# 操作符

ECMA-262 描述了一组用于操作数据值的操作符，包括算术操作符（如加号和减号）、位操作符、关系操作符和布尔操作符等。

## 位运算符

### 按位非（NOT）

按位非操作符由一个波浪线（~）表示，执行按位非的结果就是返回数值的反码。按位非是  
ECMAScript 操作符中少数几个与二进制计算有关的操作符之一。

**正数的取非**

例：65

       128        64        32        16        8        4        2        1

        0            1          0          0         0        0        0        1          65的原码

        1            0          1          1         1        1        1        0          对65的二进制进行按位取反，不是求反码

1            1          0          0         0        0        0        1          对取反后的值再取反，符号位不变

        1            1          0          0         0        0        1        0          对取反后的数据进行加“1”操作

按位非操作的本质：操作数的负值减 1

下面的代码也能得到相同的结果：  
var num1 = 65;  
var num2 = -num1 - 1;  
alert(num2); // "-66"  
虽然以上代码也能返回同样的结果，但由于按位非是在数值表示的最底层执行操作，因此速度更快。

**负数的取非**

例：-66

       128        64        32        16        8        4        2        1

        1            1          0          0         0        0        1        0       -66的原码

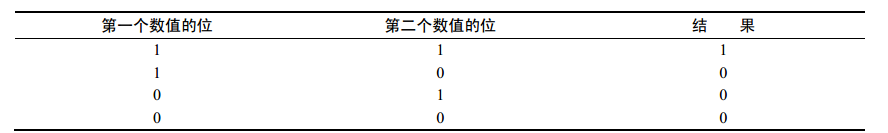
        0            0          1          1         1        1        0        1        按位取反

        0            0          1          1         1        1        1        0        对取反后的二进制数据进行加“1”操作，符合位不变

        0            1          0          0         0        0        0        1         对加“1”后的数据进行取反

### 按位与（AND）

按位与操作符由一个和号字符（&）表示，它有两个操作符数。从本质上讲，按位与操作就是将两个数值的每一位对齐，然后根据下表中的规则，对相同位置上的两个数执行 AND 操作：



正数的与按原码按位与：

25 = 0000 0000 0000 0000 0000 0000 0001 1001  
 3 = 0000 0000 0000 0000 0000 0000 0000 0011  
 ---------------------------------------------  
 AND = 0000 0000 0000 0000 0000 0000 0000 0001

负数的与按补码按位与:

-25

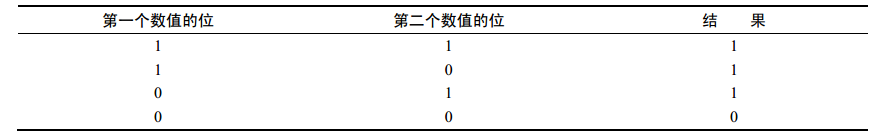
1000 0000 0000 0000 0000 0000 0001 1001 原码

1111 1111 1111 1111 1111 1111 1110 0110 反码（除符号位外，每位取反）

1111 1111 1111 1111 1111 1111 1110 0111 补码 （反码 +１）  
 3 = 0000 0000 0000 0000 0000 0000 0000 0011  
 ---------------------------------------------  
 AND = 0000 0000 0000 0000 0000 0000 0000 0011

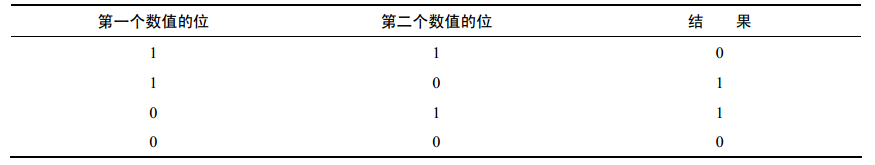
### 按位或（OR）

按位或操作符由一个竖线符号（|）表示，同样也有两个操作数。



### 按位异或（XOR）

按位异或操作符由一个插入符号（^）表示，也有两个操作数。以下是按位异或的真值表



### 左移

左移操作符由两个小于号（<<）表示，这个操作符会将数值的所有位向左移动指定的位数。例如果将数值 2（二进制码为 10）向左移动 5 位，结果就是 64（二进制码为 1000000）.

