的值是多少，大家知道吗？

代码清单 2-1　将变量 a 的值左移两位的 C 语言程序

```
a = 39;
b = a << 2;
```

如果你认为“由于移位运算是针对二进制数值的位操作，十进制数 39 的移位操作就行不通了”，那么就请重新读一下本章的内容。无论程序中使用的是几进制，计算机内部都会将其转换成二进制数来处理，因此都能进行移位操作。但是，“左移后空出来的低位，要补上什么样的数值呢？”想到这个问题的人真是思维敏锐！空出来的低位要进行补 0 操作。不过，这一规则只适用于左移运算。至于右移时空出来的高位要进行怎样的操作，我们会在后面说明。此外，移位操作使最高位或最低位溢出的数字，直接丢弃就可以了。

接下来让我们继续来看代码清单 2-1。十进制数 39 用 8 位的二进制表示是 00100111，左移两位后是 10011100，再转换成十进制数就是 156。不过这里没有考虑数值的符号。至于其原因，之后大家就知道了。

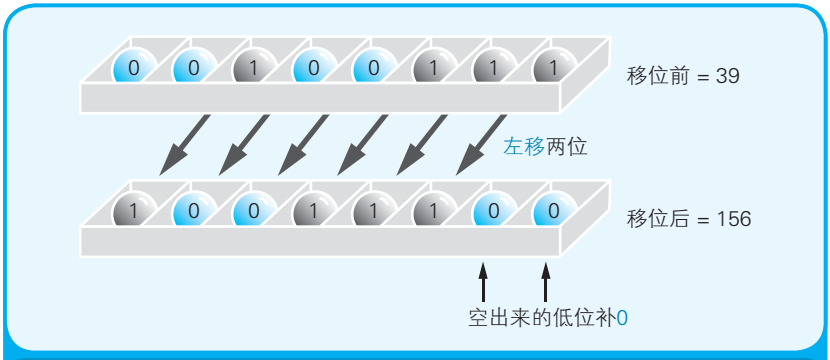

图 2-4　左移两位的运算

实际的程序中，移位运算以及将在本章最后介绍的逻辑运算在使用位单位处理信息的情况下比较常用。虽然这里没有列举具体的程序示例，但对程序员来说，掌握位运算和逻辑运算的机制是一项基本技能，所以一定要掌握。形象地说，移位运算就好比使用二进制表示的图片模式像霓虹灯一样左右流动的样子。

不过，移位运算也可以通过数位移动来代替乘法运算和除法运算。例如，将 00100111 左移两位的结果是 10011100，左移两位后数值变成了原来的 4 倍。用十进制数表示的话，数值从 39（00100111）变成了 156（10011100），也正好是 4 倍（39×4 = 156）。

其实，反复思考几遍后就会发现确实如此。十进制数左移后会变成原来的 10 倍、100 倍、1000 倍……同样，二进制数左移后就会变成原来的 2 倍、4 倍、8 倍……反之，二进制数右移后则会变成原来的 1/2、1/4、1/8……这样一来，大家应该能够理解为什么移位运算能代替乘法运算和除法运算了吧。

2.4 便于计算机处理的“补数”

刚才之所以没有介绍有关右移的内容，是因为用来填充右移后空出来的高位的数值，有 0 和 1 两种形式。要想区分什么时候补 0 什么时候补 1，只要掌握了用二进制数表示负数的方法即可。这部分内容较多，接下来我们就一起来看看表示负数的方法和右移的方法。

二进制数中表示负数值时，一般会把最高位作为符号来使用，因此我们把这个最高位称为符号位。**符号位**是 0 时表示正数，符号位是 1 时表示负数。那么 –1 用 8 位二进制数来表示的话是什么样的呢？可能很多人会认为“1 的二进制数是 00000001，因此 –1 就是 10000001”，

但这个答案是错的，正确答案是 11111111。

计算机在做减法运算时，实际上内部是在做加法运算。用加法运算来实现减法运算，是不是很新奇呢？为此，在表示负数时就需要使用“二进制的补数”。**补数**就是用正数来表示负数，很不可思议吧。

