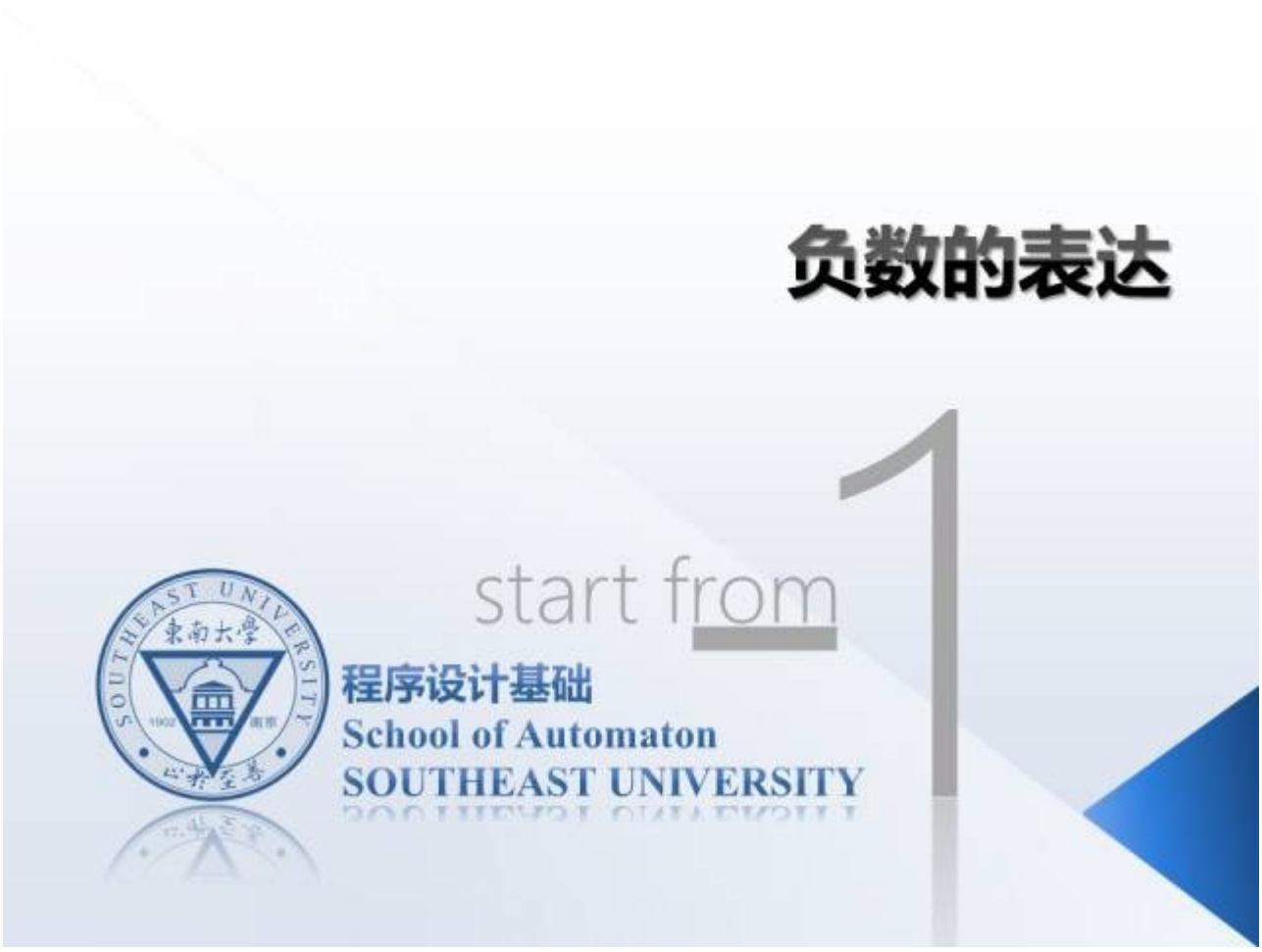
《程序设计基础》第三讲 从-1开始（补码）

原创 2016-09-26 李骏扬 骏扬工作室

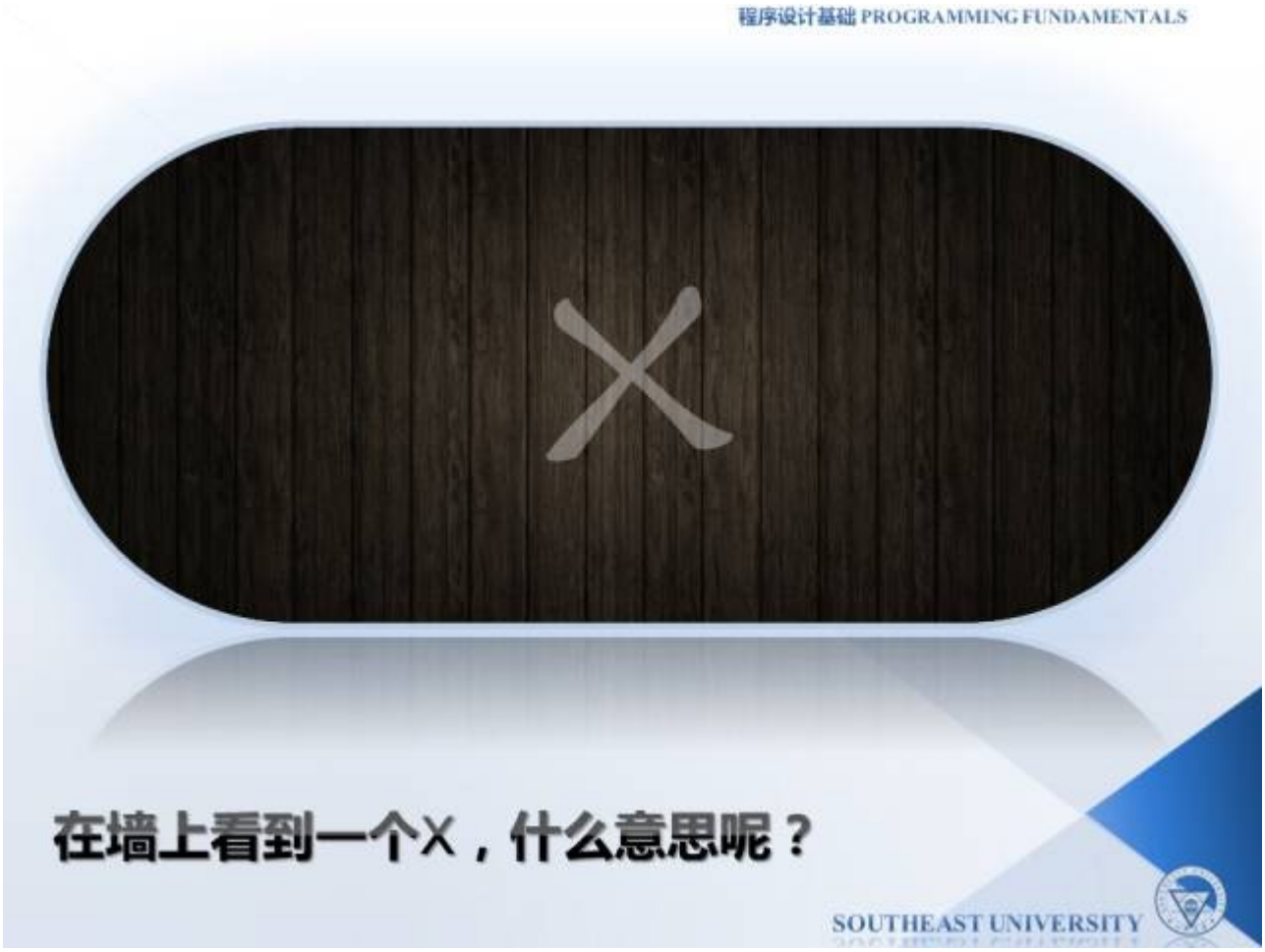


今天，我们来介绍计算机是如何表达负数的。其实我们日常表达负数的方式非常的直观，以0为中心，正负数左右对称，正负数字只是相差一个符号。这种方式方便书写，方便认知，但是正负数据的数值计算却略微有点小麻烦，比如同样是加法：2+3和-2+3，本质上一个做的是加法，一个做的是减法，所以这样的正负数表达方式容易识别，却不太利于计算。但是计算机当中，并不一定要以人类习惯的方式来表达数据，只要有利于高速计算，表达方式不太直观也是可以的。实际上计算机中对负数的表达就是这样的：不直观，但方便计算。

