維基百科

补码

维基百科，自由的百科全书

补码（英语：**2's complement**）是一种用二进制表示有号数的方法，也是一种将数字的正负号变号的方式，常在计算机科学中使用。补码以有符号比特的二进制数定义。

“补码”的各地常用别名	
中国大陆	补码、二的补码
台湾	二补数
港澳	二补码

正数和0的补码就是该数字本身。负数的补码则是将其对应正数按位取反再加1。

补码系统的最大优点是可以在加法或减法处理中，不需因为数字的正负而使用不同的计算方式。只要一种加法电路就可以处理各种有号数加法，而且减法可以用一个数加上另一个数的补码来表示，因此只要有加法电路及补码电路即可完成各种有号数加法及减法，在电路设计上相当方便。

另外，补码系统的0就只有一个表示方式，这和反码系统不同（在反码系统中，0有二种表示方式），因此在判断数字是否为0时，只要比较一次即可。

右侧的表是一些8-bit补码系统的整数。它的可表示的范围包括-128到127，总共256（=2^8）个整数。

符号									
0	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**=**	**127**
0	**0**	**0**	**0**	**0**	**0**	**1**	**0**	**=**	**2**
0	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**=**	**1**
0	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**=**	**0**
1	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**=**	**−1**
1	**1**	**1**	**1**	**1**	**1**	**1**	**0**	**=**	**−2**
1	**0**	**0**	**0**	**0**	**0**	**0**	**1**	**=**	**−127**
1	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**=**	**−128**

目录

数字表示方式

说明

以下用4位的补码数字来说明补码系统的数字表示方式。

- 在表示正数和零时，补码数字和一般二进制一样，唯一的不同是在补码系统中，正数的最高比特恒为0，因此4位的补码正数，最大数字为0111 (7)。
- 补码数字的负数，最高比特恒为1，4位补码的数字中，最接近0的负数为1111 (-1)，以此类推，因此绝对值最大的负数是1000 (-8)。

以上的表示方式在电脑处理时格外方便，用以下的例子说明：

```
    0011 (3)
  + 1111 (-1)
--------------
   10010 (2)
```

补码	十进制
0111	7
0110	6
…	…
0010	2
0001	1
0000	0
1111	−1
1110	−2
…	…
1001	−7
1000	−8

结果10010似乎是错的，因为已经超过四个比特，不过若忽略掉（从右开始数）第5个比特，结果是0010 (2)，和我们计算的结果一样。而且若可以将二进制的1111 (-1)变号为0001 (1)，以上的式子也可以计算减法：3-1 = 2。

在n比特的补码加减法中，忽略第n+1个比特的作法在各种有号数加法下都适用（不过在判断是否溢出（overflow）时，仍然会用到第n+1个比特）。因此在补码的系统，加法电路就可以处理有负数的加法，不需另外处理减法的电路。而且，只要有电路负责数字的变号（例如将1变换为 -1），也可以用加法电路来处理减法。而数字的变号就用计算数字的补码来完成。

在一般n比特的二进制数字中，最高有效比特（MSB）第 n比特代表的数字为 2^{n-1}。不过，在n比特的补码系统中，最高有效比特（MSB）第 n比特表示符号比特，若符号比特为0，数字为正数或0，若符号比特为1，数字为负数。以下是n比特的补码系统中，几个特别的数字：

补码	实际数字	附注
0 111....111	2^{n-1}-1	当前有符号位区分的**最大正数**
...	...	
0 000....001	1	
0 000....000	0	
1 111....111	-1	
...	...	
1 000....001	- 2^{n-1}+1	当n<当前所在二补数系统内所含的最大比特数量（最大比特数量是4的整数倍，且整数倍数大于等于1），得出根据当前n的值从而解得有符号位区分的**负整数**（同时根据n的值也确定了最大比特数量的值）。
1 000....000	- 2^{n-1}	当n=当前所在二补数系统内所含的最大比特数量（最大比特数量是4的整数倍，且整数倍数大于等于1），得出根据当前n的值从而解得有符号位区分的**最小负数**（同时根据n的值也确定了最大比特数量的值）。

因此，在8位的补码系统中，可以表示的最大正数为2^{8-1}-1 = 127，可以表示最小的负数为 -2^{8-1} = -128

计算补码

在计算二进制数字的补码时，会将数字进行比特反相运算，再将结果加1，不考虑溢出比特（一般情形，溢出比特会为0），就可以得到该数字的补码。

以下考虑用有符号位8位二进制表示的数字5：

0000 0101 (5)

其最高比特为0，因为此数字为正数。若要用补码系统表示 -5，首先要将5的二进制进行反相运算〔1变为0，0变为1〕：

1111 1010

目前的数字是数字5的反码，因此需要再加1，才是补码：

1111 1011 (-5)

以上就是在补码系统中 -5的表示方式。其最高比特为1，因为此数字确实为负数。

一个负数的补码就是其对应的正数。以 -5为例，先求数字的反码：

0000 0100

再加一就是 -5的补码，也就是5。

0000 0101 (5)

