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：

- 算数运算符
- 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。

算数运算符

算术运算符：用于执行两个变量或值的算术运算。

常见的算数运算符有以下几种：

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
```

算数运算符的运算规则

（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：关于算数运算符的注意事项，详见上一篇文章里的“**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。

- `++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。

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 result1 = n1++;

console.log(n1); // 11
console.log(result1); // 10

result = ++n1;
console.log(n1); //12
console.log(result); //12

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

result2 = --n2;
console.log(n2); // 18
console.log(result2); // 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);
console.log(typeof a);

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。
- `||` 或：只要有一个是真，结果就是真。
- `!`　　非：对一个布尔值进行取反。

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18~60岁之间）

```
var a = prompt("请输入您的年龄");
alert(a>=18 && a<= 65);
```

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

注意事项

（1）能参与逻辑运算的，都是布尔值。

（2）JS中的`&&`属于**短路**的与，如果第一个值为false，则不会看第二个值。举例：

```
//第一个值为true，会检查第二个值
true && alert("看我出不出来！！");  // 可以弹出 alert 框

//第一个值为false，不会检查第二个值
false && alert("看我出不出来！！"); // 不会弹出 alert 框
```

（3）JS中的`||`属于**短路**的或，如果第一个值为true，则不会看第二个值。举例：

（4）如果对**非布尔值**进行逻辑运算，则会**先将其转换为布尔值**，然后再操作。举例：

```
var a = 10;
a = !a;

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

上面的例子，我们可以看到，对非布尔值进行`!`操作之后，返回结果为布尔值。

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：（以两个非布尔值的运算为例）

- 如果第一个值为false，则直接返回第一个值；不会再往后执行。
- 如果第一个值为true，则返回第二个值（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：（以两个非布尔值的运算为例）

- 如果第一个值为true，则直接返回第一个值；不会再往后执行。
- 如果第一个值为false，则返回第二个值（（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 || './img/404.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`./img/404.jpg`作为**兜底**图片。这种写法，在实际开发中经常用到。

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

比较运算符

比较运算符可以比较两个值之间的大小关系，如果关系成立它会返回true，如果关系不成立则返回false。

比较运算符有很多种，比如：

```
>    大于号
<    小于号
>=   大于或等于
<=   小于或等于
==   等于
===  全等于
!=   不等于
!==  不全等于
```

比较运算符，得到的结果都是布尔值：要么是true，要么是false。

举例如下：

```
var result = 5 > 10; // false
```

非数值的比较

（1）对于非数值进行比较时，会将其转换为数字然后再比较。

举例如下：

```
console.log(1 > true); //false
console.log(1 >= true); //true
console.log(1 > "0"); //true

//console.log(10 > null); //true

//任何值和NaN做任何比较都是false

console.log(10 <= "hello"); //false
console.log(true > false); //true

```

（2）特殊情况：如果符号两侧的值都是字符串时，**不会**将其转换为数字进行比较。比较两个字符串时，比较的是字符串的**Unicode编码**。【非常重要，这里是个大坑，很容易踩到】

比较字符编码时，是一位一位进行比较。如果两位一样，则比较下一位。

比如说，当你尝试去比较`"123"`和`"56"`这两个字符串时，你会发现，字符串"56"竟然比字符串"123"要大。也就是说，下面这样代码的打印结果，其实是true:（这个我们一定要注意，在日常开发中，很容易忽视）

```
// 比较两个字符串时，比较的是字符串的字符编码，所以可能会得到不可预期的结果
console.log("56" > "123");  // true
```

因此：当我们在比较两个字符串型的数字时，**一定一定要先转型**再比较大小，比如`parseInt()`。

（3）任何值和NaN做任何比较都是false。

`==`符号的强调

注意`==`这个符号，它是**判断是否等于**，而不是赋值。

（1）`==`这个符号，还可以验证字符串是否相同。例如：

```
console.log("我爱你中国" == "我爱你中国");        // 输出结果为true
```

