# 什么是位运算,你了解 JavaScript 中的位运算么?

在了解位运算之前,我们先必须了解一下什么是原码、反码和补码以及二进制与十进制的转换。

原码、补码和反码

原码

一个数在计算机中是以二进制的形式存在的,其中第一位存放符号,正数为0,负数为1。 原码就是用第一位存放符号的二进制数值。例如2的原码为0000010,-2的原码为10000010。

反码

正数的反码是其本身。负数的反码是在其原码的基础上,符号位不变,其余各位取反,即0变1,1变0。

- [+3]=[00000011]原=[00000011]反
- [-3]=[10000011]原=[11111100]反

可见如果一个反码表示的是负数,并不能直观的看出它的数值,通常要将其转换成原码再计算。

补码

正数的补码是其本身。负数的补码是在其原码的基础上,符号位不变,其余各位取反,最后+1。(即负数的补码为在其反码的基础上+1)。

[+3]=[00000011]原=[00000011]反=[00000011]补

[-3]=[10000011]原=[11111100]反=[11111101]补

可见对于负数,补码的表示方式也是让人无法直观看出其数值的,通常也需要转换成原码再计算。

二进制与十进制的转换

二进制与十进制的区别在于数运算时是逢几进一位。二进制是逢 2 进一位,十进制也就是我们常用的 0-9 是逢 10 进一位

正整数的十进制转二进制

正整数的十进制转二进制的方法为将一个十进制数除以 2, 得到的商再除以 2, 以此类推直到商等于 1 或 0 时为止, 倒取除得的余数, 即为转换所得的二进制数的结果。

例如把52换算成二进制数,计算过程如下图:

2	52				 				 ::	ું	0
2	26	- 1		 	 	 	 		 		0.
7	2 13	3		 	 	 	 		 ٠.		.1
	2	6	_8	 	 	 	 	 *	 		.0
	2	3	170	 • •	 •	 	 	•			.1
	32(4)	1	55	 	 	 	 	 	 		1

52 除以 2 得到的余数依次为: 0、0、1、0、1、1, 倒序排列, 所以 52 对应的二进制数就是110100。

## 负整数的十进制转二进制

负整数的十进制转二进制为将该负整数对应的正整数先转换成二进制,然后对其"取反", 再对取反后的结果+1。即负整数采用其二进制补码的形式存储。

至于负数为什么要用二进制补码的形式存储,可参考一篇阮一峰的文章《关于2的补码》。例如 -52 的原码为 10110100,其反码为 11001011,其补码为 11001100。所以 -52 转换为二进制后为 11001100。

#### 十进制小数转二进制

十进制小数转二进制的方法为"乘2取整",对十进制小数乘2得到的整数部分和小数部分,整数部分即是相应的二进制数码,再用2乘小数部分(之前乘后得到新的小数部分),又得到整数和小数部分。如此不断重复,直到小数部分为0或达到精度要求为止。第一次所得到为最高位,最后一次得到为最低位。

如:0.25 的二进制

- 0.25\*2=0.5 取整是0
- 0.5\*2=1.0 取整是1

即 0.25 的二进制为 0.01 (第一次所得到为最高位,最后一次得到为最低位)

0.8125 的二进制

0.8125\*2=1.625 取整是1

0.625\*2=1.25 取整是1

0.25\*2=0.5 取整是0

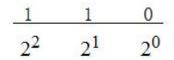
0.5\*2=1.0 取整是1

即 0.8125 的二进制是 0.1101

二进制转十进制

从最后一位开始算,依次列为第0、1、2...位,第n位的数(0或1)乘以2的n次方,将得到的结果相加就是得到的十进制数。

例如二进制为110的数,将其转为十进制的过程如下



个位数 0 与 2° 相乘: 0 × 2° = 0

十位数 1 与 2<sup>1</sup> 相乘: 1 × 2<sup>1</sup> = 2

百位数 1 与 2<sup>2</sup> 相乘: 1 × 2<sup>2</sup> = 4

将得到的结果相加: 0+2+4=6

所以二进制 110 转换为十进制后的数值为 6。

小数二进制用数值乘以2的负幂次然后相加。

JavaScript 中的位运算

在 ECMAScript 中按位操作符会将其操作数转成补码形式的有符号 32 位整数。 JavaScript 中的位运算有: & (按位与)、| (按位或)、~(取反)、^(按位异或)、<< (左移)、>>(有符号右移)和>>>(无符号右移)。

#### &按位与

对每一个比特位执行与 (AND) 操作。只有 a 和 b 都是 1 时, a & b 才是 1。例如: 9(base 10) & 14(base 10) = 1001(base2) & 1110(base 2) = 1000(base 2) = 8(base 10)

因为当只有 a 和 b 都是 1 时, a&b 才等于 1, 所以任一数值 x 与 0 (二进制的每一位都 是 0) 按位与操作, 其结果都为 0。将任一数值 x 与 -1 (二进制的每一位都是 1) 按位与操作, 其结果都为 x。利用 & 运算的特点, 我们可以用以简单的判断奇偶数, 公式:

(n & 1) === 0 //true 为偶数, false 为奇数。

因为 1 的二进制只有最后一位为 1, 其余位都是 0, 所以其判断奇偶的实质是判断二进制数最后一位是 0 还是 1。奇数的二进制最后一位是 1, 偶数是 0。

# 按位或

对每一个比特位执行或 (OR) 操作。如果 a 或 b 为 1, 则 a | b 结果为 1。例如: 9 (base 10) | 14 (base 10) = 1001 (base 2) | 1110 (base 2) = 1111 (base 2) = 15 (base 10)

