运算符的定义和分类

运算符的定义

运算符：也叫操作符，是一种符号。通过运算符可以对一个或多个值进行运算，并获取运算结果。

表达式：由数字、运算符、变量的组合（组成的式子）。

表达式最终都会有一个运算结果，我们将这个结果称为表达式的**返回值**。

比如：`+`、`*`、`/`、`(` 都是**运算符**，而 `（3+5）/2` 则是**表达式**。

比如：typeof 就是运算符，可以来获得一个值的类型。它会将该值的类型以**字符串**的形式返回，返回值可以是 number、string、boolean、undefined、object。

运算符的分类

JS 中的运算符，分类如下：

- 算术运算符
- 自增/自减运算符
- 一元运算符
- 逻辑运算符
- 赋值运算符
- 比较运算符
- 三元运算符（条件运算符）

下面来逐一讲解。

算术运算符

算术运算符：用于执行两个变量或值的算术运算。

常见的算术运算符有以下几种：

运算符	描述
+	加、字符串连接
-	减
*	乘
/	除
%	获取余数（取余、取模）

求余的举例：

假设用户输入345，怎么分别得到3、4、5这三个数呢？

答案：

```
得到3的方法：345 除以100，得到3.45然后取整，得到3。即：parseInt(345/100)

得到4的方法：345 除以100，余数是45，除以10，得到4.5，取整。即：parseInt(345 % 100 / 10)

得到5的方法：345 除以10，余数就是5。即：345 % 10
```

算术运算符的运算规则

（1）先算乘除、后算加减。

（2）小括号`( )`：能够影响计算顺序，且可以嵌套。没有中括号、没有大括号，只有小括号。

（3）百分号：取余。只关心余数。

举例1：取余

```
console.log(3 % 5);
```

输出结果为3。

举例2：注意运算符的优先级

```
var a = 1 + 2 * 3 % 4 / 3;
```

结果分析：

原式 = 1 + 6 % 4 / 3 = 1 + 2 / 3 = 1.66666666666666

补充：关于算术运算符的注意事项，详见前面课程提到的“**数据类型转换**”的知识点。

浮点数运算的精度问题

浮点数值的最高精度是 17 位小数，但在进行算术计算时，会丢失精度，导致计算不够准确。比如：

```
console.log(0.1 + 0.2); // 运算结果不是 0.3，而是 0.30000000000000004

console.log(0.07 * 100); // 运算结果不是 7，而是 7.000000000000001
```

因此，**不要直接判断两个浮点数是否相等。**

自增和自减

自增 `++`

自增分成两种：`a++`和`++a`。

（1）一个变量自增以后，原变量的值会**立即**自增1。也就是说，无论是`a++`还是`++a`，都会立即使原变量的值自增1。

（2）**我们要注意的是**：`a`是变量，而`a++`和`++a`是**表达式。**

那这两种自增，有啥区别呢？区别是：`a++`和`++a`的值不同：（也就是说，表达式的值不同）

- `a++`这个表达式的值等于原变量的值（a自增前的值）。你可以这样理解：先把 a 的值赋值给表达式，然后 a 再自增。-- 先用再加

- `++a`这个表达式的值等于新值 （a自增后的值）。 你可以这样理解：a 先自增，然后再把自增后的值赋值给表达式。-- 先加再用

```
var a = 1;
console.log(a++);
console.log(++a);

var a = 1;
console.log(a++);
var b = 1;
console.log(++b);
console.log(a,b);
```

自减 `--`

原理同上。

开发时，大多使用后置的自增/自减，并且代码独占一行，例如：`num++`，或者 `num--`。

代码举例

```
var n1 = 10;
var n2 = 20;

var result = n1++;
console.log(n1); // 11
console.log(result); // 10

var result1 = ++n1;
console.log(n1); //12
console.log(result1); //12

var result2 = n2--;
console.log(n2); // 19
console.log(result2); // 20

var result3 = --n2;
console.log(n2); // 18
console.log(result3); // 18
```