（2）`==`这个符号并不严谨，会做隐式转换，将不同的数据类型，**转为相同类型**进行比较（大部分情况下，都是转换为数字）。例如：

```
console.log("6" == 6);      // 打印结果：true。这里的字符串"6"会先转换为数字6，然后再进行比较
console.log(true == "1");   // 打印结果：true
console.log(0 == -0);       // 打印结果：true

console.log(null == 0);   // 打印结果：false
```

（3）undefined 衍生自 null，所以这两个值做相等判断时，会返回true。

```
console.log(undefined == null);  //打印结果：true。
```

（4）NaN不和任何值相等，包括他本身。

```
console.log(NaN == NaN); //false
console.log(NaN === NaN); //false
```

问题：那如果我想判断 b的值是否为NaN，该怎么办呢？

答案：可以通过isNaN()函数来判断一个值是否是NaN。举例：

```
console.log(isNaN(b));
```

如上方代码所示，如果 b 为 NaN，则返回true；否则返回false。

`===`全等符号的强调

全等在比较时，不会做类型转换。如果要保证**绝对等于（完全等于）**，我们就要用三个等号`===`。例如：

```
    console.log("6" === 6);     //false
    console.log(6 === 6);       //true
```

上述内容分析出：

- `==`两个等号，不严谨，"6"和6是true。
- `===`三个等号，严谨，"6"和6是false。

另外还有：**`==`的反面是`!=`，`===`的反面是`!==`**。例如：

```
    console.log(3 != 8);    //true
    console.log(3 != "3");  //false，因为3=="3"是true，所以反过来就是false。
    console.log(3 !== "3"); //true，应为3==="3"是false，所以反过来是true。
```

三元运算符

三元运算符也叫条件运算符。

语法：

```
条件表达式 ? 语句1 : 语句2;
```

执行的流程：

条件运算符在执行时，首先对条件表达式进行求值：

- 如果该值为true，则执行语句1，并返回执行结果
- 如果该值为false，则执行语句2，并返回执行结果

如果条件的表达式的求值结果是一个非布尔值，会将其转换为布尔值然后再运算。

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”。

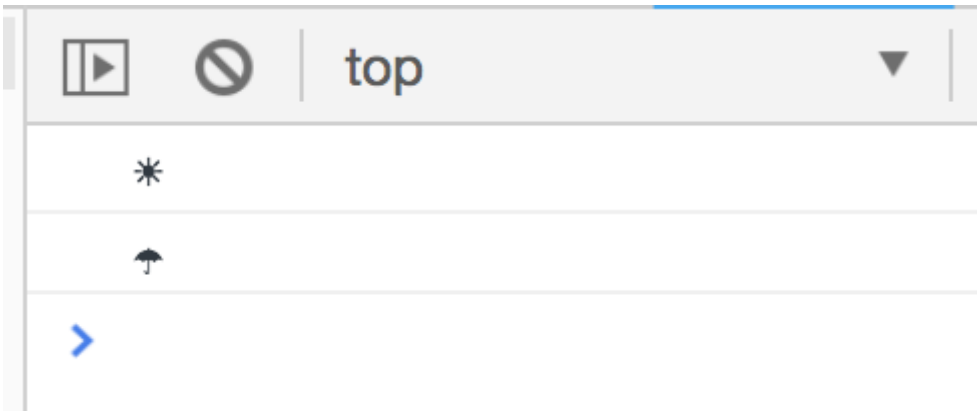
1、在字符串中可以使用转义字符输入Unicode编码。格式如下：

```
\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：



2、我们还可以在 HTML 网页中使用Unicode编码。格式如下：

```
&#四位编码;
```

PS：我们知道，Unicode编码采用的是16进制，但是，这里的编码需要使用10进制。

举例如下：

```
<h1 style="font-size: 100px;">&#9860;</h1>

```

打印结果：

demo.html ×

```
1   <!DOCTYPE html>
2   <html lang="">
3
4   <head>
5       <meta>
6       <meta>
7       <meta>
8       <title>Document</title>
9   </head>
10
11  <body>
12  </body>
13
14  <!-- 筛子这个字符的unicode编码是2684，但它是16进制的，所以我们要将其转换成10进制的9860 -->
15  <h1 style="font-size: 100px;">&#9860;</h1>
16
17  </html>
```