一个正数的原码、反码、补码是相同的，负数则不同。

负数：

原码就是原来的表示方法。

反码是除符号位（最高位）外取反。

补码=反码+1。

原码：



反码：

补码：



移码：



**原码：**

如果机器字长为n，那么一个数的原码就是用一个n位的二进制数，其中最高位为符号位：正数为0，负数为1。剩下的n-1位表示数值的绝对值。

例如： X=+101011 , [X]原= 00101011    X=-101011 , [X]原= 10101011

位数不够的用0补全。

PS：正数的原、反、补码都一样：0的原码跟反码都有两个，因为这里0被分为+0和-0。

**反码：**

知道了什么是原码，那反码就更是张飞吃豆芽——小菜一碟了。知道了原码，那么你只需要具备区分0跟1的能力就可以轻松求出反码，为什么呢？因为反码就是在原码的基础上，符号位不变其他位按位取反(就是0变1，1变0)就可以了。

例如：X=-101011 , [X]原= 10101011 ，[X]反=11010100

**补码：**

补码也非常的简单就是在反码的基础上按照正常的加法运算加1。

例如：X=-101011 , [X]原= 10101011 ，[X]反=11010100，[X]补=11010101

PS：0的补码是唯一的，如果机器字长为8那么[0]补=00000000。

**移码：**

移码最简单了，不管正负数，只要将其补码的符号位取反即可。

例如：X=-101011 , [X]原= 10101011 ，[X]反=11010100，[X]补=11010101，[X]移=01010101

**原码** 最高位为符号位，“0”表示正，“1”表示负，其余位表示数值的大小。  
**反码** 正数的反码与其原码相同；负数的反码是对其原码逐位取反，但符号位除外。  
**补码** 正数的补码与其原码相同；负数的补码是在其反码的末位加1。  
**移码** 将补码的符号位取反。

[**在8位二进制中，-128 没有原码、反码形式，那么它的补码是怎么计算出来的？还是约定的？**](https://www.zhihu.com/question/20458542)

要说清这个问题，需要颠覆你对补码的理解  
  
第一步，就像练北冥神功要先散功一样，先把你心中对原码，反码，补码的一套认识全部忘掉  
｜  
｜  
｜  
｜  
｜  
V  
  
第二步，正式开讲  
首先灌输一个新的概念叫，模。  
什么是“模”，想象日常使用的钟表，它可以显示0～12点的时间，假设现在是2点钟，请用手动拨动时针的方式将时间减4小时，你会怎么做？  
有两种方式：

1. 逆时针将时针拨4小时
2. 顺时针将时针拨8（12-4）小时

这里要讲的是第二种方式，为什么顺时针拨12-4也可以达到和正常思维的第一种方式一样的位置。  
12就是模。  
同样的，如果是十进制的两位数，80-10 和 80＋90在不考虑百位数的基础上都是70。这里的90就是100-10得来的，这种情况下100就是模。  
模就好比是一个极限，在它的范围内，两个相加等于模的数互为补数，还是举100的例子  
90和10， 55和45，68和32，互为补数。  
在模的范围内做减法，可以将“X－Y”的减法变更为“X＋Y的补数“的加法，当然前提是不考虑百位数。  
  
思考题，*上面举的例子是大数减小数，那么如果是小数减大数会怎么样呢？*  
如果是10-80，结果应该是－70，但如果按照10+（100-80），结果是30。  
而很明显－70和30不是一回事，这里也没有百位数的问题，这种情况应该怎么破？  
当初的那些先贤们想出来的办法很简单，就是把这两个数直接划上等号，正好顺便解决了负数的表达方式。再来仔细看看这两个数的关系：－70绝对值的补数就正好是30。  
所以在计算机中，**负数的表达方式就是它绝对值的补数**  
但是问题又来了，看起来这个解决方式很完美了，但别忘了，30他已经代表了正数的30了，现在又要用来代表负数的－70，谁知道它出现的时候到底是代表哪个数？  
为了解决这个问题，需要给这套规则划定一个范围，原来是0～99的正数，现在既然要用部分正数来代替负数了，那就要规定一个范围来使得一个数只代表一个含义，正好一人一半，0～49这个区间就代表正数，50～99的区间就用来代表各自补数的负值，例：98就代表－2  
｜  
｜  
｜  
V  
  