简单来说，数字a（正负数皆可）的补码即为 -a。

若要计算n位数补码二进制对应的十进制，需要知道每位数对应的数字，除了最高比特外，其他比特的对应数字均和一般二进制相同，即第i位数表示数字2^{i-1}。但最高比特若为1时，其表示数字为 -2^{n-1}，因此若用此方式计算0000 0101表示的数字，其结果为：

1111 1011 (−5) = −128 + 64 + 32 + 16 + 8 + 0 + 2 + 1 = $(-2^7 + 2^6 + ...)$ = −5

特别的数字

有两个数字的补码等于本身：一个是0，另一个为该比特内可表示有符号位区分的二进制形式的最大负数（即1000...）。

0的补码计算方式（以8位为例）如下：先计算它的反码：

1111 1111

再将反码加一：

0000 0000，溢出比特二进制值 = 1（二进制）

忽略溢出，其结果为0（0是唯一计算补码过程中会出现溢出的数字。）。因此0的补码为0。而0 x (-1) = 0，因此其补码仍满足“数字a的补码为 -a”的原则。

若计算1000 0000（这是8位内可表示有符号位区分的二进制形式的最大负数**-128**）的补码：先计算它的反码：

0111 1111

再加一就是它的补码。

1000 0000

1000 0000 (-128)的补码仍为1000 0000 (-128)。但(-128) x (-1) = 128，因此其补码是以上规则的例外。

总结：由于0可以等于0的二补数**-0**，以及同样因为8位的二补数可显示的值范围为 -128 ~ 127，但-128的二补数**128**无法用在已有比特数量为8的比特数量内的可用二补数表示。【在计算其他位数内的可表示有符号位区分的二进制形式的最大负数（即1000...000）时，也会有类似情形。】

所以：**0**和**-128**的确是“数字a的补码为 -a”原则中**两个特别的数字**。

其他计算方法

方法一

Ps：这方法不好用。有符号位区分的$N=2^{8a-1}$【a为非0正整数倍的Byte数量，这个公式得出的值一律按二进制默认自带符号位为0的正数理解从而对应下面的推导公式运用】，同时还有0 这个类型的特殊的数字，主要原因是得出的值运用到下面的公式去，并根据溢出的bit值为0的原则，反之会出现不同Byte数量内的加减法得出的实际值没有任何意义。

另一种正式计算一数字（以欲求得其二补数-N为例）的补码N* 的公式如下：

$$N^* = 2^n + (-N)$$

其中N* 是 -N的补码，而n是十进制数字N用二进制表示时需要的在进位字节内的所有有效比特数量。

以八比特的“-123”求其二补数为例【只有负数才有补码概念，无论用哪种公式或简便方法（**比如方法二**)求得补码与之最传统方法其二进制正数取反加1的结果相同】：

n = 8
N = + $123_{(10)}$ = 111 $1011_{(2)}$（扣除表示正负号的第一个位，实际只有七个位可用）

-123的补码计算方式如下：

$$N^* = 2^n - N = 2^8{}_{(10)} - 111\ 1011 = 1\ 0000\ 0000 - 111\ 1011 = 1000\ 0101$$
$$-123_{(10)} = 1000\ 0101_{(2)}$$

验证

$$-123 = -128 + 5 = 1000\ 0000 + 101 = 1000\ 0101$$

方法二

以另一种较简单的方式，可以找出二进制数字的补码：

- 先由最低比特开始找。
- 若该比特为0，将补码对应比特填0，继续找下一比特（较高的比特）。
- 若找到第一个为1的比特，将补码对应比特填1。
- 将其余未转换的比特进行比特反相，将结果填入对应的补码。

以0011 1100为例（图中的^表示目前转换的数字，-表示还不确定的位数）：

```
  原数字       补码
0011 1100  ---- ---0（此比特为0）
        ^

0011 1100  ---- --00（此比特为0）
       ^

0011 1100  ---- -100（找到第1个为1的比特）
      ^

0011 1100  1100 0100（其余比特直接反相）
^
```

因此其结果为1100 0100

符号延展

将一个特定比特补码系统的数字要以较多比特表示时（例如，将一个字节的变量复制到另一个二个字节），所有增加的高比特都要填入原数字的符号比特。在一些微处理机中，有指令可以运行上述的动作。若是没有，需要自行在程序中处理。==>

十进制	4位补码	8位补码
5	0101	0000 0101
-3	1101	1111 1101

在补码系统中，当数字要向右位移几个比特时，在位移后需将符号比特再填入原位置（算术移位），保持符号比特不变。以下是二个例子：

```
    數字       0010 1010            1010 1010
向右位移一次   0001 0101            1101 0101
向右位移二次   0000 1010            1110 1010
```

而当一个数字要向左位移n个比特时，最低比特填n个0，权值最高的n个位被抛弃。以下是二个例子：

```
    數字       0010 1010            1010 1010
向左位移一次   0101 0100            0101 0100
向左位移二次   1010 1000            1010 1000
```