为了获得补数，我们需要将二进制数的各数位的数值全部取反[①]，然后再将结果加 1。例如，用 8 位二进制数表示 −1 时，只需求得 1，也就是 00000001 的补数即可。具体来说，就是将各数位的 0 取反成 1，1 取反成 0，然后再将取反的结果加 1，最后就转化成了 11111111（图 2-5）。

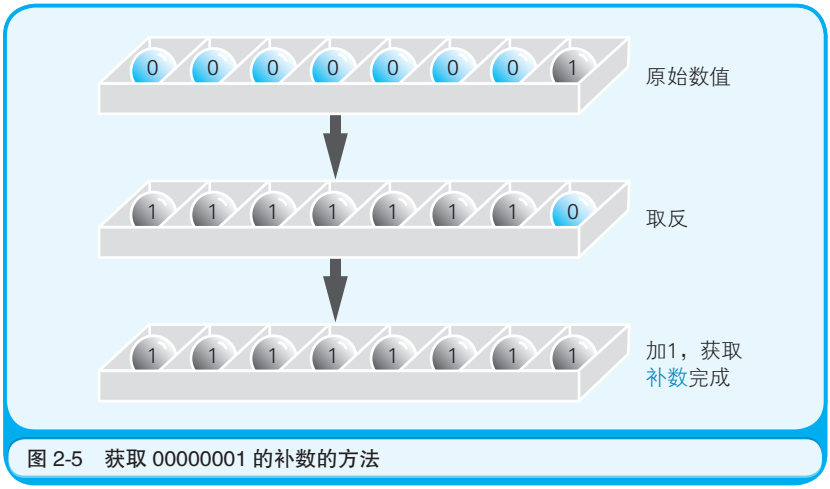

图 2-5　获取 00000001 的补数的方法

补数的思考方式，虽然直观上不易理解，但逻辑上却非常严谨。

① 这里所说的取反是指，把二进制数各数位的 0 变成 1，1 变成 0。例如 00000001 这个 8 位二进制数取反后就成了 11111110。

例如 1 – 1，也就是 1 +(– 1) 这一运算，我们都知道答案应该是 0。首先，让我们将 –1 表示成 10000001（错误的表示方法）来运算，看看结果如何。00000001 + 10000001 = 10000010，很明显结果不是 0（图 2-6）。如果结果是 0，那么所有的数位都应该是 0 才对。

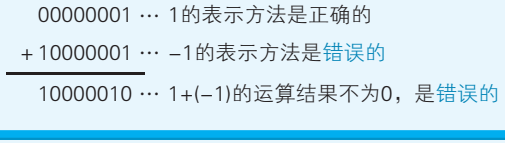

图 2-6　负数表示有误时的情况

接下来，让我们把 – 1 表示成 11111111（正确的表示方法）来进行运算。 00000001 + 11111111 确实为 0（= 00000000）。这个运算中出现了最高位溢出的情况，不过，正如之前所介绍的那样，对于溢出的位，计算机会直接忽略掉。在 8 位的范围内进行计算时，100000000 这个 9 位二进制数就会被认为是 00000000 这一 8 位二进制数（图 2-7）。

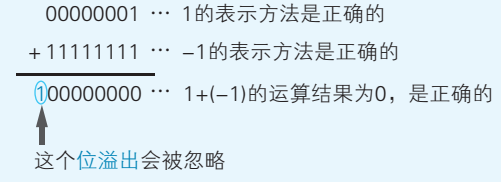

图 2-7　负数表示正确时的情况

补数求解的变换方法就是“取反 + 1”。为什么使用补数后就能正确地表示负数了呢？为了加深印象，我们来看一下图 2-7 ，与此同时也希望大家能够牢记“将二进制数的值取反后加 1 的结果，和原来的值相

加，结果为 0” 这一法则[①]。首先，大家可以用 1 和 –1 的二进制形式，来彻底地了解补数的相关内容。除了 1 + (–1) 之外，2 + (–2)、39 + (–39) 等同样如此。总之，要想使结果为 0，就必须通过补数来实现。

当然，结果不为 0 的运算同样可以通过使用补数来得到正确的结果。不过，有一点需要注意，当运算结果为负数时，计算结果的值也是以补数的形式来表示的。比如 3 – 5 这个运算，用 8 位二进制数表示 3 时为 00000011，而 5 = 00000101 的补数为“取反 + 1”，也就是 11111011。因此 3 – 5 其实就是 00000011 + 11111011 的运算。