第三步，现在回到二进制的计算机世界  
8位二进制数一共可以表示2的8次方，256个数，即0～255 （别忘了0也要占一位的），他们的极限就是256，即256是8位二进制数的模 ，应该不难理解吧，同上十进制的两位数0～99的模是100。  
还是用二进制来说明清楚，8位二进制能表示的数的极限是  
1 1 1 1 1 1 1 1， 就是255，在这基础上加0 0 0 0 0 0 0 1，出现了进一位 即 1 0 0 0 0 0 0 0 0  
这个1 0 0 0 0 0 0 0 0就是8位二进制数的模，256  
  
同样按照第二步讲的逻辑，一半的数0～127，代表其正数本身，另一半的数 128～255，代表其补数的负值，即“－1～－128”的区间。   
而 “X－Y”的减法 就用 “X＋Y的补数” 的加法来表示，完美！ 唯一需要注意的事情是任何计算的输入值和输出结果值都需要严格遵守－128～127的范围，一旦溢出就会报错。  
这也就是我们在编程里强调的为什么 byte＋byte还得是byte，int＋int还得是int，数据溢出问题也是每一个程序员都需要注意的问题。  
  
这样一说是不是可以理解－128的补码是怎么来的了吧？ 他就是256-｜－128｜＝128  
二进制的128是不是就是1 0 0 0 0 0 0 0 ？  
｜  
｜  
｜  
｜  
｜  
V  
  
最终问题，那书和老师为什么要用原码，反码来讲补码 ？  
空穴来风，未必无因。  
那是因为计算机就是这样求负数的补码的，我们在键盘上敲一个负数的时候，计算机要把它用补码的形式存储下来，还记得上面我们讲的补码是怎么来的吗？  
模－绝对值，这是不是个减法公式？但计算机没有减法逻辑，我们费了那么大的劲搞了一套补码的规则就是为了用加法来替代减法，但为了实现这么套规则，却跨不过一个坎，就是把负数计算成补码仍然是需要减法逻辑的。怎么办呢，那些伟大的先贤们 （膜拜）就想出了这么个办法：  
首位不变，其余位取反后，再加一（这个原理是什么？请参见阮一峰的[《关于2的补码》](http://www.ruanyifeng.com/blog/2009/08/twos_complement.html)）  
｜  
｜  
｜  
｜  
｜  
V  
下面是吐槽  
不知道是哪个书呆子教书，照搬了机器的逻辑，把取反加一的方法当做补码的计算逻辑就这么教下来了。搞笑的是，还保留了补码这个名字，照理说这种教法应该叫 取反加一码 更合理，你还补什么啊？   
不仅如此，还搞出了个首位符号位的说法，弄出了个正0负0，还用负0来充当－128，真是不把人弄疯不罢休啊！！

众所周知，计算机内部的所有数都是以二进制的形式存在的。而二进制在计算机里又有多种编码方式——原码、反码、补码等。而在这些编码方式里面用得最多的不是最简单、最直接的原码而是补码。这是为什么呢？想搞懂这个问题首先得明白什么是原码、反码以及补码，如果你对他们还不太了解，那就先看看我另一篇博客——[原码、反码、补码其实很简单](http://blog.csdn.net/liushuijinger/article/details/7429197)。如果你对他们已经很熟悉，那么我们继续往下看。  
  
A、B、C三种相似的东西，选C而不选A和B，那么C肯定具有其他两者所没有的优势。那么补码究竟有什么优势让他备受青睐呢？下面我们具体的分析一下：  
  
**原码：**  
原码的特点就是编码简单直观，与真值转换非常方便。既然原码这么好，那为什么不选他而选补码呢？接下来就是不选他的关键所在，虽然原码非常的简单直观，但是当用原码表示0的时候就会出问题。0用原码表示分为+0和-0，当机器字长为8时，

