二进制位运算符

概述

二进制位运算符用于直接对二进制位进行计算，一共有7个。

- **二进制或运算符**（or）：符号为`|`，表示若两个二进制位都为`0`，则结果为`0`，否则为`1`。
- **二进制与运算符**（and）：符号为`&`，表示若两个二进制位都为1，则结果为1，否则为0。
- **二进制否运算符**（not）：符号为`~`，表示对一个二进制位取反。
- **异或运算符**（xor）：符号为`^`，表示若两个二进制位不相同，则结果为1，否则为0。
- **左移运算符**（left shift）：符号为`<<`，详见下文解释。
- **右移运算符**（right shift）：符号为`>>`，详见下文解释。
- **头部补零的右移运算符**（zero filled right shift）：符号为`>>>`，详见下文解释。

这些位运算符直接处理每一个比特位（bit），所以是非常底层的运算，好处是速度极快，缺点是很不直观，许多场合不能使用它们，否则会使代码难以理解和查错。

有一点需要特别注意，位运算符只对整数起作用，如果一个运算子不是整数，会自动转为整数后再执行。另外，虽然在 JavaScript 内部，数值都是以64位浮点数的形式储存，但是做位运算的时候，是以32位带符号的整数进行运算的，并且返回值也是一个32位带符号的整数。

```javascript
i = i | 0;
```

上面这行代码的意思，就是将`i`（不管是整数或小数）转为32位整数。

利用这个特性，可以写出一个函数，将任意数值转为32位整数。

```javascript
function toInt32(x) {
  return x | 0;
}
```

上面这个函数将任意值与`0`进行一次或运算，这个位运算会自动将一个值转为32位整数。下面是这个函数的用法。

```javascript
toInt32(1.001) // 1
toInt32(1.999) // 1
toInt32(1) // 1
toInt32(-1) // -1
toInt32(Math.pow(2, 32) + 1) // 1
toInt32(Math.pow(2, 32) - 1) // -1
```

上面代码中，`toInt32`可以将小数转为整数。对于一般的整数，返回值不会有任何变化。对于大于或等于2的32次方的整数，大于32位的数位都会被舍去。

二进制或运算符

二进制或运算符（`|`）逐位比较两个运算子，两个二进制位之中只要有一个为`1`，就返回`1`，否则返回`0`。

```javascript
0 | 3 // 3
```

上面代码中，`0`和`3`的二进制形式分别是`00`和`11`，所以进行二进制或运算会得到`11`（即`3`）。

位运算只对整数有效，遇到小数时，会将小数部分舍去，只保留整数部分。所以，将一个小数与`0`进行二进制或运算，等同于对该数去除小数部分，即取整数位。

```javascript
2.9 | 0 // 2
-2.9 | 0 // -2
```

需要注意的是，这种取整方法不适用超过32位整数最大值`2147483647`的数。

```javascript
2147483649.4 | 0;
// -2147483647
```

二进制与运算符

二进制与运算符（`&`）的规则是逐位比较两个运算子，两个二进制位之中只要有一个位为`0`，就返回`0`，否则返回`1`。

```javascript
0 & 3 // 0
```

上面代码中，0（二进制`00`）和3（二进制`11`）进行二进制与运算会得到`00`（即`0`）。

二进制否运算符

二进制否运算符（`~`）将每个二进制位都变为相反值（`0`变为`1`，`1`变为`0`）。它的返回结果有时比较难理解，因为涉及到计算机内部的数值表示机制。

```javascript
```

~ 3 // -4
```

上面表达式对`3`进行二进制否运算，得到`-4`。之所以会有这样的结果，是因为位运算时，JavaScript 内部将所有的运算子都转为32位的二进制整数再进行运算。

`3`的32位整数形式是`00000000000000000000000000000011`，二进制否运算以后得到`11111111111111111111111111111100`。由于第一位（符号位）是1，所以这个数是一个负数。JavaScript 内部采用补码形式表示负数，即需要将这个数减去1，再取一次反，然后加上负号，才能得到这个负数对应的10进制值。这个数减去1等于`11111111111111111111111111111011`，再取一次反得到`00000000000000000000000000000100`，再加上负号就是`-4`。考虑到这样的过程比较麻烦，可以简单记忆成，一个数与自身的取反值相加，等于-1。