左移不会影响操作数的符号位。

### 6）有符号的右移

有符号的右移操作符由两个大于号（>>）表示，这个操作符会将数值向右移动，但保留符号位（即正负号标记）。有符号的右移操作与左移操作恰好相反，即如果将 64 向右移动 5 位，结果将变回 2：

var oldValue = 64; // 等于二进制的 1000000

var newValue = oldValue >> 5; // 等于二进制的 10 ，即十进制的 2

### 无符号右移

无符号右移操作符由 3 个大于号（>>>）表示，这个操作符会将数值的所有 32 位都向右移动。对正数来说，无符号右移的结果与有符号右移相同。仍以前面有符号右移的代码为例，如果将 64 无符号右移 5 位，结果仍然还是 2：  
var oldValue = 64; // 等于二进制的 1000000  
var newValue = oldValue >>> 5; // 等于二进制的 10 ，即十进制的 2

但是对负数来说，情况就不一样了。首先，无符号右移是以 0 来填充空位，而不是像有符号右移那样以符号位的值来填充空位。所以，对正数的无符号右移与有符号右移结果相同，但对负数的结果就不 一样了。其次，无符号右移操作符会把负数的二进制码当成正数的二进制码。而且，由于负数以其绝对值的二进制补码形式表示，因此就会导致无符号右移后的结果非常之大，如下面的例子所示：

var oldValue = -64; // 等于二进制的 11111111111111111111111111000000

var newValue = oldValue >>> 5; // 等于十进制的 134217726

这里，当对64 执行无符号右移 5 位的操作后，得到的结果是 134217726。之所以结果如此之大，是因为64 的二进制码为 11111111111111111111111111000000，而且无符号右移操作会把这个二进制码当成正数的二进制码，换算成十进制就是 4294967232。如果把这个值右移 5 位，结果就变成了00000111111111111111111111111110，即十进制的 134217726。

## 布尔操作符

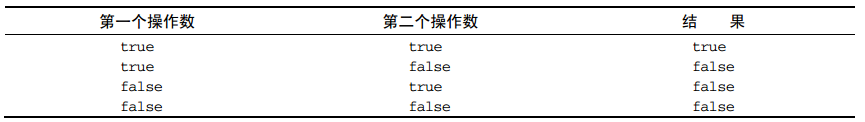
在一门编程语言中，布尔操作符的重要性堪比相等操作符。如果没有测试两个值关系的能力，那么诸如 if...else 和循环之类的语句就不会有用武之地了。布尔操作符一共有 3 个：非（NOT）、与（AND）和或（OR）。

### **1）逻辑非**

逻辑非操作符由一个叹号（！）表示，可以应用于 ECMAScript 中的任何值。无论这个值是什么数据类型，这个操作符都会返回一个布尔值。逻辑非操作符首先会将它的操作数转换为一个布尔值，然后再对其求反。也就是说，逻辑非操作符遵循下列规则：  
 如果操作数是一个对象，返回 false；  
 如果操作数是一个空字符串，返回 true；  
 如果操作数是一个非空字符串，返回 false；  
 如果操作数是数值 0，返回 true；  
 如果操作数是任意非 0 数值（包括 Infinity），返回 false；  
 如果操作数是 null，返回 true；  
 如果操作数是 NaN，返回 true；  
 如果操作数是 undefined，返回 true。

### 逻辑与

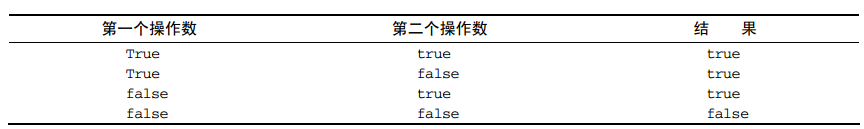
逻辑与操作符由两个和号（&&）表示，有两个操作数，如下面的例子所示：  
var result = true && false;  
逻辑与的真值表如下：



逻辑与操作可以应用于任何类型的操作数，而不仅仅是布尔值。在有一个操作数不是布尔值的情况下，逻辑与操作就不一定返回布尔值；此时，它遵循下列规则：  
 如果第一个操作数是对象，则返回第二个操作数；  
 如果第二个操作数是对象，则只有在第一个操作数的求值结果为 true 的情况下才会返回该  
对象；  
 如果两个操作数都是对象，则返回第二个操作数；  
 如果有一个操作数是 null，则返回 null；  
 如果有一个操作数是 NaN，则返回 NaN；  
 如果有一个操作数是 undefined，则返回 undefined。  
逻辑与操作属于短路操作，即如果第一个操作数能够决定结果，那么就不会再对第二个操作数求值。对于逻辑与操作而言，如果第一个操作数是 false，则无论第二个操作数是什么值，结果都不再可能是true 了。

