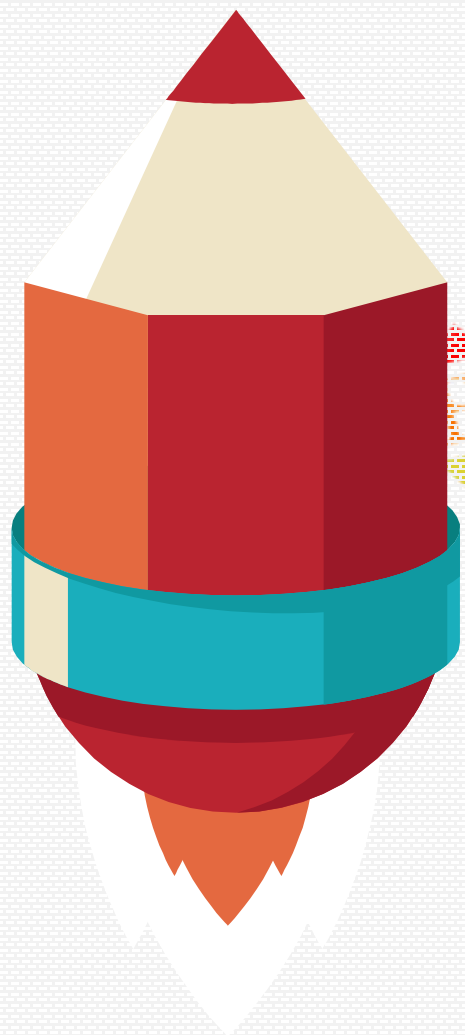




# 第八课：运算符详解

■ 主讲老师：万章



## 四则

加性操作符

布尔操作符

一元操作符

乘性操作符

关系操作符



# 加性操作符

# 加性操作符之加法操作符

**加性** 操作符 包括加法和减法(从本质上讲, 减法其实就是加一个负数)  
值在进行加性操作符操作的时候会**自动**进行一系列的**数据类型转换** !!! !!!

加法操作符的用法如下

变量 = 值1 + 值2 + .....;

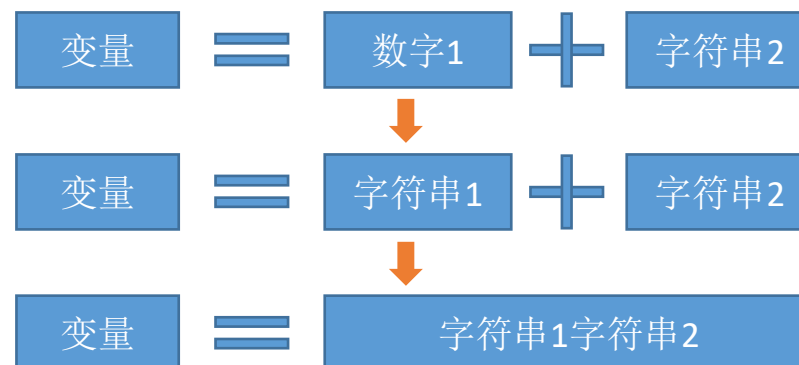
值1, 值2被称为: 操作数  
+ 被称为: 操作符

1: 如果两个操作数都是数值, 执行常规的加法计算, 如果有一个操作数是NaN, 则结果是NaN;