00000011 + 11111011 的运算结果为 11111110，最高位变成了 1。这就表示结果是一个负数，这点大家应该都能理解。那么 11111110 表示的负数是多少大家知道吗？这时我们可以利用负负得正这个性质。假若 11111110 是负△△，那么 11111110 的补数就是正△△。通过求解补数的补数，就可知该值的绝对值。11111110 的补数，取反加 1 后为 00000010。这个是 2 的十进制数。因此，11111110 表示的就是 –2。我们也就得到了 3 – 5 的正确结果（图 2-8）。

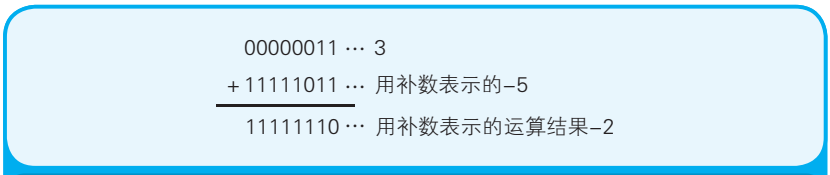

图 2-8　3 – 5 的运算结果

① 例如，00000001 和取反后的 11111110 相加，结果为 11111111，全部数位均为 1。因此，比 11111110 大 1 的数加上 00000001 后，11111111 变为 9 位的 100000000。由于在 8 位的范围内运算时第 9 位会被计算机忽略，因此结果就变成了 00000000。

编程语言包含的整数数据类型[①]中，有的可以处理负数，有的则不能处理。例如，C 语言的数据类型中，既有不能处理负数的 unsigned short 类型，也有能处理负数的 short 类型。这两种类型，都是 2 字节（=16 位）的变量，都能表示 2 的 16 次幂 = 65536 种值，这一点是相同的。不过，值的范围有所不同，short 类型是 –32768～32767，unsigned short 类型是 0～65535。此外，short 类型和 unsigned short 类型的另一个不同点在于，short 类型是将最高位为 1 的数值看作补数，而 unsigned short 类型则是 32768 以上的值。

仔细思考一下补数的机制，大家就会明白像 –32768～32767 这样负数比正数多一个的原因了。最高位是 0 的正数，有 0～32767 共 32768 个，这其中也包含 0。最高位是 1 的负数，有 –1～ –32768 共 32768 个，这其中不包含 0。也就是说，0 包含在正数范围内，所以负数就要比正数多 1 个。虽然 0 不是正数，但考虑到符号位，就将其划分到了正数中。

2.5 逻辑右移和算术右移的区别

在了解了补数后，让我们返回到右移这个话题。前文已经介绍过，右移有移位后在最高位补 0 和补 1 两种情况。当二进制数的值表示图形模式而非数值时，移位后需要在最高位补 0。类似于霓虹灯往右滚动的效果。这就称为**逻辑右移**（图 2-9）。

① 多数编程语言都会把数据代入变量来进行处理。变量中会指定可以存储的数值的种类（整数还是小数）和表示数值大小（位数）的数据类型。C 语言的数据类型中，有用于整数的 char、unsigned char、short、unsigned short、int、unsigned int 和用于小数的 float、double 等。关于数据类型的详细内容，我们会在第 4 章进行说明。

