链滴

计算机中带符号的整数为何采用二进制的补码进行存储?

作者：lxlcs201

原文链接：https://ld246.com/article/1533730284879

来源网站：链滴

我们都知道在计算机内部数据的存储和运算都采用二进制，是因为计算机是由很多晶体管组成的而晶体管只有 2 种状态，恰好可以用二进制的 0 和 1 表示，并且采用二进制可以使得计算机内部的算规则简单，稳定性高。在计算机中存在实数和整数，而整数又分为无符号整数和有符号整数，无符的整数表示很简单，直接采用其二进制形式表示即可，而对于有符号数的表示却成了问题，如何表示负？如何去处理正负号？下面来具体说下其中的原因，在这之前先了解一下原码、反码和补码这几个念。
1.原码、反码和补码的概念
　　在了解原码、反码和补码之前先说一下有符号数和无符号数。用过 C 语言的都知道在 C 语中用 signed 和 unsigned 来标识一个数是否是有符号还是无符号类型的。对于一个 8bit 的二进制来，若当做无符号数处理，其能表示的整型值范围是 0~~255，但是这样表示数据就有个局限性，果数据是负的该如何表示？因此就引入了有符号类型的概念，对于有符号类型，规定取最高位为符号，若最高位为 0，则为正数，否则为负数，这样一来对于 8 位二进制，示数值的就只有 7 位了，能够示的非负数值范围变为 0~~127，负值范围为-127~~-1，相当于可以理解为将无符号类型能表示的 128~~255 拿来去表示-127~-1 了。事实上，在计算机内部存储中，计算机自己是无法区分无符号还是有符号类型的，对于 255 和-1，在计算机内部存储的都是 11111111。换个角度来说如果事先知道内存中存储了这样一个 8 位二进制 11111111，但是谁也不能肯定它具体表示什么数值是-1 还是 255？这个是需要靠程序员自己去指定的，如果指定为无符号类型，则编译器则通过相应令将其转换为数值 255。事实上对于-x 的二进制补码表示形式和(256-x)（256-x 当做无符号类型处）的二进制表示形式相同，从这里可以略微了解了补码的含义了。在教材中对于原码、反码以及补码般是这么定义的：
　　对于正数原码、反码以及补码是其本身。负数的原码是其本身，反码是对原码除符号位之外各位取反，补码则是反码加 1。
　　因为（-x）的二进制补码形式和 256-x 的二进制表示形式相同，而 255-x 相当于对 x 的每位取反，那么 256-x 就是 255-x 后加 1。
　　注意：1）原码、反码、补码的概念是针对有符号类型而言的。
　　2）实数始终是有符号类型的（实数并不是采用补码形式存储的，具体可参考《浅谈 C/C++ 的浮点数在内存中的存储方式》一文），型数据包括无符号和有符号类型的。
2.采用补码表示带符号的整数的原因
　　对于有符号类型的整数，有原码、反码和补码三种形式，最后选择了补码来表示，具体来说下面几点原因。
　　1）能够统一 +0 和-0 的表示
　　采用原码表示，+0 的二进制表示形式为 0 000 0000，而-0 的二进制表示形式为 1 000 000；
　　采用反码表示，+0 的二进制表示形式为 0 000 0000，而-0 的二进制表示形式为 1 111 111；
　　采用补码表示，+0 的二进制表示形式为 0 000 0000，而-0 的二进制表示形式为 1 111 111 +1=1 0000 0000，因为计算机会进行截断，只取低 8 位，所以-0 的补码表示形式为 0000 0000。
　　从上面可以看出只有用补码表示，+0 和-0 的表示形式才一致。正因为如此，所以补码的表范围比原码和反码表示的范围都要大，用补码能够表示的范围为-128~~127，0~~127 分别用00000000~~01111111 来表示，而-127~~-1 则用 10000001~11111111 来表示，多出的0000000 则用来表示-128。因此对于任何一个 n 位的二进制，假若表示带符号的整数，其表示范围为2^(n-1)~2^(n-1)-1，且有 MAX+1=MIN。看下面一段代码：
char ch=127;
ch++;
　　ch 的值是多少？它的值是-128，读者可以上机验证一下。
　　假如不采用补码来表示，那么计算机中需要对 +0 和-0 区别对待，显然这个对于设计来说要加难度，而且不符合运算规则。
　　2）对于有符号整数的运算能够把符号位同数值位为一起处理
　　由于将最高位作为符号位处理，不具有实际的数值意义，那么如何在进行运算时处理这个符位？如果单独把符号位进行处理，显然又会增加电子器件的设计难度和 CPU 指令设计的难度，但是

用补码能够很好地解决这个问题。下面举例说明：
　　比如-2+3=1
　　如果采用原码表示（把符号位同数值位一起处理）：
　　1 000 0010+0 000 0011=1 000 0101=(-5)原，显然这个结果是错误的。
　　如果采用反码表示
　　1 111 1101+0 000 0011=1 0000 0000=0 0000000=(+0)反，显然这个结果也是错误的。
　　如果采用补码表示
　　1 111 1110+0 000 0011=1 0000 0001=0000 0001=(1)补，结果是正确的。
　　从上面可以看出，当把符号位同数值位一起进行处理时，只有补码的运算才是正确的。如果把符号位和数值位一起处理，会给 CPU 指令的设计带来很大的困难，如果把符号位单独考虑的话，CU 指令还要特意对最高位进行判断，这个对于计算机的最底层实现来说是很困难的。
　　3）能够简化运算规则
　　对于-2+3=1 这个例子来说，可以看作是 3-2=1，也即[3]+[-2]=1，从上面的运算过程可知用补码运算相当于是
　　[3]补 +[-2]补=[1]补，也即可以把减法运算转换为加法运算。这样一来的好处是在设计电子件时，只需要设计加法器即可，不需要单独再设计减法器。
　　总的来说，采用补码主要有以上几点好处，从而使得计算机从硬件设计上更加简单以及简化PU 指令的设计。
测试代码
#include
int main(void)
{ char ch=-1; char *p=(char *)&ch;
unsigned char uch=*p;
printf("%d\n",uch);　//输出结果为 255
return 0;
}
Why int32 has max value 2^31 -1
<pre><code class="highlight-chroma">Int32.MaxValue = 2^31 - 1 = 01111111111111111111111111111111
                1
 00000000000000000000000000000001
                0
 00000000000000000000000000000000
               -1
 11111111111111111111111111111111
Int32.MinValue = 2^31   = 10000000000000000000000000000000
</code></pre>