2: 如果有一个操作数是字符串, 那么就要应用如下规则

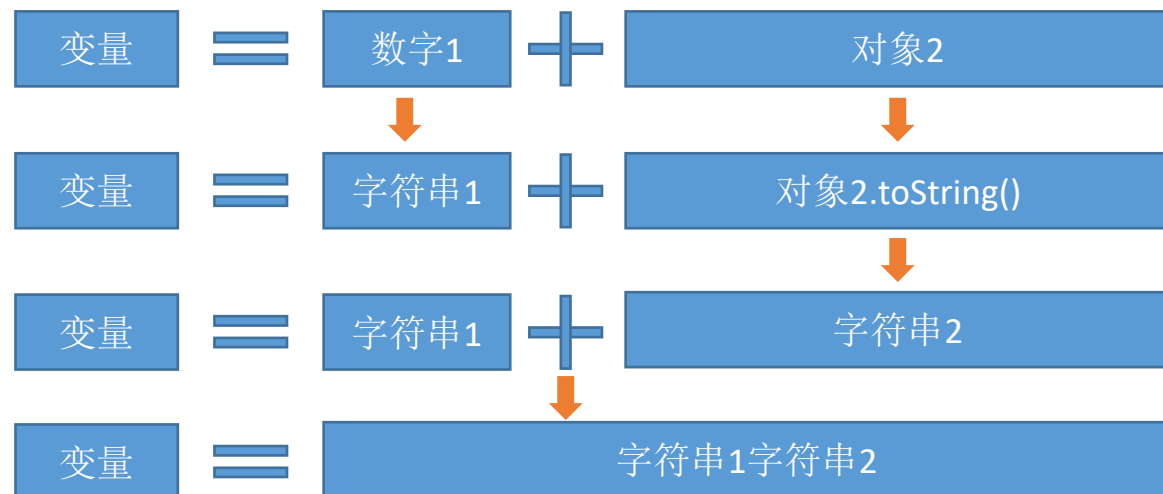
如果两个操作数都是字符串, 则将第二个操作数与第一个操作数拼接起来;

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。



## 加性操作符之加法操作符

3: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。



```
Elements Console Sources Network
top
> var a=["hello","world","1"];
< undefined
> var o={x:"hello",y:"world"};
< undefined
> var x=10;
< undefined
> var n=x+a;
< undefined
> n
< "10hello,world,1"
> var m=x+o;
< undefined
> m
< "10[object Object]"
> |
```

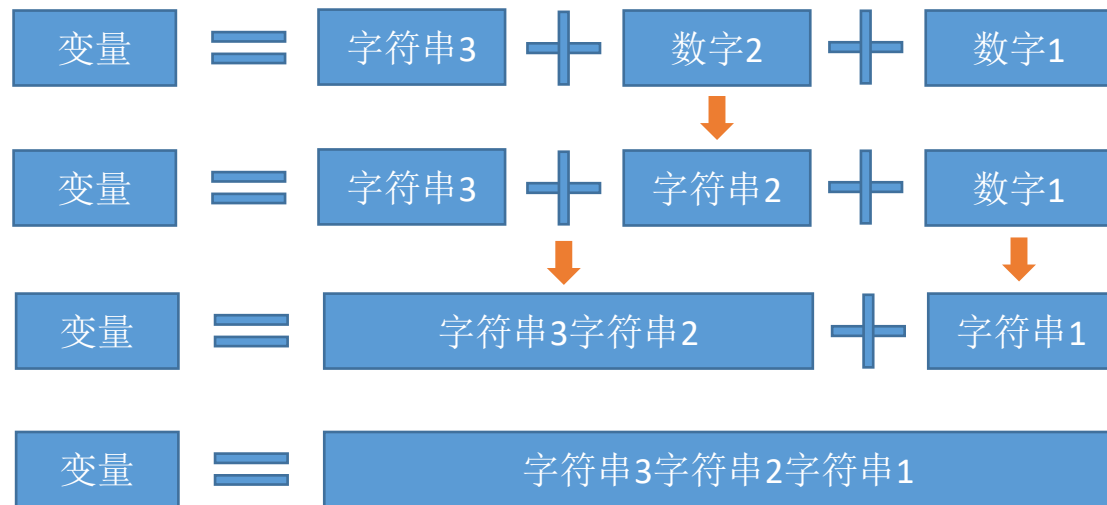
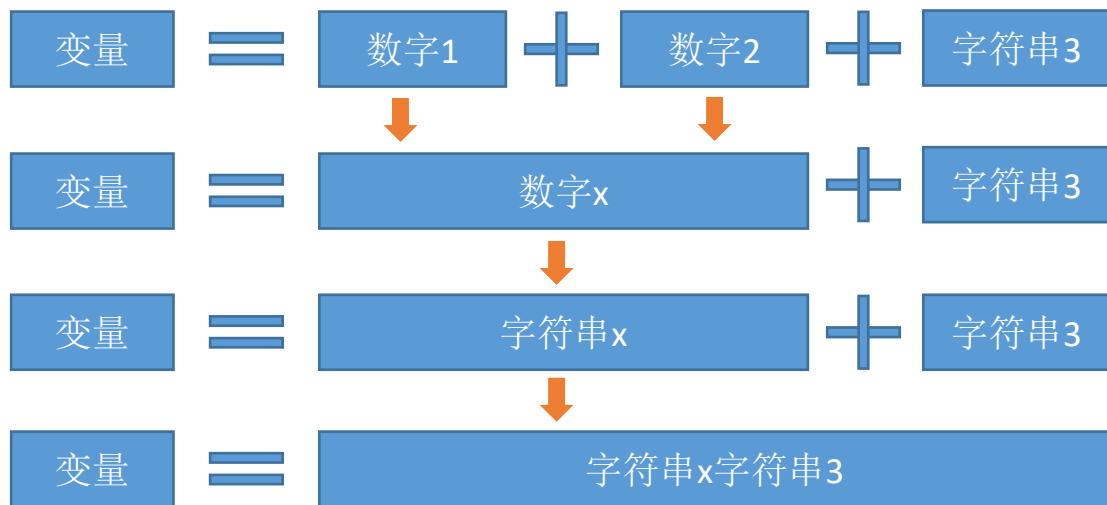
4:对于undefined 和null，则分别调用Number()函数并取得数字, 再参与计算