**[+0]原=00000000，[-0]原=10000000。**

这就有问题了，同一个数却有两种表示，产生了二义性，从而给机器判断0带来了麻烦；二是用原码运算时，符号位需要单独处理，而且运算规则很复杂。例如加法运算，若两个数异号，则先要让绝对值大的数减去绝对值小的数，然后把绝对值大的数的符号付给结果。还有就是，借位操作如果用计算机硬件实现起来是很困难的。正是因为原码有这些不足之处，才促使人们研究其他的编码方法。

**反码：**

反码很少会被用到，他主要的用途就是作为原码与补码的一个桥梁。他和原码一样对0有两种表示方法，

**[+0]反=00000000，[-0]反=11111111。**

不采用反码的原因跟原码差不多，就不赘述了。

**补码：**

说到补码，就不得不引人另一个概念——模数。模数从屋里意义上讲是某种计量器的容量。这里我们经常举的一个例子就是钟表，其模数为12，即每到12就重新从0开始，数学上叫取模或求余(mod)，java、C#和C++里用%表示求余操作。例如：

**14%12=2**

如果此时的正确时间为6点，而你的手表指向的是8点，如何把表调准呢？有两种方法：一把表逆时针拨两个小时；二是把表顺时针拨10个小时，即

**8-2=6**

**(8+10)%12=6**

也就是说在此模数系统里面有

**8-2=8+10**

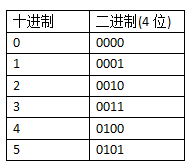
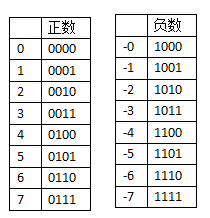
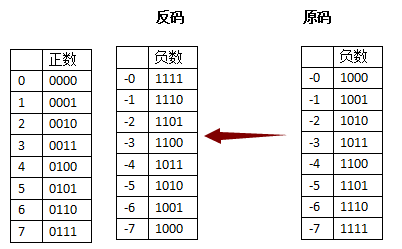
这是因为2跟10对模数12互为补数。因此有一下结论：在模数系统中，A-B或A+(-B)等价于A+[B补]，即

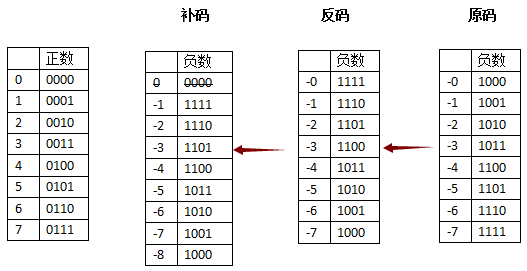
**8-2/8+(-2)=8+10**

我们把10叫做-2在模12下的补码。这样用补码来表示负数就可以将加减法统一成加法来运算，简化了运算的复杂程度。

采用补码进行运算有两个好处，一个就是刚才所说的统一加减法；二就是可以让符号位作为数值直接参加运算，而最后仍然可以得到正确的结果符号，符号位无需再单独处理。

**原码、反码、补码的产生、应用以及优缺点有哪些？**

我尝试硬生生的把它们串起来哈   
  
数字在自然界中抽象出来的时候，一棵树，两只猪，是没有正数和负数的概念的  
  
计算机保存最原始的数字，也是没有正和负的数字，叫没符号数字  
  
如果我们在内存分配4位（bit）去存放无符号数字，是下面这样子的  
  
  
后来在生活中为了表示“欠别人钱”这个概念，就从无符号数中，划分出了“正数”和“负数”  
  
正如上帝一挥手，从混沌中划分了“白天”与“黑夜”  
  
