能精确到元。

粗略地讲，单精度浮点数可精确到$1/2^{24}$，或1/16777216，或约百万分之六。这到底是什么意思呢？

从某种意义上讲，它意味着如果想用单精度浮点数来表示16 777 216和16 777 217，其结果是一样的。而且，在这两个数之间的任何数（如16 777 216.5）也认为是与它们一样的。所

有这3个十进制数都按32位单精度浮点数

4B800000h

来存放。当把此数分成符号位，指数和有效数位时，如下所示：

0 10010111 00000000000000000000000

也即为

$$1.00000000000000000000000_{TWO} \times 2^{24}$$

下一个表示的最大有效数是16 777 218，它的二进制浮点表示为：

$$1.00000000000000000000001_{TWO} \times 2^{24}$$

两个不同的十进制数却以相同的浮点数存放可能是也可能不是一个问题。

如果是为银行编写程序且用单精度浮点数来存储元、分等，则你可能会很苦恼地发现$262 144.00与$262 144.01是一样的。两个数字都是：

$$1.00000000000000000000000_{TWO} \times 2^{18}$$

这就是当处理元、分的时候，为什么要用定点数的原因。在处理浮点数的时候，可能还会发现其他足以使人发疯的小毛病。程序原本计算的结果是3.50却成了3.499999999999。浮点计算中这种事情经常发生，但也没有别的更好的处理方法。

如果想用浮点表示法，又不想出现单精度那样的问题，可以用双精度浮点格式。这样的数需要8个字节来存放，格式如下：

s＝1位符号	e＝11位指数	f＝52位有效数

指数偏移量为1023，即3FFh，所以，以这种格式存放的数为

$$(-1)^{s} \times 1.f \times 2^{e-1023}$$

它具有与单精度格式中所提到适用于0、无穷大和NaN等情形相同的规则。

最小的双精度浮点格式的正数或负数为

$$(1.0\underbrace{\ldots\ldots}_{52个0}0)_{TWO} \times 2^{-1022}$$

最大的数为

$$(1.1\underbrace{\ldots\ldots}_{52个1}1)_{TWO} \times 2^{1023}$$

用十进制表示，它的范围近似为2.2250738585072014×10^{-308}～1.7976931348623158×10^{308}。10的308次幂是一个非常大的数，在1后面有308个十进制零。

53位有效数（包括没有包含在内的那1位）的精度与16个十进制位表示的精度十分接近。相对于单精度浮点数来说这种表示要好多了，但它仍然意味着最终还是有一些数与另一些数是相等的。例如，140 737 488 355 328.00与140 737 488 355 328.01是相同的，这两个数按照64位双精度浮点格式存储，结果都是：

42E0000000000000h

可把它转换为：

$$(1.0\underbrace{.....0}_{52个0})_{TWO}\times 2^{47}$$

当然，开发一种格式用来在存储器中存储浮点数只是在汇编语言程序中实际使用这些数的工作中的一小部分。如果真的要研制与世隔绝的计算机，则你需要面对编写浮点数的加、减、乘、除的函数集的工作。幸运的是，这些工作可以被分解成许多小的只涉及到整数的加、减、乘、除的工作，而整数的四则运算我们已经知道如何实现了。

例如，浮点加法的关键是有效数相加，因而用的技巧是用两个数的指数部分确定有效数如何移位。假设要做以下加法：

$$(1.1101\times 2^5)+(1.0010\times 2^2)$$

需要把11101与10010相加，但不是就这样相加。指数部分的不同表明第二个数必须进行移位。实际上，需要进行11101000和10010的整数加法。最后的和是：

$$1.1111010\times 2^5$$

有时两个数的指数部分差距很大，其中一个数甚至对和没有影响。就像这种情况：把地球到太阳的距离与氢原子的半径相加。

两个浮点数的相乘是把有效数部分像整型数那样相乘并把两个整型指数相加。通常，规格化有效数部分可能会引起对新的指数调整一、二次。

浮点算术运算中另一个复杂问题牵涉到较麻烦的计算，如方根、幂、对数和三角函数。但是，所有这些工作都可以用四个基本的浮点操作：加、减、乘、除来完成。

例如，三角函数Sin可以通过下列展开式来计算，如下：

$$\frac{}{}\quad\frac{}{}\quad\frac{}{}$$

参数x必须是弧度，360度的弧度为2π。感叹号是阶乘符号，其含义是把1到该数之间的所有整数相乘，如：$5!=1\times 2\times 3\times 4\times 5$。这只是进行乘法运算，其中每一项的指数部分也是乘法。其余的是一些除法、加法和减法。唯一真正麻烦的部分是在最后的省略，它意味着要永远地计算下去。然而，实际上，如果局限在$0\sim\pi/2$的范围（从这里可以推导出所有其他的正弦函数值），并不需要进行多少展开运算。在展开大约12项以后，已经精确到了双精度数的53位。