undefined -> NaN

null -> 0

# 加性操作符之加法操作符

每个加法操作是独立执行的



```
Elements Console Sources Network >>
top
> 1+3+"hello"
< "4hello"
>
```

```
Elements Console Sources Network >>
top
> "hello"+1+2
< "hello12"
> |
```

## 加性操作符之加法操作符

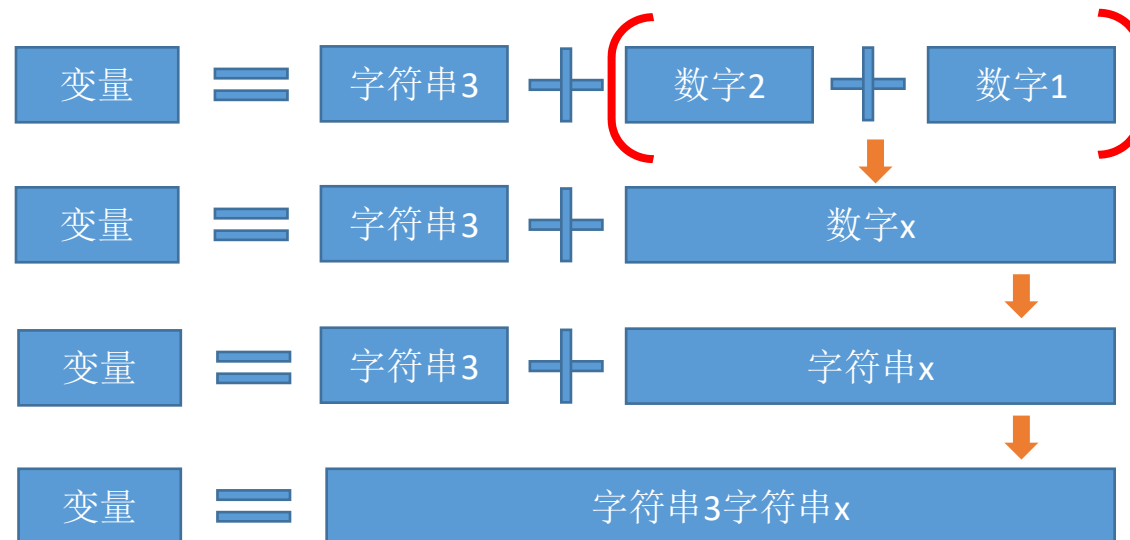
可以通过圆括号来改变计算的顺序

比如:

`x=5*6+3;`//结果是33

但是加一个圆括号

`x=5*(6+3);`//结果就是45



```
Elements Console Sources Network >>
top
Filter Default levels
> "hello" + (1+2)
< "hello3"
>
```

