算术运算符

目录

概述
加法运算符
基本规则
对象的相加
余数运算符
自增和自减运算符
数值运算符，负数值运算符
指数运算符
赋值运算符

运算符是处理数据的基本方法，用来从现有的值得到新的值。 JavaScript 提供了多种运算符，覆盖了所有主要的运算。

概述
JavaScript 共提供 10个算术运算符，用来完成基本的算术运算。

加法运算符 ： x + y
减法运算符 ： x - y
乘法运算符 ： x * y
除法运算符 ： x / y
指数运算符 ： x ** y
余数运算符 ： x % y
自增运算符 ： ++x 或者 x++
自减运算符 ： --x 或者 x--
数值运算符 ： +x
负数值运算符 ： -x

减法、乘法、除法运算法比较单纯，就是执行相应的数学运算。下面介绍其他几个算术运算符，重点是加法运算符。

加法运算符

基本规则

加法运算符（ `+` ）是最常见的运算符，用来求两个数值的和。

```
1 + 1 // 2
```

JavaScript 允许非数值的相加。

```
true + true // 2
1 + true // 2
```

上面代码中，第一行是两个布尔值相加，第二行是数值与布尔值相加。这两种情况，布尔值都会自动转成数值，然后再相加。

比较特殊的是，如果是两个字符串相加，这时加法运算符会变成连接运算符，返回一个新的字符串，将两个原字符串连接在一起。

```
'a' + 'bc' // "abc"
```

如果一个运算子是字符串，另一个运算子是非字符串，这时非字符串会转成字符串，再连接在一起。

```
1 + 'a' // "1a"
false + 'a' // "falsea"
```

加法运算符是在运行时决定，到底是执行相加，还是执行连接。也就是说，运算子的不同，导致了不同的语法行为，这种现象称为 “重载”（overload）。由于加法运算符存在重载，可能执行两种运算，使用的时候必须很小心。

```
'3' + 4 + 5 // "345"
3 + 4 + '5' // "75"
```

上面代码中，由于从左到右的运算次序，字符串的位置不同会导致不同的结果。

除了加法运算符，其他算术运算符（比如减法、除法和乘法）都不会发生重载。它们的规则是：所有运算子一律转为数值，再进行相应的数学运算。

```
1 - '2' // -1
1 * '2' // 2
```

```
1 / '2' // 0.5
```

上面代码中，减法、除法和乘法运算符，都是将字符串自动转为数值，然后再运算。

对象的相加

如果运算子是对象，必须先转成原始类型的值，然后再相加。

```
var obj = { p: 1 };
obj + 2 // "[object Object]2"
```

上面代码中，对象 `obj` 转成原始类型的值是 `[object Object]`，再加 `2` 就得到了上面的结果。

对象转成原始类型的值，规则如下。

首先，自动调用对象的 `valueOf` 方法。

```
var obj = { p: 1 };
obj.valueOf() // { p: 1 }
```

一般来说，对象的 `valueOf` 方法总是返回对象自身，这时再自动调用对象的 `toString` 方法，将其转为字符串。

```
var obj = { p: 1 };
obj.valueOf().toString() // "[object Object]"
```

对象的 `toString` 方法默认返回 `[object Object]`，所以就得到了最前面那个例子的结果。

知道了这个规则以后，就可以自己定义 `valueOf` 方法或 `toString` 方法，得到想要的结果。

```
var obj = {
  valueOf: function () {
    return 1;
  }
};

obj + 2 // 3
```

上面代码中，我们定义 `obj` 对象的 `valueOf` 方法返回 `1`，于是 `obj + 2` 就得到了 `3`。这个例子中，由于 `valueOf` 方法直接返回一个原始类型的值，所以不再调用 `toString` 方法。

下面是自定义 `toString` 方法的例子。

```
var obj = {
  toString: function () {
    return 'hello';
  }
};

obj + 2 // "hello2"
```

上面代码中，对象 `obj` 的 `toString` 方法返回字符串 `hello` 。前面说过，只要有一个运算子是字符串，加法运算符就变成连接运算符，返回连接后的字符串。

这里有一个特例，如果运算子是一个 `Date` 对象的实例，那么会优先执行 `toString` 方法。

```
var obj = new Date();
obj.valueOf = function () { return 1 };
obj.toString = function () { return 'hello' };

obj + 2 // "hello2"
```

上面代码中，对象 `obj` 是一个 `Date` 对象的实例，并且自定义了 `valueOf` 方法和 `toString` 方法，结果 `toString` 方法优先执行。