```javascript
~ -3 // 2
```

上面表达式可以这样算，`-3`的取反值等于`-1`减去`-3`，结果为`2`。

对一个整数连续两次二进制否运算，得到它自身。

```javascript
~~3 // 3
```

所有的位运算都只对整数有效。二进制否运算遇到小数时，也会将小数部分舍去，只保留整数部分。所以，对一个小数连续进行两次二进制否运算，能达到取整效果。

```javascript
~~2.9 // 2
~~47.11 // 47
~~1.9999 // 1
~~3 // 3
```

使用二进制否运算取整，是所有取整方法中最快的一种。

对字符串进行二进制否运算，JavaScript 引擎会先调用`Number`函数，将字符串转为数值。

```javascript
// 相当于~Number('011')
~'011'  // -12

// 相当于~Number('42 cats')
~'42 cats' // -1
```
```

```
// 相当于~Number('0xcafebabe')
~'0xcafebabe' // 889275713

// 相当于~Number('deadbeef')
~'deadbeef' // -1
```

`Number`函数将字符串转为数值的规则，参见《数据的类型转换》一章。

对于其他类型的值，二进制否运算也是先用`Number`转为数值，然后再进行处理。

```javascript
// 相当于 ~Number([])
~[] // -1

// 相当于 ~Number(NaN)
~NaN // -1

// 相当于 ~Number(null)
~null // -1
```

异或运算符

异或运算（`^`）在两个二进制位不同时返回`1`，相同时返回`0`。

```javascript
0 ^ 3 // 3
```

上面表达式中，`0`（二进制`00`）与`3`（二进制`11`）进行异或运算，它们每一个二进制位都不同，所以得到`11`（即`3`）。

“异或运算”有一个特殊运用，连续对两个数`a`和`b`进行三次异或运算，`a^=b; b^=a; a^=b;`，可以[互换](http://en.wikipedia.org/wiki/XOR_swap_algorithm)它们的值。这意味着，使用“异或运算”可以在不引入临时变量的前提下，互换两个变量的值。

```javascript
var a = 10;
var b = 99;

a ^= b, b ^= a, a ^= b;

a // 99
b // 10
```

这是互换两个变量的值的最快方法。

异或运算也可以用来取整。

```javascript
12.9 ^ 0 // 12
```

左移运算符

左移运算符（`<<`）表示将一个数的二进制值向左移动指定的位数，尾部补`0`，即乘以`2`的指定次方。向左移动的时候，最高位的符号位是一起移动的。

```javascript
// 4 的二进制形式为100，
// 左移一位为1000（即十进制的8）
// 相当于乘以2的1次方
4 << 1
// 8

-4 << 1
// -8
```

上面代码中，`-4`左移一位得到`-8`，是因为`-4`的二进制形式是`11111111111111111111111111111100`，左移一位后得到`11111111111111111111111111111000`，该数转为十进制（减去1后取反，再加上负号）即为`-8`。

如果左移0位，就相当于将该数值转为32位整数，等同于取整，对于正数和负数都有效。

```javascript
13.5 << 0
// 13

-13.5 << 0
// -13
```

左移运算符用于二进制数值非常方便。

```javascript
var color = {r: 186, g: 218, b: 85};

// RGB to HEX
// (1 << 24)的作用为保证结果是6位数
var rgb2hex = function(r, g, b) {
  return '#' + ((1 << 24) + (r << 16) + (g << 8) + b)
    .toString(16) // 先转成十六进制，然后返回字符串
```

```
    .substr(1);  // 去除字符串的最高位，返回后面六个字符串
}

rgb2hex(color.r, color.g, color.b)
// "#bada55"
```