## 加性操作符之减法操作符

**减性** 操作符 是一个暴躁老哥!!! !!!

减法操作符在执行的时候操作数也会进行数据类型转化

但是 是把所有类型的数据**先转化为数字类型**之后在计算, 不能直接转数字的就**先转成字符串再转成数字**

减法操作符的用法如下

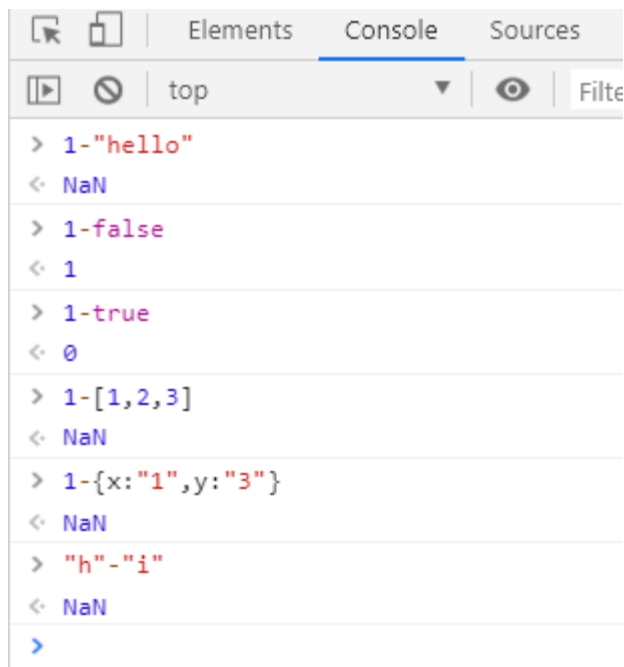
变量= 值1- 值2- .....;

值1, 值2被称为: 操作数  
- 被称为:操作符

❑ 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。



```
> 1-"hello"
< NaN

> 1-false
< 1

> 1-true
< 0

> 1-[1,2,3]
< NaN

> 1-{x:"1",y:"3"}
< NaN

> "h"-"i"
< NaN

>
```





# 布尔操作符

## 布尔操作符之与(&&)

逻辑与操作符由两个和号 (&&) 表示，有两个操作数

```
var result = true && false;
```

第一个操作数	第二个操作数	结 果
true	true	true
true	false	false
false	true	false
false	false	false