一元运算符

一元运算符，只需要一个操作数。

常见的一元运算符如下。

typeof

typeof就是典型的一元运算符，因为后面只跟一个操作数。

举例如下：

```
var a = '123';
console.log(typeof a); // 打印结果：string
```

正号 +

（1）正号不会对数字产生任何影响。比如说，2 和 +2 是一样的。

（2）我们可以对一个其他的数据类型使用 +，来将其转换为number【重要的小技巧】。比如：

```
var a = true;
a = +a;   // 注意这行代码的一元运算符操作
console.log('a：' + a);   // 这里的a经过隐式转换变成了字符串，字符串拼接
console.log(typeof a);   // 这里的a是数字

console.log('-----------------');

var b = '18';
b = +b;   // 注意这行代码的一元运算符操作
console.log('b：' + b);
console.log(typeof b);
```

打印结果：

```
a：1
number
-----------------
b：18
number
```

负号 -

负号可以对数字进行取反。

逻辑运算符

逻辑运算符有三个：

- && 与（且）：两个都为真，结果才为真。and
- || 或：只要有一个是真，结果就是真。or
- ! 非：对一个布尔值进行取反。

注意：能参与逻辑运算的，都是布尔值。

连比的写法：

来看看逻辑运算符连比的写法。

举例1：

```
console.log(3 < 2 && 2 < 4);
```

输出结果为false。

举例2：（判断一个人的年龄是否在18~65岁之间）

```
const a = prompt('请输入您的年龄');

if (a >= 18 && a < 65) {
    alert('可以上班');
} else {
    alert('准备退休');
}
```

PS：上面的`a>=18 && a<= 65`千万别想当然地写成`18<= a <= 65`，没有这种语法。

非布尔值的与或运算【重要】

> 之所以重要，是因为在实际开发中，我们经常用这种代码做容错处理或者兜底处理。

非布尔值进行**与或运算**时，会先将其转换为布尔值，然后再运算，但返回结果是**原值**。比如说：

```
var result = 5 && 6; // 运算过程：true && true;
console.log('result：' + result); // 打印结果：6（也就是说最后面的那个值。）
```

上方代码可以看到，虽然运算过程为布尔值的运算，但返回结果是原值。

那么，返回结果是哪个原值呢？我们来看一下。

与运算的返回结果：（以多个非布尔值的运算为例）

- 如果第一个值为false，则执行第一条语句，并直接返回第一个值；不会再往后执行。
- 如果第一个值为true，则继续执行第二条语句，并返回第二个值（如果所有的值都为true，则返回的是最后一个值）。

或运算的返回结果：（以多个非布尔值的运算为例）

- 如果第一个值为true，则执行第一条语句，并直接返回第一个值；不会再往后执行。
- 如果第一个值为false，则继续执行第二条语句，并返回第二个值（如果所有的值都为false，则返回的是最后一个值）。

实际开发中，我们经常是这样来做「容错处理」的：

当前端成功调用一个接口后，返回的数据为 result 对象。这个时候，我们用变量 a 来接收 result 里的图片资源。通常的写法是这样的：

```
if (result.resultCode == 0) {
    var a = result && result.data && result.data.imgUrl ||
'http://www.baidu.com/20211113_01.jpg';
}
```

上方代码的意思是，获取返回结果中的`result.data.imgUrl`这个图片资源；如果返回结果中没有`result.data.imgUrl`这个字段，就用`http://www.baidu.com/20211113_01.jpg`作为**兜底**图片。这种写法，在实际开发中经常用到。

非布尔值的`!`运算

非布尔值进行**非运算**时，会先将其转换为布尔值，然后再运算，但返回结果是**布尔值**。

举例：

```
let a = 10;
a = !a

console.log(a);  // false
console.log(typeof a); // boolean
```

赋值运算符

可以将符号右侧的值赋值给符号左侧的变量。

举例：

- `=` 直接赋值。比如 `var a = 5`
- `+=`。a += 5 等价于 a = a + 5
- `-=`。a -= 5 等价于 a = a - 5
- `*=`。a *= 5 等价于 a = a * 5
- `/=`。a /= 5 等价于 a = a / 5
- `%=`。a %= 5 等价于 a = a % 5