### 逻辑或

逻辑或操作符由两个竖线符号（||）表示，有两个操作数，如下面的例子所示：  
var result = true || false;  
逻辑或的真值表如下：



与逻辑与操作相似，如果有一个操作数不是布尔值，逻辑或也不一定返回布尔值；此时，它遵循下列规则：  
 如果第一个操作数是对象，则返回第一个操作数；  
 如果第一个操作数的求值结果为 false，则返回第二个操作数；  
 如果两个操作数都是对象，则返回第一个操作数；  
 如果两个操作数都是 null，则返回 null；  
 如果两个操作数都是 NaN，则返回 NaN；  
 如果两个操作数都是 undefined，则返回 undefined。  
与逻辑与操作符相似，逻辑或操作符也是短路操作符。也就是说，如果第一个操作数的求值结果为true，就不会对第二个操作数求值了。

## 算术操作符

### 1）递增和递减操作符

递增和递减操作符直接借鉴自 C，而且各有两个版本：前置型和后置型。顾名思义，前置型应该位于要操作的变量之前，而后置型则应该位于要操作的变量之后。因此，在使用前置递增操作符给一个数值加 1 时，要把两个加号（++）放在这个数值变量前面，如下所示：  
var age = 29;  
++age;  
在这个例子中，前置递增操作符把 age 的值变成了 30（为 29 加上了 1）。实际上，执行这个前置递增操作与执行以下操作的效果相同：  
var age = 29;  
age = age + 1;  
执行前置递减操作的方法也类似，结果会从一个数值中减去 1。使用前置递减操作符时，要把两个减号（--）放在相应变量的前面，如下所示：  
var age = 29;  
--age;  
这样， age 变量的值就减少为 28（从 29 中减去了 1）。  
执行前置递增和递减操作时，变量的值都是在语句被求值以前改变的。  
var age = 29;  
var anotherAge = --age + 2;  
alert(age); // 输出 28  
alert(anotherAge); // 输出 30

由于前置递增和递减操作与执行语句的优先级相等，因此整个语句会从左至右被求值。

### 乘法

乘法操作符由一个星号（\*）表示，用于计算两个数值的乘积。其语法类似于 C，如下面的例子  
所示：  
var result = 34 \* 56;  
在处理特殊值的情况下，乘法操作符遵循下列特殊的规则：  
 如果操作数都是数值，执行常规的乘法计算，即两个正数或两个负数相乘的结果还是正数，而如果只有一个操作数有符号，那么结果就是负数。如果乘积超过了 ECMAScript 数值的表示范围，则返回 Infinity 或-Infinity；  
 如果有一个操作数是 NaN，则结果是 NaN；  
 如果是 Infinity 与 0 相乘，则结果是 NaN；  
 如果是 Infinity 与非 0 数值相乘，则结果是 Infinity 或-Infinity，取决于有符号操作数的符号；  
 如果是 Infinity 与 Infinity 相乘，则结果是 Infinity；  
 如果有一个操作数不是数值，则在后台调用 Number()将其转换为数值，然后再应用上面的  
规则。

### 3）除法

除法操作符由一个斜线符号（/）表示，执行第二个操作数除第一个操作数的计算，如下面的例子所示：  
**var result = 66 / 11;**与乘法操作符类似，除法操作符对特殊的值也有特殊的处理规则。这些规则如下：  
 如果操作数都是数值，执行常规的除法计算，即两个正数或两个负数相除的结果还是正数，而如果只有一个操作数有符号，那么结果就是负数。如果商超过了 ECMAScript 数值的表示范围，则返回 Infinity 或-Infinity；  
 如果有一个操作数是 NaN，则结果是 NaN；  
 如果是 Infinity 被 Infinity 除，则结果是 NaN；  
 如果是零被零除，则结果是 NaN；  
 如果是非零的有限数被零除，则结果是 Infinity 或-Infinity，取决于有符号操作数的符号；  
 如果是 Infinity 被任何非零数值除，则结果是 Infinity 或-Infinity，取决于有符号操作数的符号；

 如果有一个操作数不是数值，则在后台调用 Number()将其转换为数值，然后再应用上面的规则。

### 4）求模

求模（余数）操作符由一个百分号（%）表示，用法如下：  
**var result = 26 % 5; //** 等于 **1**与另外两个乘性操作符类似，求模操作符会遵循下列特殊规则来处理特殊的值：  
 如果操作数都是数值，执行常规的除法计算，返回除得的余数；  
 如果被除数是无穷大值而除数是有限大的数值，则结果是 NaN；  
 如果被除数是有限大的数值而除数是零，则结果是 NaN；  
 如果是 Infinity 被 Infinity 除，则结果是 NaN；  
 如果被除数是有限大的数值而除数是无穷大的数值，则结果是被除数；  
 如果被除数是零，则结果是零；  
 如果有一个操作数不是数值，则在后台调用 Number()将其转换为数值，然后再应用上面的规则。

