

位运算

整数类型的二进制存储

C++里的整数类型，包括 `char`，`int`，`long` 等等，都是用二进制表示存储的。

下面是完整的 C++ 整数类型及其变体的详细信息表，包括 `char` 类型和其他整数类型的字节大小和范围，这里使用的范围表达是基于 2 的次方：

类型	大小 (字节)	范围	备注
<code>char</code>	1	实现定义，通常为 $[-2^7, 2^7 - 1]$ 或 $[0, 2^8 - 1]$	符号性由编译器决定
<code>signed char</code>	1	$[-2^7, 2^7 - 1]$	有符号字符类型
<code>unsigned char</code>	1	$[0, 2^8 - 1]$	无符号字符类型
<code>short</code>	2	$[-2^{15}, 2^{15} - 1]$	
<code>unsigned short</code>	2	$[0, 2^{16} - 1]$	
<code>int</code>	4	$[-2^{31}, 2^{31} - 1]$	在大多数系统上为4字节
<code>unsigned int</code>	4	$[0, 2^{32} - 1]$	在大多数系统上为4字节
<code>long</code>	4 或 8	32位: $[-2^{31}, 2^{31} - 1]$, 64位: $[-2^{63}, 2^{63} - 1]$	系统依赖，32位或64位
<code>unsigned long</code>	4 或 8	32位: $[0, 2^{32} - 1]$, 64位: $[0, 2^{64} - 1]$	系统依赖，32位或64位
<code>long long</code>	8	$[-2^{63}, 2^{63} - 1]$	C++11 引入
<code>unsigned long long</code>	8	$[0, 2^{64} - 1]$	C++11 引入

整型数据类型有无符号和有符号之分。无符号整数只能表示非负数，而有符号整数可以表示负数。例如，十进制数 $234 = 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$ ，那么二进制数转换为十进制数是类似的，比如 $(1101)_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$ 。无符号整数表示的数可以按照前面这种公式计算。而有符号整数则有所不同，其是按补码存储的。我们考虑 `signed char` 范围内的数，显然十进制的1在二进制里是00000001，我们考虑-1应该怎么表示，由于计算机做加法的时候，其实和人类似的，也可以看成是从最低位开始加起，加到高位，产生进位要加上去，但是，如果在最高位产生了进位，由于整数类型有范围限制，那里的进位将不得不放弃，这就启发我们，十进制的-1可以表示为11111111，这样计算11111111 + 00000001的时候，由于 `signed char` 最多就只有8个二进制位，从而这个加法里将不得不放弃掉最高位产生的进位，从而这个加法最后得到的是00000000，最高位的1存不下了，放弃了。

所以我们发现，对于一个数，要求它的相反数，就是要构造出一个数，两个数加起来，恰好在最高位产生进位，并且其余位加起来最后恰好等于0。我们定义 `~` 这个运算符，这个运算符是取反，即每一位的1变为0，0变为1，那容易知道对于任意 `signed char` 数 x ，有 $x + \sim x = 11111111$ ，而 $x + \sim x + 1 = 00000000$ ，从而 $-x = \sim x + 1$ ，这就得到了一个数的负数在计算机里的表示。正整数（还有0）的最高位是0，而负整数的最高位是1，最高位被称为符号位。注意，符号位只在有符号整数里有效。

位运算

位运算是对于整数在位级别上的操作，通常用于处理二进制数。这里有五种常用的位运算符：

- `&`（按位与）：两个位都为1时，结果位为1。
 - 示例：`5 & 3`（二进制 `101 & 011`）结果为 `001`，即1。
- `|`（按位或）：两个位中有一个为1时，结果位为1。
 - 示例：`5 | 3`（二进制 `101 | 011`）结果为 `111`，即7。
- `^`（按位异或）：两个位相异时，结果位为1。
 - 示例：`5 ^ 3`（二进制 `101 ^ 011`）结果为 `110`，即6。
- `~`（按位取反）：位为0则结果位为1，位为1则结果位为0。
 - 示例：`~5`（二进制 `~101`）结果为 `...010`，前面的...都填成1，计算时需考虑整数类型的位宽。
- `<<`（左移）：将二进制全部左移指定位数，右边空出的位用0填充。
 - 示例：`5 << 1`（二进制 `101` 左移1位）结果为 `1010`，即10。
- `>>`（右移）：将二进制全部右移指定位数，左边空出的位根据数的符号位填充（正数填充0，负数填充1）。
 - 示例：`5 >> 1`（二进制 `101` 右移1位）结果为 `010`，即2。

如果我们将1视为真，0视为假，那么 `&` 和逻辑联结词 `&&` 非常相似，即真 `&` 真也等于真，等等。`|` 和 `~` 也有类似的意思。`^` 可以理解为，当且仅当两个数其中一个是真的时候，结果才是真。

<< 对于无符号整数来说，相当于做了一次“乘以 2”的操作，当然在不考虑溢出的情况下。如果溢出，那么得到的数很可能不是预期的值。例如 `unsigned char x = 128`，其二进制表示是 `10000000`，那么 `x << 1` 将会得到 `0`，因为高位溢出了，直接消失了。<< 对于有符号整数来说，如果不溢出，那么效果也是乘以 2。读者可以验证。可以用补码的思维过一遍，就能理解。

>> 对于无符号整数来说，相当于做了一次“除以 2”的操作，且下取整，也就是说，对于无符号整数 `x` 来说，`x >> 1 = x / 2` 是恒成立的。而 >> 对于有符号整数来说，如果是非负整数，那么效果也是除以 2。但是对于负整数来说，事实上如果 `x` 是奇数，`x >> 1 = x / 2 - 1`，如果 `x` 是偶数，`x >> 1 = x / 2`，我们也可以用补码的思维想一遍，因为正整数是向下取整，所以对于负整数，如果 `x` 是偶数，那么 `x` 和 `-x` 的最低位一定是 0，从而 `x >> 1` 和 `(-x) >> 1` 仍然加起来是 0，所以 `x` 是负偶数的时候，`x >> 1 = x / 2`；但是如果 `x` 是负奇数，那么 `-x` 也是奇数，最低位都是 1，右移之后，只有 `x >> 1` 和 `((-x) >> 1) + 1` 加起来才是 0，所以负数右移 1 是向下取整，即 `x >> 1 = x / 2 - 1`。

lowbit运算

定义 `lowbit(x) = x & -x`，那么 `lowbit(x)` 表示的是 `x` 的最低位的 1 所表示的整数。用补码的原理想一遍即可知道。

例题：给定一个 `int`，求其二进制表示里 1 的个数。

```
1 #define lowbit(x) ((x) & -(x))
2 int cnt1(int x) {
3     int cnt = 0;
4     while (x) {
5         x -= lowbit(x);
6         cnt++;
7     }
8     return cnt;
9 }
```

异或的特殊性质

对于异或 `^` 运算来讲，其有很多很有意思的性质。异或满足：

交换律：`x ^ y = y ^ x`

结合律：`(x ^ y) ^ z = x ^ (y ^ z)`

有个单位元，为 0，即对于任何 `x`，有 `x ^ 0 = 0 ^ x = x`

每个元素的逆元都是自己，即 `x ^ x = 0`

这个性质可以解决一些有趣的问题。

例题：给定一个数组 `a`，其中只有一个数出现了 1 次，别的数都恰好出现 2 次，问那个只出现了 1 次的数是多少。

```
1 int solve(vector<int>& a) {  
2     int res = 0;  
3     for (int x : a) res ^= x;  
4     return res;  
5 }
```

作业

上面例题。