上面代码使用左移运算符，将颜色的 RGB 值转为 HEX 值。

右移运算符

右移运算符（`>>`）表示将一个数的二进制值向右移动指定的位数。如果是正数，头部全部补`0`；如果是负数，头部全部补`1`。右移运算符基本上相当于除以`2`的指定次方（最高位即符号位参与移动）。

```javascript
4 >> 1
// 2
/*
// 因为4的二进制形式为 00000000000000000000000000000100,
// 右移一位得到 00000000000000000000000000000010,
// 即为十进制的2
*/

-4 >> 1
// -2
/*
// 因为-4的二进制形式为 11111111111111111111111111111100,
// 右移一位，头部补1，得到 11111111111111111111111111111110,
// 即为十进制的-2
*/
```

右移运算可以模拟 2 的整除运算。

```javascript
5 >> 1
// 2
// 相当于 5 / 2 = 2

21 >> 2
// 5
// 相当于 21 / 4 = 5

21 >> 3
// 2
// 相当于 21 / 8 = 2
```

```
21 >> 4
// 1
// 相当于 21 / 16 = 1
```

头部补零的右移运算符

头部补零的右移运算符（`>>>`）与右移运算符（`>>`）只有一个差别，就是一个数的二进制形式向右移动时，头部一律补零，而不考虑符号位。所以，该运算总是得到正值。对于正数，该运算的结果与右移运算符（`>>`）完全一致，区别主要在于负数。

```javascript
4 >>> 1
// 2

-4 >>> 1
// 2147483646
/*
// 因为-4的二进制形式为11111111111111111111111111111100,
// 带符号位的右移一位，得到01111111111111111111111111111110,
// 即为十进制的2147483646。
*/
```

这个运算实际上将一个值转为32位无符号整数。

查看一个负整数在计算机内部的储存形式，最快的方法就是使用这个运算符。

```javascript
-1 >>> 0 // 4294967295
```

上面代码表示，`-1`作为32位整数时，内部的储存形式使用无符号整数格式解读，值为4294967295（即`(2^32)-1`，等于`11111111111111111111111111111111`）。

开关作用

位运算符可以用作设置对象属性的开关。

假定某个对象有四个开关，每个开关都是一个变量。那么，可以设置一个四位的二进制数，它的每个位对应一个开关。

```javascript
var FLAG_A = 1; // 0001
var FLAG_B = 2; // 0010
var FLAG_C = 4; // 0100
var FLAG_D = 8; // 1000
```

上面代码设置 A、B、C、D 四个开关，每个开关分别占有一个二进制位。

然后，就可以用二进制与运算检验，当前设置是否打开了指定开关。

```javascript
var flags = 5; // 二进制的0101

if (flags & FLAG_C) {
  // ...
}
// 0101 & 0100 => 0100 => true
```

上面代码检验是否打开了开关`C`。如果打开，会返回`true`，否则返回`false`。

现在假设需要打开`A`、`B`、`D`三个开关，我们可以构造一个掩码变量。

```javascript
var mask = FLAG_A | FLAG_B | FLAG_D;
// 0001 | 0010 | 1000 => 1011
```

上面代码对`A`、`B`、`D`三个变量进行二进制或运算，得到掩码值为二进制的`1011`。

有了掩码，二进制或运算可以确保打开指定的开关。

```javascript
flags = flags | mask;
```

二进制与运算可以将当前设置中凡是与开关设置不一样的项，全部关闭。

```javascript
flags = flags & mask;
```

异或运算可以切换（toggle）当前设置，即第一次执行可以得到当前设置的相反值，再执行一次又得到原来的值。

```javascript
flags = flags ^ mask;
```

二进制否运算可以翻转当前设置，即原设置为`0`，运算后变为`1`；原设置为`1`，运算后变为`0`。

```javascript
flags = ~flags;
```

参考链接

- Michal Budzynski, [JavaScript: The less known parts. Bitwise Operators](http://michalbe.blogspot.co.uk/2013/03/javascript-less-known-parts-bitwise.html)
- Axel Rauschmayer, [Basic JavaScript for the impatient programmer](http://www.2ality.com/2013/06/basic-javascript.html)
- Mozilla Developer Network, [Bitwise Operators](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_Operators)