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

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；

如果第一个操作数为true，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。

# 布尔操作符之与(&&)

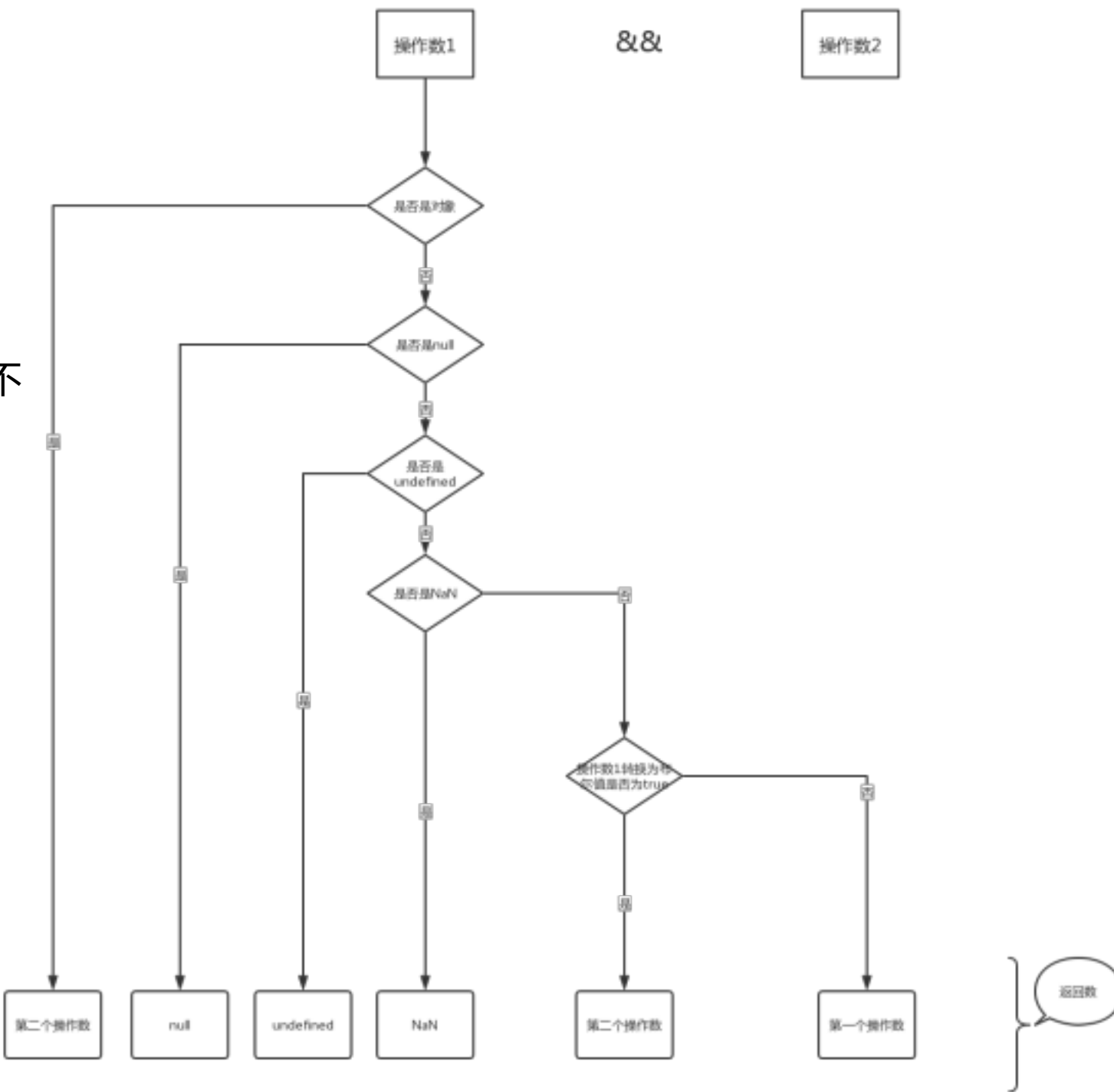
**var result = true && false;**

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

```
> var a;  
< undefined  
> var c=1&&a;  
< undefined  
> c  
< undefined  
> var d=1&&u;  
⚠ Uncaught ReferenceError: u is not defined VM193:1  
  at <anonymous>:1:8  
> |
```

不能在逻辑与操作中使用未定义的变量



## 布尔操作符之或(||)

逻辑或操作符由两个竖线符号 (||) 表示，有两个操作数

```
var result = true || false;
```

第一个操作数	第二个操作数	结 果
True	true	true
True	false	true
false	true	true
false	false	false