运算符的优先级

运算符的优先级如下：（优先级从高到低）

- `.`、`[]`、`new`
- `()`
- `++`、`--`
- `!`、`~`、`+`（单目）、`-`（单目）、`typeof`、`void`、`delete`
- `%`、`*`、`/`
- `+`（双目）、`-`（双目）
- `<<`、`>>`、`>>>`
- 关系运算符：`<`、`<=`、`>`、`>=`
- `==`、`!==`、`===`、`!==`
- `&`
- `^`
- `|`
- `&&`
- `||`
- `?:`
- `=`、`+=`、`-=`、`*=`、`/=`、`%=`、`<<=`、`>>=`、`>>>=`、`&=`、`^=`、`|=`
- `,`

注意：逻辑与 `&&` 比逻辑或 `||` 的优先级更高。

备注：你在实际写代码的时候，如果不清楚哪个优先级更高，可以把括号运用上。

Unicode 编码

> 这一段中，我们来讲引申的内容：Unicode编码的使用。

各位同学可以先在网上查一下“Unicode 编码表”。

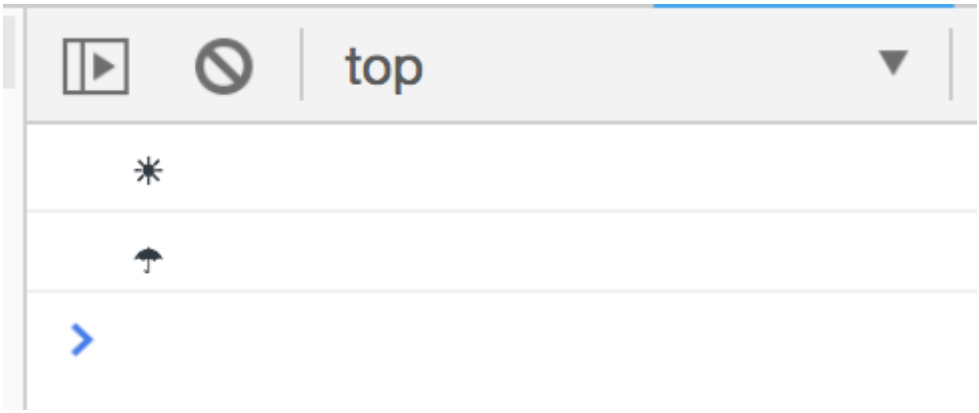
1、在字符串中可以使用转义字符输入Unicode编码。格式如下：

```
\u四位编码
```

举例如下：

```
console.log("\u2600");  // 这里的 2600 采用的是16进制
console.log("\u2602");  // 这里的 2602 采用的是16进制
```

打印结果：



事件句柄

HTML 4.0 的新特性之一是有能力使 HTML 事件触发浏览器中的动作（action），比如当用户点击某个 HTML 元素时启动一段 JavaScript。下面是一个属性列表，这些属性可插入 HTML 标签来定义事件动作，相当于是在 HTML 标签中插入了事件句柄，可以接收JS代码并执行。

属性	当以下情况发生时，出现此事件	FF	N	IE
onabort	图像加载被中断	1	3	4
onblur	元素失去焦点	1	2	3
onchange	用户改变域的内容	1	2	3
onclick	鼠标点击某个对象	1	2	3
ondblclick	鼠标双击某个对象	1	4	4
onerror	当加载文档或图像时发生某个错误	1	3	4
onfocus	元素获得焦点	1	2	3
onkeydown	某个键盘的键被按下	1	4	3
onkeypress	某个键盘的键被按下或按住	1	4	3
onkeyup	某个键盘的键被松开	1	4	3
onload	某个页面或图像被完成加载	1	2	3
onmousedown	某个鼠标按键被按下	1	4	4
onmousemove	鼠标被移动	1	6	3
onmouseout	鼠标从某元素移开	1	4	4
onmouseover	鼠标被移到某元素之上	1	2	3
onmouseup	某个鼠标按键被松开	1	4	4
onreset	重置按钮被点击	1	3	4
onresize	窗口或框架被调整尺寸	1	4	4
onselect	文本被选定	1	2	3
onsubmit	提交按钮被点击	1	2	3
onunload	用户退出页面	1	2	3