### 5）加法

加法操作符（+）的用法如下所示：  
var result = 1 + 2;  
如果两个操作符都是数值，执行常规的加法计算，然后根据下列规则返回结果：  
 如果有一个操作数是 NaN，则结果是 NaN；  
 如果是 Infinity 加 Infinity，则结果是 Infinity；  
 如果是-Infinity 加-Infinity，则结果是-Infinity；  
 如果是 Infinity 加-Infinity，则结果是 NaN；  
不过，如果有一个操作数是字符串，那么就要应用如下规则：  
 如果两个操作数都是字符串，则将第二个操作数与第一个操作数拼接起来；  
 如果只有一个操作数是字符串，则将另一个操作数转换为字符串，然后再将两个字符串拼接  
起来。如果有一个操作数是对象、数值或布尔值，则调用它们的 toString()方法取得相应的字符串值，然后再应用前面关于字符串的规则。对于 undefined 和 null，则分别调用 String()函数并取得字符串"undefined"和"null"。

### 6）减法

减法操作符（）是另一个极为常用的操作符，其用法如下所示：  
var result = 2 - 1;  
与加法操作符类似， ECMAScript 中的减法操作符在处理各种数据类型转换时，同样需要遵循一些特殊规则，如下所示：  
 如果两个操作符都是数值，则执行常规的算术减法操作并返回结果；  
 如果有一个操作数是 NaN，则结果是 NaN；  
 如果是 Infinity 减 Infinity，则结果是 NaN；  
 如果是-Infinity 减-Infinity，则结果是 NaN；  
 如果是 Infinity 减-Infinity，则结果是 Infinity；  
 如果是-Infinity 减 Infinity，则结果是-Infinity；

 如果有一个操作数是字符串、布尔值、 null 或 undefined，则先在后台调用 Number()函数将其转换为数值，然后再根据前面的规则执行减法计算。如果转换的结果是 NaN，则减法的结果就是 NaN；  
 如果有一个操作数是对象，则调用对象的 valueOf()方法以取得表示该对象的数值。如果得到的值是 NaN，则减法的结果就是 NaN。如果对象没有 valueOf()方法，则调用其 toString()  
方法并将得到的字符串转换为数值。

## 关系操作符

小于（<）、大于（>）、小于等于（<=）和大于等于（>=）这几个关系操作符用于对两个值进行比较，比较的规则与我们在数学课上所学的一样。这几个操作符都返回一个布尔值，如下面的例子所示：

var result1 = 5 > 3; //true

var result2 = 5 < 3; //false

与 ECMAScript 中的其他操作符一样，当关系操作符的操作数使用了非数值时，也要进行数据转换或完成某些奇怪的操作。以下就是相应的规则。  
 如果两个操作数都是数值，则执行数值比较。  
 如果两个操作数都是字符串，则比较两个字符串对应的字符编码值。  
 如果一个操作数是数值，则将另一个操作数转换为一个数值，然后执行数值比较。  
 如果一个操作数是对象，则调用这个对象的 valueOf()方法，用得到的结果按照前面的规则执行比较。如果对象没有 valueOf()方法，则调用 toString()方法，并用得到的结果根据前面的规则执行比较。  
 如果一个操作数是布尔值，则先将其转换为数值，然后再执行比较。  
在使用关系操作符比较两个字符串时，实际比较的是两个字符串中对应位置的每个字符的字符编码值。经过这么一番比较之后，再返回一个布尔值。比如大写字母的字符编码全部小于小写字母的字符编码

## 相等操作符

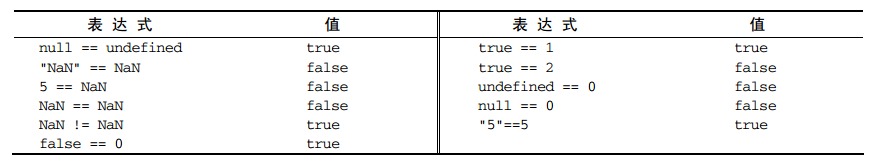
确定两个变量是否相等是编程中的一个非常重要的操作。在比较字符串、数值和布尔值的相等性时，问题还比较简单。但在涉及到对象的比较时，问题就变得复杂了。最早的 ECMAScript 中的相等和不等操作符会在执行比较之前，先将对象转换成相似的类型。后来，有人提出了这种转换到底是否合理的质疑。最后， ECMAScript 的解决方案就是提供两组操作符： 相等和不相等——先转换再比较， 全等和不全等——仅比较而不转换。