为了能够了解计算机中负数的表达方式，我们首先必须理解下面一个问题：**数据就在那里，不同的理解将会产生不同的结果。**这个问题对于我们理解负数的表达，甚至于后面很多对数据和内存的理解，都**至关重要**。

3.A 对同一个数据的不同理解

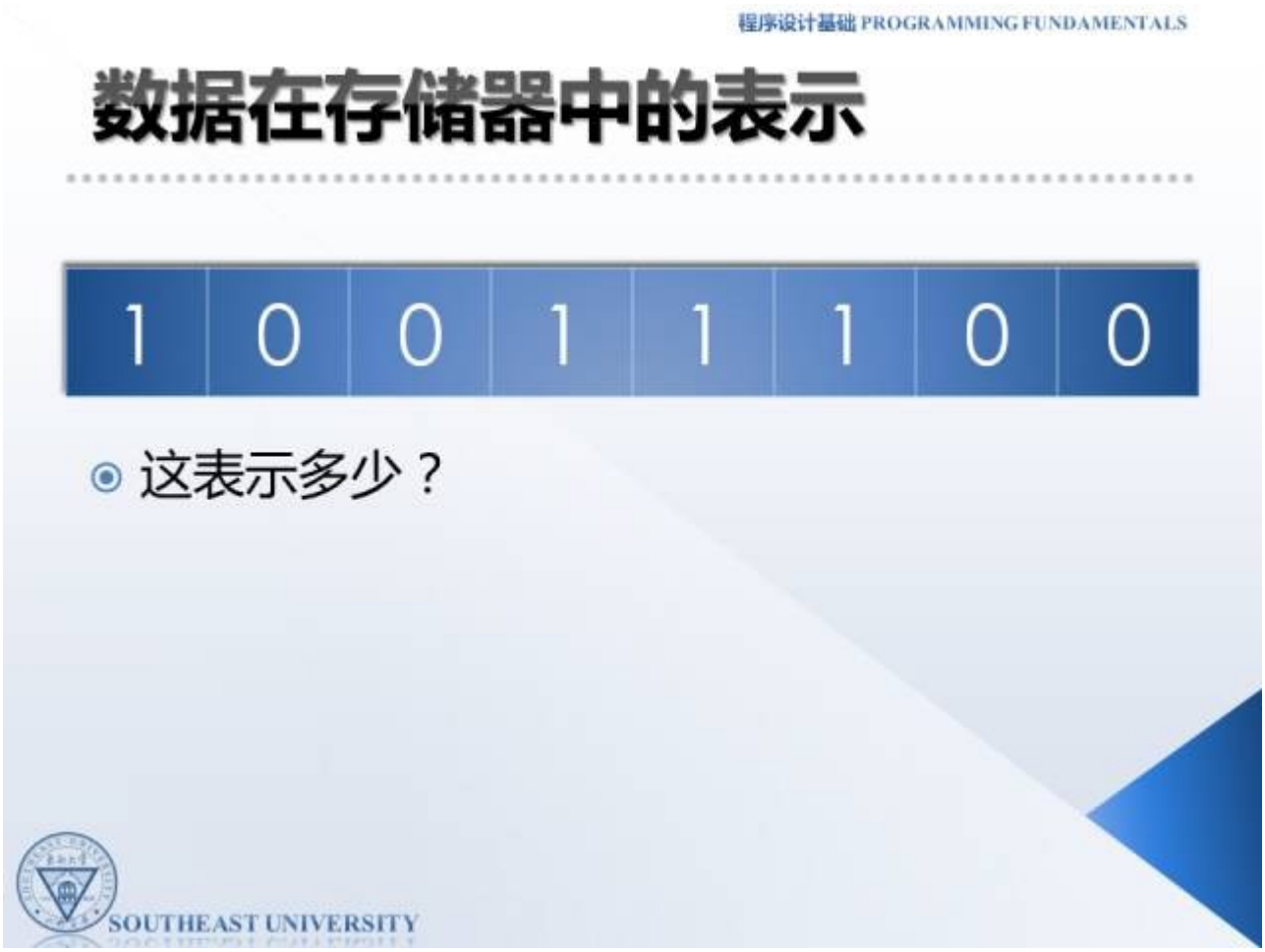
今天小明（朋友圈的小明同学请回避，嘿嘿）回到家，看到墙上写了一个X，咦？这是啥？