为了表示正与负，**人们发明了"原码"，把生活应该有的正负概念，原原本本的表示出来**  
  
把左边第一位腾出位置，存放符号，正用0来表示，负用1来表示  
  
但使用“原码”储存的方式，方便了看的人类，却苦了计算机  
https://pic1.zhimg.com/7cec066778fbb42aa57598d64336f8b8_b.png  
我们希望 （+1）和（-1）相加是0，但计算机只能算出0001+1001=1010 (-2)  
  
这不是我们想要的结果 (╯' - ')╯︵ ┻━┻  
  
另外一个问题，这里有一个（+0）和（-0）  
  
**为了解决“正负相加等于0”的问题，在“原码”的基础上，人们发明了“反码”**  
  
“反码”表示方式是用来处理负数的，符号位置不变，其余位置相反  
  
  
当“原码”变成“反码”时，完美的解决了“正负相加等于0”的问题  
  
过去的（+1）和（-1）相加，变成了0001+1101=1111，刚好反码表示方式中，1111象征-0  
  
人们总是进益求精，历史遗留下来的问题—— 有两个零存在，+0 和 -0  
  
**我们希望只有一个0，所以发明了"补码"**，同样是针对"负数"做处理的  
  
"补码"的意思是，从原来"反码"的基础上，补充一个新的代码，（+1）  
  
我们的目标是，没有蛀牙（-0）



有得必有失，在补一位1的时候，要丢掉最高位  
  
我们要处理"反码"中的"-0",当1111再补上一个1之后，变成了10000，丢掉最高位就是0000，刚好和左边正数的0，完美融合掉了  
  
这样就解决了+0和-0同时存在的问题  
  
另外"正负数相加等于0"的问题，同样得到满足  
  
举例，3和（-3）相加，0011 + 1101 =10000，丢掉最高位，就是0000（0）  
  
同样有失必有得，我们失去了(-0) , 收获了（-8）  
  
以上就是"补码"的存在方式  
  
**结论：保存正负数，不断改进方案后，选择了最好的"补码"方案**。

**关于2的补码**

问一个基本的问题。

**负数在计算机中如何表示？**

举例来说，+8在计算机中表示为二进制的1000，那么-8怎么表示呢？

很容易想到，可以将一个二进制位（bit）专门规定为符号位，它等于0时就表示正数，等于1时就表示负数。比如，在8位机中，规定每个字节的最高位为符号位。那么，+8就是00001000，而-8则是10001000。

但是，随便找一本《计算机原理》，都会告诉你，实际上，计算机内部采用2的补码（Two's Complement）表示负数。

**什么是2的补码？**

它是一种数值的转换方法，要分二步完成：

第一步，每一个二进制位都取相反值，0变成1，1变成0。比如，00001000的相反值就是11110111。

第二步，将上一步得到的值加1。11110111就变成11111000。

所以，00001000的2的补码就是11111000。也就是说，-8在计算机（8位机）中就是用11111000表示。

不知道你怎么看，反正我觉得很奇怪，为什么要采用这么麻烦的方式表示负数，更直觉的方式难道不好吗？

昨天，我在一本书里又看到了这个问题，然后就花了一点时间到网上找资料，现在总算彻底搞明白了。

**2的补码的好处**

首先，要明确一点。计算机内部用什么方式表示负数，其实是无所谓的。只要能够保持一一对应的关系，就可以用任意方式表示负数。所以，既然可以任意选择，那么理应选择一种最方便的方式。

2的补码就是最方便的方式。它的便利体现在，所有的加法运算可以使用同一种电路完成。

还是以-8作为例子。

假定有两种表示方法。一种是直觉表示法，即10001000；另一种是2的补码表示法，即11111000。请问哪一种表示法在加法运算中更方便？

随便写一个计算式，16 + (-8) = ?