当然，使用计算机是为了使人们更容易完成某些工作，所以，编写浮点运算程序这样的工作似乎离使用计算机的目的相差甚远。然而，这正是软件的可爱之处：一旦某人为某台机器编写了浮点运算程序，其他人都可以使用。对科学和工程应用程序来说，浮点运算非常重要，所以通常有很高的优先权。在计算机出现的早期，一旦新的类型的计算机出来，编写浮点运算程序通常是第1项软件工作。

事实上，甚至可以设计计算机机器码指令直接进行浮点运算！显然，说起来容易做起来难，但这也说明了浮点运算的重要性。如果可以用硬件来实现浮点运算——与16位微处理器的乘法和除法指令一样——则计算机中所有浮点运算工作将会完成得更快。

最早把浮点运算硬件作为选件的商用计算机是1954年的IBM 704，704把所有的数按36位来存储。对浮点数，分成27位的有效数、8位指数和1个符号位。浮点运算硬件可做加法、减法、乘法和除法，其他浮点运算功能必须用软件来实现。

桌面机的浮点运算硬件出现在1980年，当时Intel发布了8087数字数据协处理器芯片，一种集成电路芯片，今天通常称为数学协处理器或浮点运算单元（floating-point unit，FPU）。8087之所以称为协处理器是因为它不能自己单独使用，它只能与8086或8088一起使用，8086和8088是Intel的第一个16位微处理器。

8087有40个引脚，使用许多与8086和8088相同的信号。微处理器和数学协处理器通过这些信号连接起来。当CPU收到一个特殊指令——称为ESC，代表Escape——则协处理器接管系统控制权并执行下一条机器代码，即包括三角运算、指数、对数运算的68条指令中的一条。数据类型以IEEE标准为基础。那时，8087被认为是所生产的最高级的集成电路。

可以认为协处理器是一个小的自包含的计算机。在响应某个浮点运算机器码指令时（例如，计算平方根的FSQRT指令），协处理器内部执行存放在ROM中的自己的指令序列，这些内部指令称为微代码。这些指令通常是循环的，所以计算结果并不是马上可用。尽管如此，一般来说，数学协处理器至少比用软件来实现的同样例程要快10倍。

初始的IBM PC主板在8088芯片的右边有一个40管脚的插槽供8087用。遗憾的是，这个插槽是空的，需要加速浮点运算的用户必须单独购买8087并自己把它安装上。即使在安装了数学协处理器后，并不是所有的应用程序都可以运行得更快，一些应用程序——如，文字处理程序——几乎不需要浮点运算。其他如电子报表程序则要用到很多浮点计算。这些程序能够运行得更快，但并不是所有程序都是如此。

可以看到，程序员必须用协处理器机器码指令来编写特定的代码供协处理器执行。因为数学协处理器不是硬件的标准部分，因而许多程序员怕麻烦不愿意做。但是，他们还是不得不编写自己的浮点运算子程序（因为许多人并没有安装数学协处理器），所以支持8087芯片就成为一个额外的负担——一个不小的负担。最终，如果他们程序运行的机器上有数学协处理器，程序员要学会编写利用数学协处理器的应用程序；如果没有，则要编写浮点运算仿真程序。

经过几年后，Intel还发布了用于286芯片的287数学协处理器，用于386的387数学协处理器。但对于1989年发布的Intel 486DX，FPU已经做在了CPU里面，而不再是作为一个选件！遗憾的是，1991年Intel发布了一种低价格的486SX，它没有把FPU做在CPU里面，而是提供了487SX数学协处理器作为一个选件。1993年发布的Pentium芯片却再一次使做在CPU内部的FPU成为标准，也许以后永远会这样。Motorola在它的68040微处理器里集成了FPU，该微处理器于1990年发布。以前，Motorola销售68881和68882数学协处理器用来支持早先68000家族的微处理器。PowerPC芯片也把浮点运算硬件集成在内部。

尽管浮点运算硬件对专门从事汇编语言程序设计的程序员来说是一个很好的礼物，但是，与20世纪50年代早期开始的其他一些工作相比这只是微不足道的进步。我们的下一个主题是：计算机语言。