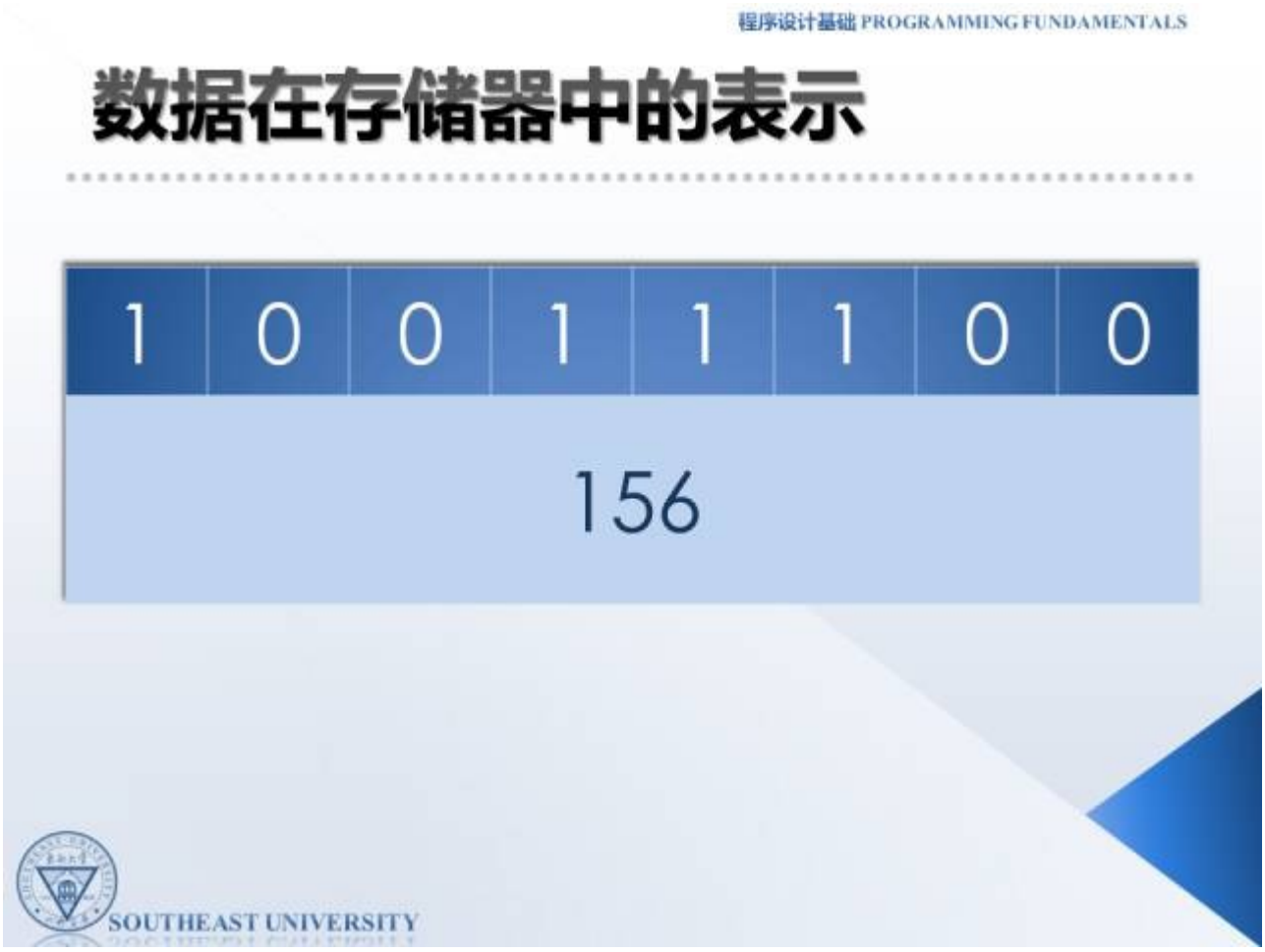


旁边一个小朋友说：小明叔叔，这个是乘法，今天老师刚教的。而另一个略微大一点的小朋友说：小明叔叔，这个是未知数x，今天我们老师也刚教的。旁边路过一个工程队的，啧啧道：没搞错吧，这房子明天要拆了？旁边的邻居则告诫小明：小伙子当心点啊，今晚要遭贼啦~~