16的二进制表示是 00010000，所以用直觉表示法，加法就要写成：

　０００１００００  
＋１０００１０００  
－－－－－－－－－  
　１００１１０００

可以看到，如果按照正常的加法规则，就会得到10011000的结果，转成十进制就是-24。显然，这是错误的答案。也就是说，在这种情况下，正常的加法规则不适用于正数与负数的加法，因此必须制定两套运算规则，一套用于正数加正数，还有一套用于正数加负数。从电路上说，就是必须为加法运算做两种电路。

现在，再来看2的补码表示法。

　０００１００００  
＋１１１１１０００  
－－－－－－－－－  
１００００１０００

可以看到，按照正常的加法规则，得到的结果是100001000。注意，这是一个9位的二进制数。我们已经假定这是一台8位机，因此最高的第9位是一个溢出位，会被自动舍去。所以，结果就变成了00001000，转成十进制正好是8，也就是16 + (-8) 的正确答案。这说明了，2的补码表示法可以将加法运算规则，扩展到整个整数集，从而用一套电路就可以实现全部整数的加法。

**2的补码的本质**

在回答2的补码为什么能正确实现加法运算之前，我们先看看它的本质，也就是那两个步骤的转换方法是怎么来的。

要将正数转成对应的负数，其实只要用0减去这个数就可以了。比如，-8其实就是0-8。

已知8的二进制是00001000，-8就可以用下面的式子求出：

　００００００００  
－００００１０００  
－－－－－－－－－

因为00000000（被减数）小于0000100（减数），所以不够减。请回忆一下小学算术，如果被减数的某一位小于减数，我们怎么办？很简单，问上一位借1就可以了。

所以，0000000也问上一位借了1，也就是说，被减数其实是100000000，算式也就改写成：

１００００００００  
－００００１０００  
－－－－－－－－－  
　１１１１１０００

进一步观察，可以发现100000000 = 11111111 + 1，所以上面的式子可以拆成两个：

　１１１１１１１１  
－００００１０００  
－－－－－－－－－  
　１１１１０１１１  
＋０００００００１  
－－－－－－－－－  
　１１１１１０００

2的补码的两个转换步骤就是这么来的。

**为什么正数加法适用于2的补码？**

实际上，我们要证明的是，X-Y或X+(-Y)可以用X加上Y的2的补码完成。

Y的2的补码等于(11111111-Y)+1。所以，X加上Y的2的补码，就等于：

X + (11111111-Y) + 1

我们假定这个算式的结果等于Z，即 Z = X + (11111111-Y) + 1

接下来，分成两种情况讨论。

第一种情况，如果X小于Y，那么Z是一个负数。这时，我们就对Z采用2的补码的逆运算，求出它对应的正数绝对值，再在前面加上负号就行了。所以，

Z = -[11111111-(Z-1)] = -[11111111-(X + (11111111-Y) + 1-1)] = X - Y

第二种情况，如果X大于Y，这意味着Z肯定大于11111111，但是我们规定了这是8位机，最高的第9位是溢出位，必须被舍去，这相当于减去100000000。所以，

Z = Z - 100000000 = X + (11111111-Y) + 1 - 100000000 = X - Y

这就证明了，在正常的加法规则下，可以利用2的补码得到正数与负数相加的正确结果。换言之，计算机只要部署加法电路和补码电路，就可以完成所有整数的加法。

**负数的补码的一种证明方法**

任意一个二进制数真值在计算机中用机器数来表示，最基本的机器数有三种，即原码、补码和反码。[1]在计算机的运算过程中，根据不同的运算运用不同的机器数，可以简化计算机运算过程并很快得到运算结果。如乘法和除法用原码参与运算比较方便，符号位通过异或运算即得结果的符号位；而加法与减法运算用补码参与运算比较方便，它使符号位不用单独处理，可以一起参与运算。在计算机中，三种机器数能方便、容易地进行转换，这给使用带来极大方便。其转换要点有三：

　　（1）将一个数的二进制真值的最高位置上符号位（正数用0，负数用1表示），就是这个数的原码。

　　（2）对于一个正数，其原码、补码、反码在计算机中的表示代码相同，其代码不用改变。

　　（3）对于一个负数，将原码的符号位（即二进制数的最高位，也即原码最左边的那一位）保持不变，数值位逐位取反（变为反码），末位（即最右边的那一位）加1，即得补码。也就是说，一个负数的补码等于该负数的反码与末位加1之和。[1]