余数运算符

余数运算符（ `%` ）返回前一个运算子被后一个运算子除，所得的余数。

```
12 % 5 // 2
```

需要注意的是，运算结果的正负号由第一个运算子的正负号决定。

```
-1 % 2 // -1
1 % -2 // 1
```

所以，为了得到负数的正确余数值，可以先使用绝对值函数。

```
// 错误的写法
function isOdd(n) {
  return n % 2 === 1;
}
isOdd(-5) // false
```

```
isOdd(-4) // false

// 正确的写法
function isOdd(n) {
  return Math.abs(n % 2) === 1;
}
isOdd(-5) // true
isOdd(-4) // false
```

余数运算符还可以用于浮点数的运算。但是，由于浮点数不是精确的值，无法得到完全准确的结果。

```
6.5 % 2.1
// 0.19999999999999973
```

自增和自减运算符

自增和自减运算符，是一元运算符，只需要一个运算子。它们的作用是将运算子首先转为数值，然后加上 1 或者减去 1。它们会修改原始变量。

```
var x = 1;
++x // 2
x // 2

--x // 1
x // 1
```

上面代码的变量 `x` 自增后，返回 `2`，再进行自减，返回 `1`。这两种情况都会使得，原始变量 `x` 的值发生改变。

运算之后，变量的值发生变化，这种效应叫做运算的副作用（side effect）。自增和自减运算符是仅有的两个具有副作用的运算符，其他运算符都不会改变变量的值。

自增和自减运算符有一个需要注意的地方，就是放在变量之后，会先返回变量操作前的值，再进行自增/自减操作；放在变量之前，会先进行自增/自减操作，再返回变量操作后的值。

```
var x = 1;
var y = 1;

x++ // 1
++y // 2
```

上面代码中， x是先返回当前值，然后自增，所以得到 1； y是先自增，然后返回新的值，所以得到 2。

数值运算符，负数值运算符
数值运算符（ +
）同样使用加号，但它是一元运算符（只需要一个操作数），而加法运算符是二元运算符（需要两个操作数）
。

数值运算符的作用在于可以将任何值转为数值（与 Number函数的作用相同）。

+true // 1
+[] // 0
+{} // NaN

上面代码表示，非数值经过数值运算符以后，都变成了数值（最后一行 NaN 也是数值）。具体的类型转换规则，参见《数据类型转换》一章。

负数值运算符（ -
），也同样具有将一个值转为数值的功能，只不过得到的值正负相反。连用两个负数值运算符，等同于数值运
算符。

var x = 1;
- x // -1
- (-x) // 1

上面代码最后一行的圆括号不可少，否则会变成自减运算符。

数值运算符号和负数值运算符，都会返回一个新的值，而不会改变原始变量的值。

指数运算符
指数运算符（ **）完成指数运算，前一个运算子是底数，后一个运算子是指数。

2 ** 4 // 16

注意，指数运算符是右结合，而不是左结合。即多个指数运算符连用时，先进行最右边的计算。

// 相当于 2 ** (3 ** 2)
2 ** 3 ** 2
// 512

上面代码中，由于指数运算符是右结合，所以先计算第二个指数运算符，而不是第一个。

赋值运算符
赋值运算符（ Assignment Operators）用于给变量赋值。

最常见的赋值运算符，当然就是等号（ =）。

// 将 1 赋值给变量 x
var x = 1;

// 将变量 y 的值赋值给变量 x
var x = y;

赋值运算符还可以与其他运算符结合，形成变体。下面是与算术运算符的结合。

// 等同于 x = x + y
x += y

// 等同于 x = x - y
x -= y

// 等同于 x = x * y
x *= y

// 等同于 x = x / y
x /= y

// 等同于 x = x % y
x %= y

// 等同于 x = x ** y
x **= y

下面是与位运算符的结合（关于位运算符，请见后文的介绍）。

// 等同于 x = x >> y
x >>= y

```
// 等同于 x = x << y
x <<= y

// 等同于 x = x >>> y
x >>>= y

// 等同于 x = x & y
x &= y

// 等同于 x = x | y
x |= y

// 等同于 x = x ^ y
x ^= y
```

这些复合的赋值运算符，都是先进行指定运算，然后将得到值返回给左边的变量。

版本号 #2
由 壳壳虫 创建于 2018-01-07 09:37:11 UTC
由 壳壳虫 更新于 2018-08-30 12:40:44 UTC