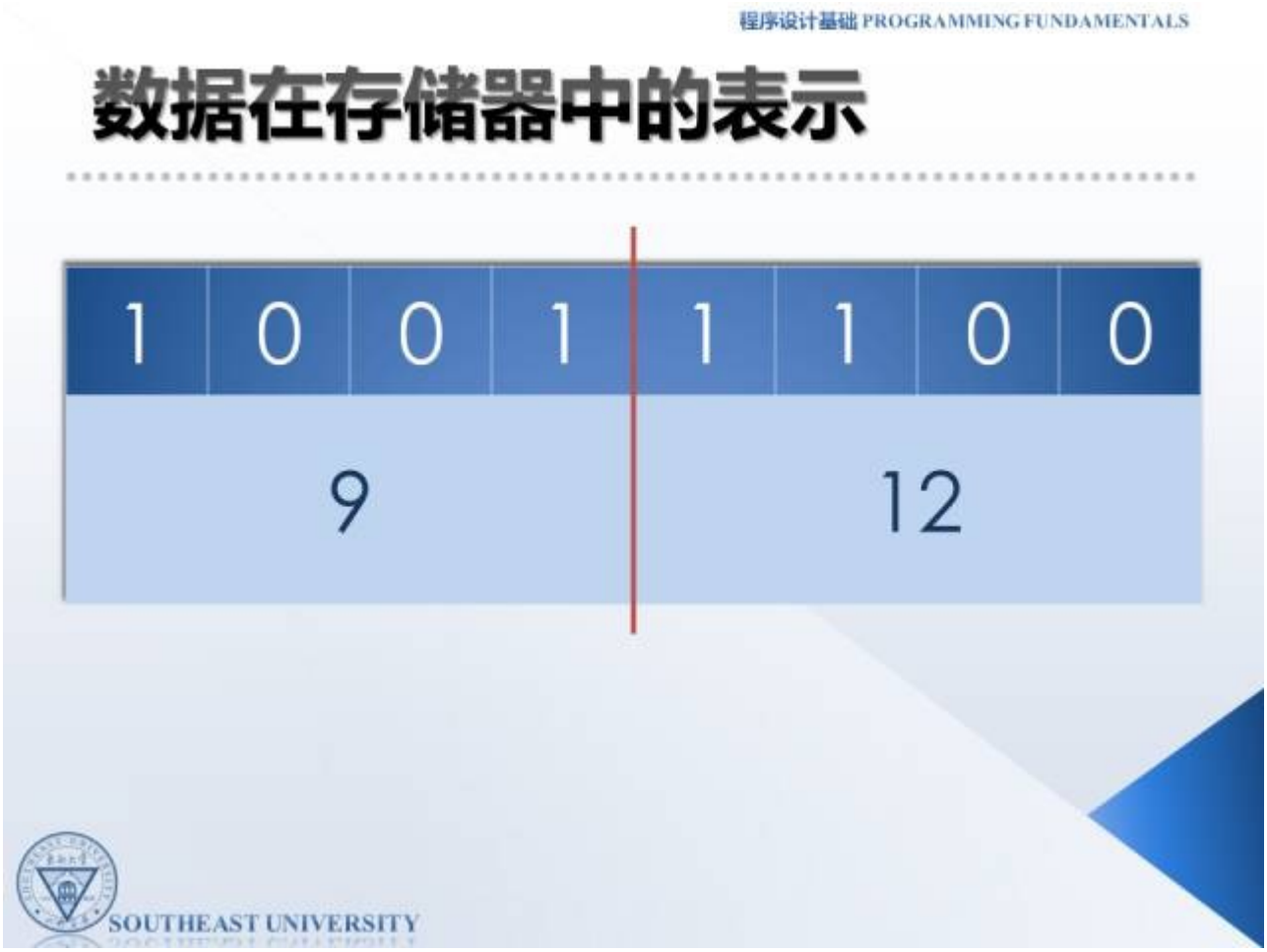
我们不去关心小明家今晚是怎么过的，我们关心的是这个故事背后的问题：同一个符号，不同的人解读，则会有不同的结果。