### 1）相等和不相等

ECMAScript 中的相等操作符由两个等于号（==）表示，如果两个操作数相等，则返回 true。而不相等操作符由叹号后跟等于号（!=）表示，如果两个操作数不相等，则返回 true。这两个操作符都会先转换操作数（通常称为强制转型），然后再比较它们的相等性。  
在转换不同的数据类型时，相等和不相等操作符遵循下列基本规则：  
 如果有一个操作数是布尔值，则在比较相等性之前先将其转换为数值——false 转换为 0，而true 转换为 1；

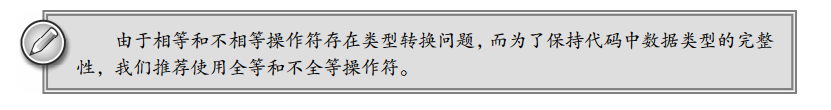
 如果一个操作数是字符串，另一个操作数是数值，在比较相等性之前先将字符串转换为数值；  
 如果一个操作数是对象，另一个操作数不是，则调用对象的 valueOf()方法，用得到的基本类型值按照前面的规则进行比较；  
这两个操作符在进行比较时则要遵循下列规则。  
 null 和 undefined 是相等的。  
 要比较相等性之前，不能将 null 和 undefined 转换成其他任何值。  
 如果有一个操作数是 NaN，则相等操作符返回 false，而不相等操作符返回 true。重要提示：即使两个操作数都是 NaN，相等操作符也返回 false；因为按照规则， NaN 不等于 NaN。  
 如果两个操作数都是对象，则比较它们是不是同一个对象。如果两个操作数都指向同一个对象，则相等操作符返回 true；否则，返回 false。

下表列出了一些特殊情况及比较结果：



### 2）全等和不全等

除了在比较之前不转换操作数之外，全等和不全等操作符与相等和不相等操作符没有什么区别。全等操作符由 3 个等于号（===）表示，它只在两个操作数未经转换就相等的情况下返回 true，如下面的例子所示：  
var result1 = ("55" == 55); //true，因为转换后相等  
var result2 = ("55" === 55); //false，因为不同的数据类型不相等



## 条件操作符

条件操作符应该算是 ECMAScript 中最灵活的一种操作符了，而且它遵循与 Java 中的条件操作符相同的语法形式，如下面的例子所示：  
variable = boolean\_expression ? true\_value : false\_value;  
本质上，这行代码的含义就是基于对 boolean\_expression 求值的结果，决定给变量 variable  
赋什么值。如果求值结果为 true，则给变量 variable 赋 true\_value 值；如果求值结果为 false，则给变量 variable 赋 false\_value 值。再看一个例子：  
var max = (num1 > num2) ? num1 : num2;  
在这个例子中， max 中将会保存一个最大的值。这个表达式的意思是：如果 num1 大于 num2（关系表达式返回 true），则将 num1 的值赋给 max；如果 num1 小于或等于 num2（关系表达式返回 false），则将 num2 的值赋给 max。

## 赋值操作符

简单的赋值操作符由等于号（=）表示，其作用就是把右侧的值赋给左侧的变量，如下面的例子所示：  
var num = 10;  
如果在等于号（=）前面再添加乘性操作符、加性操作符或位操作符，就可以完成复合赋值操作。  
这种复合赋值操作相当于是对下面常规表达式的简写形式：  
var num = 10;  
num = num + 10;  
其中的第二行代码可以用一个复合赋值来代替：  
var num = 10;  
num += 10;  
每个主要算术操作符（以及个别的其他操作符）都有对应的复合赋值操作符。这些操作符如下所示：  
 乘/赋值（\*=）；  
 除/赋值（/=）；  
 模/赋值（%=）；  
 加/赋值（+=）；  
 减/赋值（=）；  
 左移/赋值（<<=）；

## 逗号操作符

使用逗号操作符可以在一条语句中  
var num1=1, num2=2, num3=3;  
逗号操作符多用于声明多个变量.

科普：

原码：将一个整数，转换成二进制，就是其原码。

               如单字节的5的原码为：0000... 0101；-5的原码为1000... 0101。

反码：正数的反码就是其原码；负数的反码是将原码中，除符号位以外，每一位取反。

               如单字节的5的反码为：0000 0101；-5的反码为1111 1010。

补码：正数的补码就是其原码；负数的反码+1就是补码。