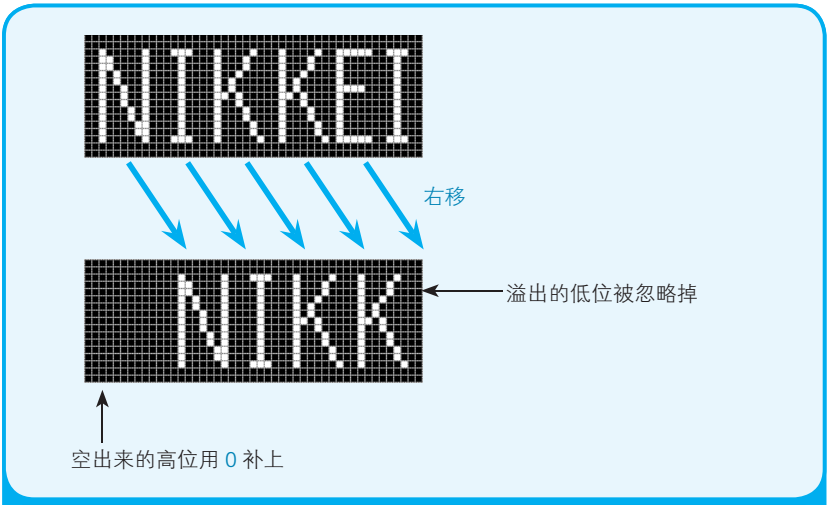

图 2-9　图形模式的右移（逻辑右移）

将二进制数作为带符号的数值进行运算时，移位后要在最高位填充移位前符号位的值（0 或 1）。这就称为算术右移。如果数值是用补数表示的负数值，那么右移后在空出来的最高位补 1，就可以正确地实现 1/2、1/4、1/8 等的数值运算。如果是正数，只需在最高位补 0 即可。

现在我们来看一个右移的例子。将 –4（=11111100）右移两位。这时，逻辑右移的情况下结果就会变成 00111111，也就是十进制数 63，显然不是 –4 的 1/4。而算术右移的情况下，结果就会变成 11111111，用补数表示就是 –1，即 –4 的 1/4（图 2-10）。

只有在右移时才必须区分逻辑位移和算术位移。左移时，无论是图形模式（逻辑左移）还是相乘运算（算术左移），都只需在空出来的低位补 0 即可。

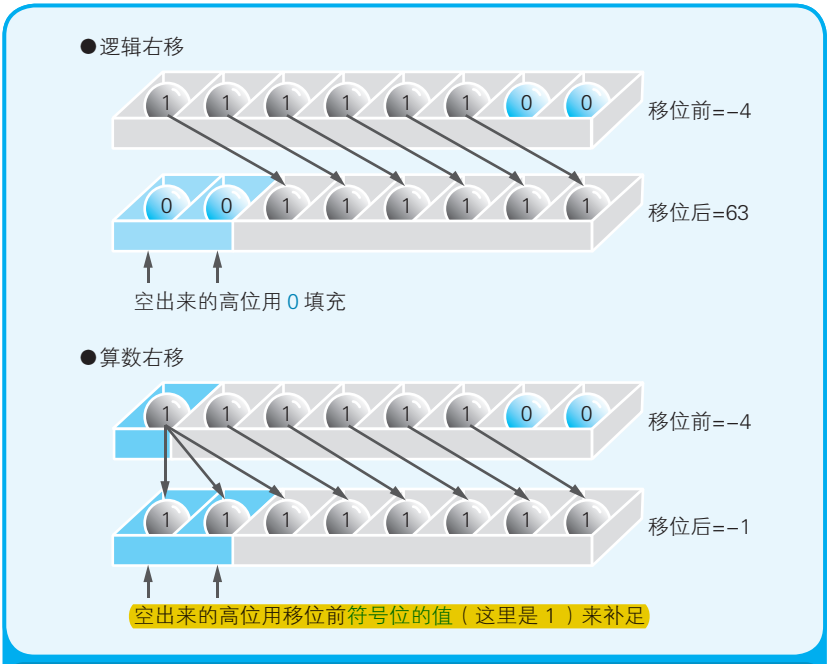

图 2-10 逻辑右移和算术右移的区别

下面顺便介绍一下符号扩充。以 8 位二进制数为例，**符号扩充就是指在保持值不变的前提下将其转换成 16 位和 32 位的二进制数**。将 01111111 这个正的 8 位二进制数转换成 16 位二进制数时，很容易就能得出 0000000001111111 这个正确结果，但是像 11111111 这样用补数来表示的数值，该如何处理比较好呢？实际上处理方法非常简单，将其表示成 1111111111111111 就可以了。也就是说，**不管是正数还是用补数表示的负数，都只需用符号位的值（0 或者 1）填充高位即可**。这就是符号扩充的方法。图 2-11 向我们展示了将符号位扩充到高位的具体流程。