同样在计算机存储器中的一个数据，比如说吧，一个字节，里面存储了8个比特位10011100，这8个比特位表示什么呢？





最有可能的，这个字节是一个完整的整数，也就是156。



当然，也有可能是这样的：某个程序需要存储一堆小数字（0~15之间），为了节约存储空间，把一个字节拆成两部分用，左边4个比特表示一个数字，右边4个比特表示另一个数字，于是这8个比特位其实表达了两个数字：9和12。

当然，也可能是这样的：这是一个照明控制程序，这8个比特位对应于8盏灯的亮和灭，所以不能把这个数字看成是一个完整的数字，而应该是8个独立的数字，每个数字只有两种可能：0和1，对应于灯的灭和亮。

因此，数据就在那里，不同的理解将会得到完全不同的结果。计算机中的数据谁去理解呢？你写的程序去理解！

通常，存储在存储器中的数据，程序对其操作有两种方式：读取数据，以及写入数据。如果写入数据的时候，这个数据是用一种方式来理解的，而读取的时候却又用另一种方式去理解，这时候读取的结果就会产生偏差。这种对数据的理解方式，我们又叫做“编码”，同一个数据可以用不同的编码去理解。采用相同的编码读写数据，不会有问题，读写数据采用不同的编码方式，读取数据的结果就会出错。

3.B 从-1的表达开始

话又说回来，计算机当中到底是如何表达负数的呢？我们知道：0 - 1 = -1，好吧，那就让计算机做一个减法，以4位二进制表达为例：0000 - 0001等于多少呢？

这是个问题，做减法，又不够减的，咋办呢？

老师，要是能借一位就好了！那就借呗！老师，借不到怎么办？那就强行借一位！好吧，就当能借到吧——以后了解了计算机到底怎么做计算的，你就会发现，根本不用借也照算不误——这里就当能借到吧，于是乎，我们就得到了下面一张PPT的结果：

好的，现在老师要郑重宣布，在大多数计算机系统里面（以4位为例）：**1111就代表了-1！**

R U kidding? 老师，1111明明是表示15的好不好！

如果你这么理解，就要想想刚才我们说过的问题了：数据就在那里，表示什么却可以各有各的说法。1111，如果你把它理解成15，没问题，可是要用1111来表示-1，原则上也没有什么不可以的。可是，可是？你让15怎么办呢？

呵呵，这个就更容易解释了：4位二进制数组，一共只用16种组合：0000~1111，如果要将1111表达为-1，那么就没有15的位置了。一共就16个数字，你可以用来表达0~15，也可以表达-1~14。

要是这么说的话，表达范围不就小了吗（至少从绝对值上）？是的，表达范围是小了，因为4个比特位只能表达16个数字，为了表达负数，必须有所牺牲，若你非得既表达15这个数字，也表达-1这个数字，那就用更多的比特位好了（比如8位、16位等等）。

这里需要指出的是，计算机中实际上很少有4位表达一个整数的，而多是用8位、16位、32位、64位等等，这里我们以4位为例，只是为了方便而已。

说到这里，读者不妨推理一下，8位系统的-1如何表达呢？对了，00000000 - 00000001 = 11111111，也就是8个1，那在16位系统中的-1呢？对，16个1，32位系统中的-1是32个1，64位系统中的-1则是64个1。

3.C 更多的负数的表达

在本文中，我们所介绍的负数的表达方式有一个专门的名称：**补码**，记住这个在考试中坑了无数人，又是程序员不得不了解的一个名词吧 ^_^ ~~~

还是回到简单的4位系统，知道了0000 - 0001 = 1111来表达 -1，那么-2呢？对了，继续做减法，也就是1111 - 0001 = 1110，1110即表示-2，以此类推：1101表示-3，1100表示-4，1011表示-5，1010表示-6，1001表示-7，1000表示-8，那继续往下，1000 - 0001 = 0111，0111呢？