向右位移一次相当于除以2，利用算术移位的方式可以确保位移后的数字正负号和原数字相同，因为一数字除以2后，不会改变其正负号。 **注意**：向左位移一次相当于乘以2，虽然乘以在理论上并不会改变一个数的符号，但是在补码系统中，用以表示数的二进制码长度有限，能够表示的数的范围也是有限的：若一个数的高权值上的数位已经被占用，此时再将这个数左移若干位（乘以2^n）的话，有可能造成数位溢出（overflow），高权值上的数将会失去，对于绝对值很大数，这将造成整体表达的错误。

补码的工作原理

为什么补码能这么巧妙实现了正负数的加减运算？答案是：指定n比特字长，那么就只有2^n个可能的值，加减法运算都存在上溢出与下溢出的情况，实际上都等价于模2^n的加减法运算。这对于n比特无符号整数类型或是n比特有符号整数类型都同样适用。

例如，8位无符号整数的值的范围是0到255. 因此4+254将上溢出，结果是2，即$(4+254) \equiv 258 \equiv 2 \pmod{256}$。

例如，8位有符号整数的值的范围，如果规定为−128到127，则126+125将上溢出，结果是−5，即$(126+125) \equiv 251 \equiv -5 \pmod{256}$。

对于8位字长的有符号整数类型，以2^8即256为模，则

$$
\begin{aligned}
-128 &\equiv 128 \pmod{256} \\
-127 &\equiv 129 \pmod{256} \\
&\vdots \\
-2 &\equiv 254 \pmod{256} \\
-1 &\equiv 255 \pmod{256}
\end{aligned}
$$

所以模256下的加减法，用0, 1, 2,…, 254,255表示其值，或者用−128, −127,…, −1, 0, 1, 2,…,127是完全等价的。−128与128，−127与129，…，−2与254，−1与255可以互换而加减法的结果不变。从而，把8位（octet）的高半部分（即二进制的1000 0000到1111 1111）解释为−128到−1，同样也实现了模256的加减法，而且所需要的CPU加法运算器的电路实现与8位无符号整数并无不同。

实际上对于8比特的存储单元，把它的取值[00000000,…, 11111111]解释为[0, 255]，或者[-1, 254]，或者[-2, 253]，或者[-128, 127]，或者[-200, 55]，甚至或者[500, 755]，对于加法硬件实现并无不同。

运算

加法

补码系统数字的加法和一般加法相同，而且在运算完成后就可以看出结果的正负号，不需特别的处理。

正数与负数相加不会出现上溢错误，因为它们的和一定会小于加数。上溢错误只有在两个正数或两个负数相加时才可能发生，这时候最高有效位（正负号）会变成相反。

以15加-5为例：

```
  11111 111（进位）
   0000 1111 (15)
+ 1111 1011 (-5)
----------------
   0000 1010 (10)
```

由于加数和被加数都是8位，因此运算结果也限制在8位内。第8位相加后产生的进位不考虑（因为不存在第9比特）的1被忽略，所以其结果为10。而15 + (-5) = 10，计算结果正确。

在以上计算式中，可以由进位列的最左侧二个比特得知结果是否出现溢出。溢出就是数字的绝对值太大，以致于无法在指定的二进制比特个数来表示（在此例中，是超过8位的范围）。若进位列的最左侧二个比特同为0或同为1，表示结果正确，若是一个为0，另一个为1，表示出现溢出错误。也可以对此二个比特进行异或运算，结果为1时，表示出现溢出错误。以下以7 + 3的4位加法说明溢出错误的情形。

```
  0111（进位）
   0111 (7)
+ 0011 (3)
------------
   1010 (−6)  结果不正确!
```

在此例中，进位列的最左侧二个比特为01，因此出现溢出错误。溢出的原因是7 + 3的结果(10)超过补码系统4位所可以表示的数字范围 -8~7。

故为防止溢出错误，二补数在进行加法运算时通常将符号位进行复制后追加到最高位之前，即设二补数B的位数为WIDTH，则B′={B[WIDTH-1],B}。应注意此处B′的位数为WIDTH+1。 如上两例用此方法进行计算：

```
   11 1111 111（进位）
    0 0000 1111 (15)
   +1 1111 1011 (-5)
-------------------
(1)0 0000 1010 (10)
```

由于WIDTH+1=8+1=9，故而第十位的1同样由于溢出而被省略，结果仍为10。两负数（符号位为1）相加时同理。

```
     111（进位）
   0 0111 (7)
+  0 0011 (3)
--------------
   0 1010 (10)  结果正确!
```

由于WIDTH+1=4+1=5，故第五位的0仍为符号位，得结果正数10（十进制）。

减法

减法通常转化为加法进行运算，将被减数与减数的补码进行加法运算，即可得出差。

乘法

乘法在电脑的世界里其实就是不断的做加法。

例如:3*5=3+3+3+3+3=15

除法

除法就是相减。

例如：10/3=10-3-3-3=1mod3 而减法又可做补码相加，所以所有四则运算的基础都是由加法而来。

相关条目

- 补数
- 反码
- 二进制
- 有符号数处理
- 加法器