　　本文要讨论的问题即第三个要点。这一要点在许多计算机原理的教材中使用，但很少见到这一命题的完整推导或证明过程，这给学生在学习本章节内容时只能被动地接受命题，不了解例题的理论依据。因此，教师在教学中适当地从理论的高度论述原码、补码和反码之间的转换关系很有必要。本文从补码、反码的定义出发，给出这一命题的简单证明，与同行分享与讨论。

**一、对于定点负整数**

　　(一)、补码的定义[2]如下：

　　[X]补=2n＋XMOD(2n)（1）

　　从定义知，一个n位二进制定点负整数，求其补码时，用2n为模数加上该负数即可。

　　例如：已知n=8，X=－1010101，则

　　[X]补=2n＋X=28＋(－1010101)

=100000000－1010101

=10101011

　　(说明：28的二进制为100000000，简记：28的二进制为1个1后有8个0)

　　（二）、反码的定义[2]如下：

　　[X]反=(2n－1)＋X[MOD(2n－1)]……（2）

　　从定义知，一个n位定点负整数，求其反码时，用2n－1为模数加上该负数即可。

　　例如：已知n=8,X=－1010101，则

　　[X]反=(2n－1)＋X=(28－1)＋(－1010101)

　　=11111111－1010101=10101010。

　　有了以上定点负整数补码，反码的定义，我们就很容易地证明，对于任一定点负整数，其补码等于其反码末位加1。

　　由（2）式，[X]反=(2n－1)＋X，得：

　　X=[X]反－2n＋1

　　将其代入（1）式，有

　　[X]补=2n＋X

　　=2n＋[X]反－2n＋1

　　=[X]反＋1（A）

　　根据以上推导，命题对于定点负整数是成立的。

**二、对于定点负小数**

　　（一）、补码的定义[2]如下：

　　[X]补=2＋XMOD(2)（3）

　　从定义知，一个n位二进制定点负小数，求其补码时，用2为模数加上该负数即可。

　　例如：已知n=8，X=－0.1010101，则

　　[X]补=2＋X=2＋(－0.1010101)

=10－0.1010101

=10.0000000－0.10101011

=1.0101011

　　(说明：2的二进制为10，写成小数即10.000000[3])

　　（二）、反码的定义[2]如下：

　　[X]反=2－2－(n－1)＋X[MOD(2－2－(n－1)]（4）

　　从定义知，一个n位定点负小数，求其反码时，用2－2－(n－1)）为模数加上该负数即可。

　　例如：已知n=8,X=－0.1010101，则

　　[X]反=2－2－(n－1)＋X

=2－2－7＋(－0.1010101)

=10－0.0000001－0.1010101

=1.1111111－0.1010101

=1.0101010。

　　(说明：2－7的二进制为0.0000001，简记为：2－7的二进制为小数点后第7位为1，其余位为0。（小数点后第7位即为8位二进制负小数的最末最右的一位)[3]

　　同理，有了以上定点负小数补码，反码的定义，我们也能很容易地证明，对于任一定点负小数[4]，其补码等于其反码末位加1。

　　由（4）式，[X]反=2－2－(n－1)＋X，得：

　　X=[X]反－2＋2－(n－1)

　　将其代入（3）式，有

　　[X]补=2＋X

　　=2＋[X]反－2＋2－(n－1)

　　=[X]反＋2－(n－1)（B）

　　上式中，2－(n－1)的二进制即为小数点后第n－1位为1，其余位为0，小数点后第n－1位为1，即n位定点小数最右最末的一位为1[4]。因此上式：[X]补=[X]反＋2－(n－1)，也就是X的补码等于其反码与末位加1之和）