在4位系统中，我们对负数的表达到-8，也就是1000为止，0111继续表示为7，而不是-9。我们不难发现，从1111降到1000，首比特位都是1，这些数字从-1降到-8，都是负数，而0000递增到0111，则表示0~7，这些都是非负数（0和正数）。不难发现，在补码系统中，第一个比特位（即“首位”）为1的数字都是负数，而首位为0的数字都是非负数。于是，四位补码系统中16个完整的数字为：

在这里，我们表达了-8 到7共16个整数。这些整数有以下几个特点：

1. 二进制中以0开头的为非负数，以1开头的为负数。

2. 从0000到0111是递增的，其表的数据0 ~ 7也是递增的。从1000 ~ 1111是递增的，其表的数据 -8 至 -1 也是递增的。

3. 二进制数据0111表达的是最大整数7，0111只要加上1，就会变成1000，而1000表达的是最小数字 -8 。

4. 0和-1同样只有一步之遥，但是表达他们的二进制数字却是最小的0000和最大的1111。

关于以上的第3和第4点，我们不如用下面这个补码环来表示：

补码环中，二进制码的分界线出现在环的上侧，也就是从1111到0000，此时其表达的数字仅仅递增了1：从-1到0，但是这是从负数跨越到了非负数。而环中所表达的数字的分界线出现在环的下侧，即从7跌落至-8，而此时二进制码也只是递增了1：从0111到1000，二进制码的首比特位从0变成了1。

3.D 补码的计算

补码的一大优点就是计算机在做加法或减法运算时，不需要考虑补码的存在而直接运算，结果却依然正确。我们可以利用竖式计算 -2+5，以及2-5（-2、2、5在四位补码系统中分别为1110、0010和0101）：

从左式中我们可以看到，1110（-2）和0101（5）求和，得到10011，去掉首位进位1，

留下0011，即为答案3。右式中，0010（2）强行借位后减去0101（5），得到1101，在表1 5中我们看到，1101即为答案-3。

3.E 补码的计算中的溢出

这种计算其实不总是正确的，如果 0110+0110，也就是 6+6，会得到1100。而1100在补码表示的是-4，却不是我们希望的12。究其原因是12已经超出了四位补码系统所能表达的最大数字7（0111）。我们把这种现象叫做“溢出”。“溢出”的问题我们之前在上一讲中已经介绍了，防止溢出通常是采用更多的数据位，有时候调整一下计算次序也可以避免溢出。

3.F 扩展到N位的补码

我们可以将四位补码推广到N位补码系统。在N位补码系统中，有以下规律：

1. 各位全部为0，总是表示0；
2. 各位全部为1，总是表示 -1；
3.首位为0，后面全部为1，表示的是最大整数：2的N-1次方减去1；
4.首位为1，后面全部为0，表示的是最小整数：负的2的N-1次方 。

计算机中常用的8、16、32、64位补码如下表：

一般的，在计算机中，一个长度为N的整数，既可以表达为不考虑符号的非负整数，我们称作**无符号整数**，也可以表达为一个用补码表达的有正有负的数字，我们称为**有符号整数**。有符号数的负数部分通常用补码表示。不同的位数所能表达的无符号整数，以及有符号整数的范围如下：

好了，我们现在可以总结一下补码的两条基本规则啦：

1. -1是如何表达的，

2. 从0000...000开始递增，产生所有的非负整数，从1111...111开始递减，产生所有的负数，且保持首比特位为0的是非负整数，首比特为1的是负数。

最后一个问题：

哈哈，如果你在猜测这个是-1还是255那就有问题了，你首先应该问的是：老师，这个字节的编码规则是什么？无符号整数，有符号整数？还是两个整数？还是八个独立的比特位……

写在本节之最后：

《程序设计基础》系列从第0讲到第三讲，已经写了四篇了。每一篇都要花去三四个小时的时间，而且很多文字在之前的讲义中早已写过。这仅仅是4个45分钟的授课内容。

至此，二进制部分还有一个内容没有讲，就是小数的表达。这部分对于学习者来说，若想要深入了解，可以看看。

关注计算机教育，关注骏扬工作室