因为只要 a 或 b 其中一个是 1 时,a b 就等于 1,所以任一数值 x 与-1(二进制的每一位都是 1)按位与操作,其结果都为-1。将任一数值 x 与 0(二进制的每一位都是 0)按位与操作,其结果都为 x。

同样,按位或也可以做向下取整运算,因为当 x 为整数时有 x|0=x,所以当 x 为小数时有 x|0==Math. floor (x) 。

## ~取反

对每一个比特位执行非(NOT)操作。~a 结果为 a 的反转(即反码)。

9 (base 10) = 00000000000000000000000000001001 (base 2)

~9 (base 10) = 11111111111111111111111111111110110 (base 2) = -10 (base 10) 负数的二进制转化为十进制的规则是,符号位不变,其他位取反后加 1。

对任一数值 x 进行按位非操作的结果为 -(x + 1)。 $^{--}x === x$ 。

同样,取反也可以做向下取整运算,因为当 x 为整数时有 ~~x===x,所以当 x 为小数时有 ~~x===Math.floor(x)。

#### ^按位异或

对每一对比特位执行异或 (XOR) 操作。当 a 和 b 不相同时, a ^ b 的结果为 1。例如: 9(base 10) ^ 14(base 10) = 1001(base 2) ^ 1110(base 2) = 0111(base 2) = 7(base 10)

将任一数值 x 与 0 进行异或操作,其结果为 x。将任一数值 x 与 -1 进行异或操作,其结果为 x0,即 x1-1=x1。

同样,接位异或也可以做向下取整运算,因为当 x 为整数时有  $(x^0)===x$ ,所以当 x 为小数时有  $(x^0)===Math$ .floor(x)。

## 《左移运算

它把数字中的所有数位向左移动指定的数量,向左被移出的位被丢弃,右侧用 0 补充。例如,把数字 2 (等于二进制中的 10) 左移 5 位,结果为 64 (等于二进制中的 1000000):

var iOld = 2: //等于二进制 10

var iNew = i0ld << 5; //等于二进制 1000000 十进制 64

因为二进制 10 转换成十进制的过程为  $1\times2^1+0\times2^0$ , 在运算中 2 的指数与位置数相对应, 当左移五位后就变成了  $1\times2^{1+}$   $5+0\times2^{0+}$  5=  $1\times2^1\times2^5+0\times2^0\times2^5=$   $(1\times2^1+0\times2^0)\times2^5$ 。所以由此可以看出当 2 左移五位就变成了  $2\times2^5=64$ 。

所以有一个数左移 n 为, 即为这个数乘以2的n次方。x<<n === x\*2<sup>n</sup>。

同样, 左移运算也可以做向下取整运算, 因为当 x 为整数时有 (x<<0)===x, 所以当 x 为 小数时有 (x<<0)===Math. floor(x)。

# >>有符号右移运算

它把 32 位数字中的所有数位整体右移,同时保留该数的符号(正号或负号)。有符号右移运算符恰好与左移运算相反。例如,把 64 右移 5 位,将变为 2。

因为有符号右移运算符与左移运算相反,所以有一个数左移 n 为,即为这个数除以 2 的 n 次方。 $x << n === x/2^n$ 。

同样,有符号右移运算也可以做向下取整运算,因为当 x 为整数时有 (x>>0)===x,所以当 x 为小数时有 (x>>0)===Math.floor(x)。

#### >>>无符号右移运算

它将无符号 32 位数的所有数位整体右移。对于正数,无符号右移运算的结果与有符号右移运算一样,而负数则被作为正数来处理。

根据无符号右移的正数右移与有符号右移运算一样,而负数的无符号右移一定为非负的特征,可以用来判断数字的正负,如下:

```
function isPos(n) {
  return (n === (n >>> 0)) ? true : false;
}
```

isPos(-1); // false

isPos(1); // true

总结

根据 JS 的位运算, 可以得出如下信息:

- 1、所有的位运算都可以对小数取底。
- 2、对于按位与&,可以用 (n & 1) === 0 //true 为偶数, false 为奇数。来判断奇偶。用 x&-1===Math. floor(x)来向下取底。
- 3、对于按位或 | , 可以用 x | 0===Math. floor (x) 来向下取底。
- 4、对于取反运算~,可以用~~x===Math.floor(x)来向下取底。
- 4、对于异或运算<sup>^</sup>, 可以用(x<sup>^</sup>0)===Math. floor(x)来向下取底。
- 4、对于左移运算<<, 可以 x<<n === x\*2<sup>n</sup> 来求 2 的 n 次方, 用 x<<0===Math. floor (x) 来向下取底。
- 4、对于有符号右移运算>>, 可以  $x<< n === x/2^n$  求一个数字的 N 等分,用 x>>0===Math. floor(x)来向下取底。
- 4、对于无符号右移运算>>>, 可以(n === (n >>> 0))? true: false;来判断数字正负,用 x>>>0===Math. floor(x)来向下取底。

用移位运算来替代普通算术能获得更高的效率。移位运算翻译成机器码的长度更短,执行更

快,需要的硬件开销更小。

DUYIEDUCATION