　　根据以上推导，命题对于定点负小数同样是成立的。

综合（A）式和（B）式，可得我们要推导证明的命题成立。

首先要明确，定点小数、定点整数均可用这三种编码来表示，在学习这几种编码时有下述两个技巧：

**技巧一**：

     不管是定点小数、定点整数，编码总位数相同的情况下，补码的表数个数总比原码、反码多一个，原因在于，真值0对应的原码、反码有两个编码(对应负零和正零)，而真值0的补码只有一个。这就造成了同等条件下，补码的表数范围跟其他两种不一样，例如，当总编码位数(包含符号位)为8时，补码的表数范围为：-27≤X≤27-1，而原码、反码的表数范围为：-27-1≤X≤27-1。因此，此时，27=128有补码，却没有原码、反码。作为一种特例，真值“-1”被纳入定点小数补码的表数范围中，即定点小数补码的表数范围为：-1≤X＜1，而定点小数原码、反码的表数范围为：：-1＜X＜1(不包含-1)。

**技巧二**：

     不管是定点小数、定点整数，在求各种编码时，都可遵循以下原则：正数的原码、反码、补码的符号位均为0，原码、反码、补码数值位均为数值本身；负数的原码、反码、补码的符号位均为1，原码的数值位为数值本身，反码的数值位为数值本身（即原码数值位）各位取反，而补码的数值位是在反码数值位的基础加1，若数值最高位有进位则丢弃（不向符号位进位）。

例如，当编码总位数为8时有：

+127的原码、反码、补码都为：0 1111111。

-127的原码、反码、补码依次为：1 1111111、1 0000000、1 0000001。

+0、-0的原码分别为：0 0000000、1 0000000，均对应真值0。

由于“编码总位数为8”的限制，真值-128无法用原码、反码来表示，似乎不能用上述规则来求解补码，但实际上是可行的——只要不管它的最高位即可，操作办法如下：

将128化为二进制为：1 0000000，最高位为1，可以只对舍去最高位后剩余的7位进行处理即可，首先取反得：1111111，加1得：1 0000000，最高位有进位需丢弃，即得：0000000，加上符号位就得补码：1 0000000。

又如，当编码总位数为4时，真值X=+0.101的原码、反码、补码均为：0 101。

真值X=-0.101的原码、反码、补码依次为：1 101、1 010、1 011。

同理，特例，-1的补码为：1 000。

在定点小数中，小数点隐含在第一位编码和第二位编码之间。

以我们原有的数学经验，在10进制中：1 表示正1，而加上负号：-1 表示和1相对的负值。

那么，我们会很容易认为在2进制中（1个字节）： 0000 0001 表示正1，则高位为1后：1000 0001应该表示-1。

然而，事实上计算机中的规定有些相反，请看下表：

|  |  |
| --- | --- |
| 二进制值（1字节） | 十进制值 |
| **1**000 0000红色的1代表负数蓝色的是补码(补码=反码+1) | -128 |
| **1**000 0001蓝色部分代表多大的值？：将补码还原为原码 | -127想化成负数？：先**减去1**再**按位取反** |
| **1**000 0010还原方法：补码-1再取反 | -126 |
| **1**000 0011 | -125 |
| ... | ... |
| 1111 1110 | -2 |
| **1**111 1111 | -1 |

首先我们看到，从-1到-128，其二进制的最高位都是1（表中标为红色），正如我们前面的学。

然后我们有些奇怪地发现，1000 0000 并没有拿来表示 -0；而1000 0001也不是拿来直观地表示-1。事实上，-1 用1111 1111来表示。

怎么理解这个问题呢？**先得问一句是-1大还是-128大**？

当然是 -1 大。-1是最大的负整数。以此对应，计算机中无论是字符类型，或者是整数类型，也无论这个整数是几个字节。它都用全1来表示 -1。比如一个字节的数值中：1111 1111表示-1，那么，1111 1111 - 1 是什么呢？和现实中的计算结果完全一致。1111 1111 - 1 = 1111 1110，而1111 1110就是-2。这样一直减下去，当减到只剩最高位用于表示符号的1以外，其它低位全为0时，就是最小的负值了，在一字节中，最小的负值是1000 0000，也就是-128。

--------小米批注：就是这部分蓝色的文字，让我终于能记清楚-1的编码方式了，汗＝。＝