它遵循下列规则：

- 如果第一个操作数为true，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；

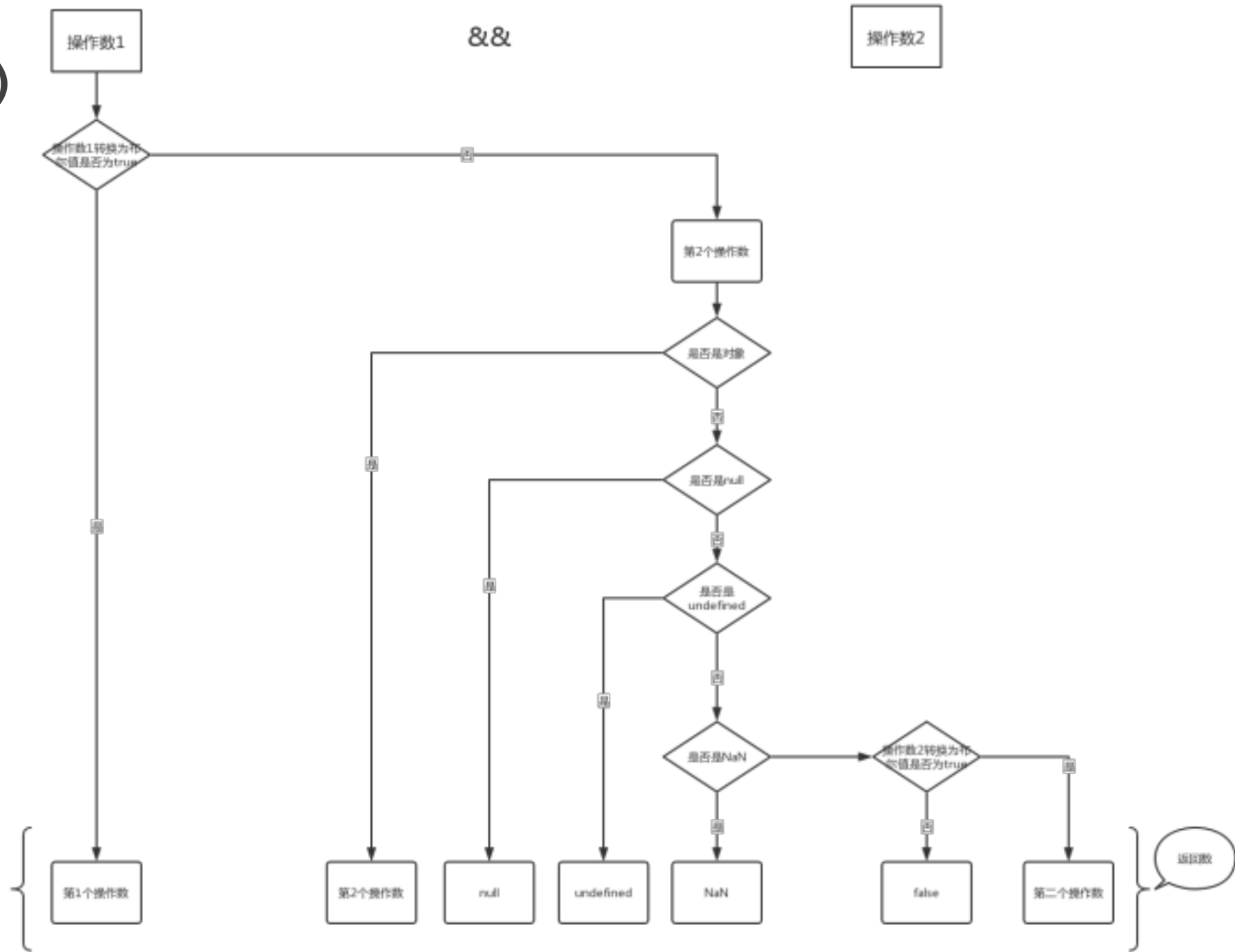
- 如果两个操作数都是NaN，则返回NaN；

- 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

# 布尔操作符之或(II)

判断模型



## 布尔操作符之或(!)

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(见数据类型转换一章)，然后再对其求反

```
> !false
< true
> !1
< false
> !0
< true
> ![]
< false
> |
```



# 一元操作符

# 前置型一元操作符

前置型递增:

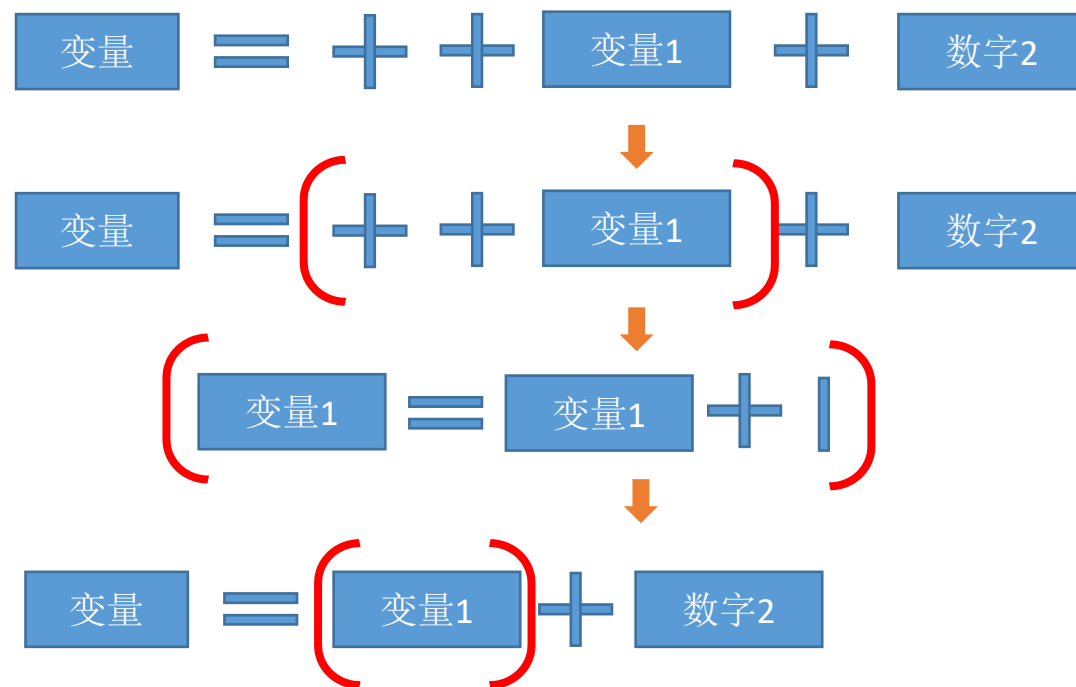
**++value;**  
等价于  
value=value+1;

前置型递减:

**--value;**  
等价于  
value=value-1;

执行前置递增和递减操作时, 先计算前置递增的数据, 然后将得到的值参与代码计算

```
> var a=22;  
< undefined  
> var b=--a+1;  
< undefined  
> b  
< 22  
> a  
< 21  
> var c=10;  
< undefined
```





# 后置型一元操作符

## 后置型递增

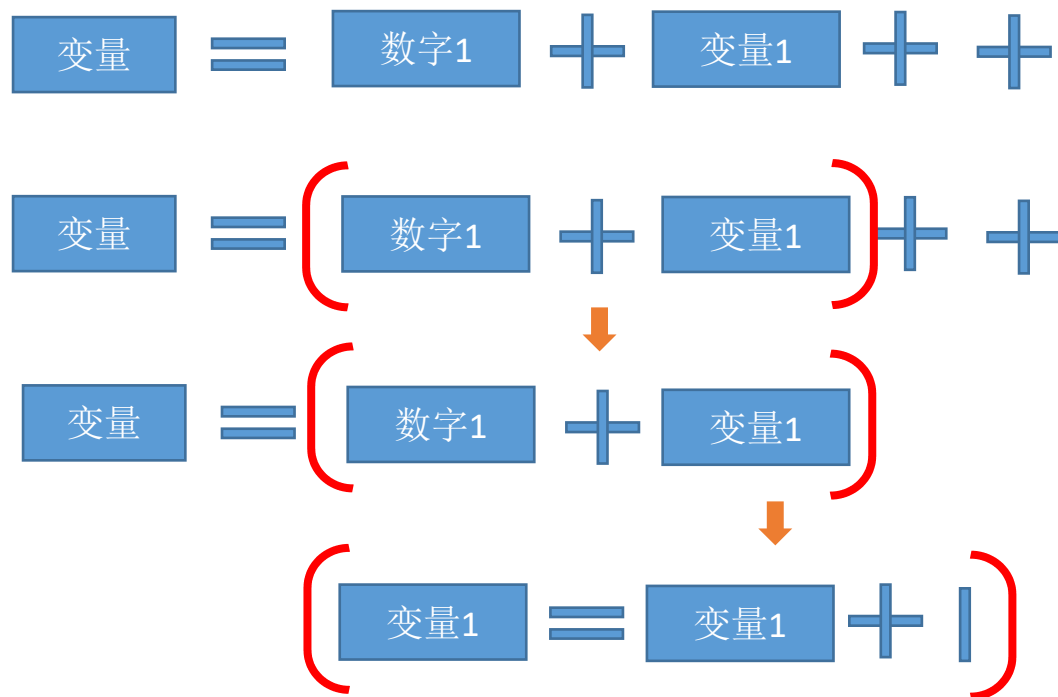
**value++;**  
等价于  
value=value+1;

## 后置型递减

**value--;**  
等价于  
value=value-1;

执行前置递增和递减操作时, 先用之前的值参与代码计算, 代码执行完之后, 再执行后置型递增

```
> var c=10;  
< undefined  
> var d=c+++1  
< undefined  
> d  
< 11  
> c  
< 11  
> var e=1+c++;  
< undefined  
> e  
< 12  
> c  
< 12  
>
```





# 乘性操作符

## 乘法 \*

如果乘性计算的某个操作数不是数值，后台会先用Number()转型函数将其转换为数值

乘法操作符由一个星号 (\*) 表示，用于计算两个数值的乘积

```
var result = 34 * 56;
```

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。

如果有一个操作数是NaN，则结果是NaN；

## 除法 /

如果乘性计算的某个操作数不是数值，后台会先用Number()转型函数将其转换为数值

除法操作符由一个斜线符号 (/) 表示

```
var result = 66 / 11;
```

```
> Infinity
< Infinity
> typeof Infinity
< "number"
> 1/0
< Infinity
> 1/-0
< -Infinity
>
```

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。

如果有一个操作数是NaN，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；

**Infinity**:表示正无穷的意思

**-Infinity**:表示负无穷的意思

## 求模 % (求余数)

如果乘性计算的某个操作数不是数值，后台会先用Number()转型函数将其转换为数值

求模（余数）操作符由一个百分号（%）表示

```
var result = 26 % 5; // 等于1
```

- ❑ 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；
- ❑ 其他的规则与除法相同



# 关系操作符

## 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(字符是有编码的)。(顺序是A->Z->a->z, 所以z>a>Z>A)

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。