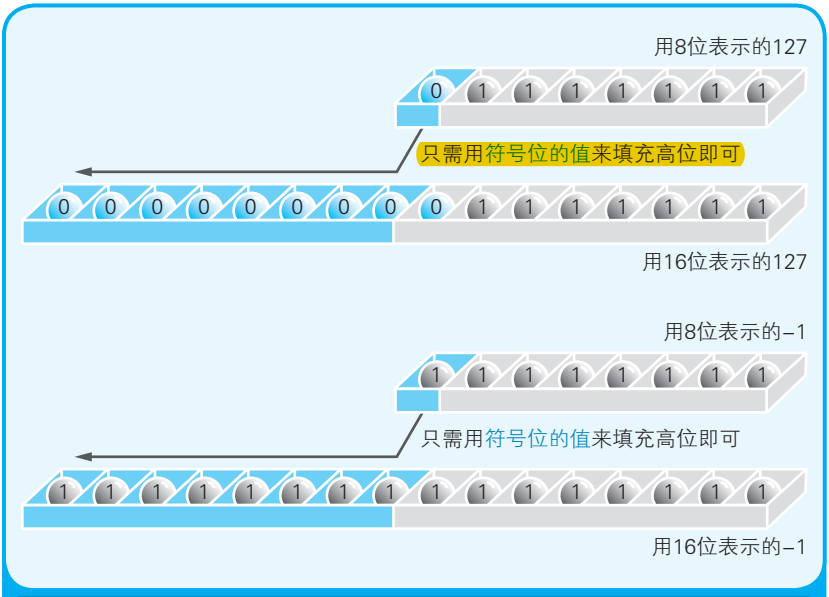

图 2-11　由 8 位转换成 16 位的符号扩充方法

2.6　掌握逻辑运算的窍门

解释逻辑右移时，提及了“逻辑”这个术语。大家听到逻辑这个词可能会感觉有些难，但实际上它很简单。在运算中，与逻辑相对的术语是算术。我们不妨这样考虑，将二进制数表示的信息作为四则运算的数值来处理就是**算术**。而像图形模式那样，将数值处理为单纯的 0 和 1 的罗列就是**逻辑**。

计算机能处理的运算，大体可分为算术运算和逻辑运算。**算术运算**是指加减乘除四则运算。**逻辑运算**是指对二进制数各数字位的 0 和 1 分别进行处理的运算，包括逻辑非（NOT 运算）、逻辑与（AND 运算）、逻辑或（OR 运算）和逻辑异或（XOR 运算[①]）四种。

① XOR 是英语 exclusive or 的缩写。有时也将 XOR 称为 EOR。

逻辑非指的是 0 变成 1、1 变成 0 的取反操作。**逻辑与**指的是“两个都是 1”时，运算结果为 1，其他情况下运算结果都为 0 的运算。**逻辑或**指的是“至少有一方是 1”时，运算结果为 1，其他情况下运算结果都是 0 的运算。**逻辑异或**指的是排斥相同数值的运算。“两个数值不同”，也就是说，当“其中一方是 1，另一方是 0”时运算结果是 1，其他情况下结果都是 0。不管是几位的二进制数，在进行逻辑运算时，都是对相对应的各数位分别进行运算。

表 2-1～表 2-4 总结了各逻辑运算的结果。这些表称为**真值表**。如果将二进制数的 0 作为假（false）、1 作为真（true）来考虑，逻辑运算也可以被认为是真假的运算。真和真的 AND 运算结果为真，实际上也确实如此。因为如果两方面都是真，答案就是真。

表 2-1　逻辑非（NOT）的真值表

A 的值	NOT A 的运算结果
0	1
1	0

表 2-2　逻辑与（AND）的真值表

A 的值	B 的值	A AND B 的运算结果
0	0	0
0	1	0
1	0	0
1	1	1

表 2-3　逻辑或（OR）的真值表

A 的值	B 的值	A OR B 的运算结果
0	0	0
0	1	1
1	0	1
1	1	1

表 2-4　逻辑异或（XOR）的真值表

A 的值	B 的值	A XOR B 的运算结果
0	0	0
0	1	1
1	0	1
1	1	0

掌握逻辑运算的窍门，就是要摒弃用二进制数表示数值这一想法。大家不要把二进制数表示的值当作是数值，而应该把它看作是图形或者开关上的 ON/OFF（1 是 ON，0 是 OFF）。逻辑运算的运算对象不是数值，因此不会出现进位的情况。看起来好像有些麻烦，总之就是不