1 JavaScript笔记

2022年11月06开始记录

2 控制HTML

2.1 改变HTML内容

```
document.getElementById("demo").innerHTML = "Hello JavaScript";
```

2.2 改变HTML属性

```
// 改变src属性
document.getElementById('myImage').src='/i/eg_bulbon.gif'
```

2.3 改变HTML样式（CSS）

```
// 改变字体大小
document.getElementById("demo").style.fontSize = "25px";
// 隐藏元素
document.getElementById("demo").style.display="none";
// 显示元素
document.getElementById("demo").style.display="block";
```

3 输出

使用 `window.alert()` 写入警告框

使用 `document.write()` 写入 HTML 输出

使用 `innerHTML` 写入 HTML 元素

使用 `console.log()` 写入浏览器控制台

```
// 警告框显示数据
window.alert(5 + 6);

// 修改元素内容
document.getElementById("demo").innerHTML = 5 + 6;

// 注意：在 HTML 文档完全加载后使用 document.write() 将删除所有已有的 HTML
document.write(5 + 6);

// 写入浏览器控制台
console.log(5 + 6);
```

4 变量

4.1 重复声明变量

在这两条语句执行后，变量 carName 的值仍然是 "porsche"：

```
var carName = "porsche";
var carName;
```

4.2 全局作用域

```
var carName = "porsche";

// 此处的代码可以使用 carName

function myFunction() {
  // 此处的代码也可以使用 carName
}
```

4.3 函数作用域

```
// 此处的代码不可以使用 carName

function myFunction() {
  var carName = "porsche";
  // code here CAN use carName
}

// 此处的代码不可以使用 carName
```

4.4 let

ES2015 引入了两个重要的 JavaScript 新关键词：`let` 和 `const`。

这两个关键字在 JavaScript 中提供了【块作用域】（*Block Scope*）变量和常量。

在 ES2015 之前，JavaScript 只有两种类型的作用域：*全局作用域*和*函数作用域*。

4.4.1 块作用域

通过 `var` 关键词声明的变量没有块*作用域*。

在块 *{}* 内声明的变量可以从块之外进行访问。

```
{
  var x = 10;
}
// 此处可以使用 x
```

可以使用 `let` 关键词声明拥有块作用域的变量。

在块 *{}* 内声明的变量无法从块外访问。

```

{
  let x = 10;
}
// 此处不可以使用 x
```

4.4.2 重新声明变量

在块中重新声明变量不会重新声明块外的变量：

```
var x = 10;
// 此处 x 为 10
{
  let x = 6;
  // 此处 x 为 6
}
// 此处 x 为 10
```

允许在程序的任何位置使用 `var` 重新声明 JavaScript 变量：

```
var x = 10;

// 现在，x 为 10

var x = 6;

// 现在，x 为 6
```

在相同的作用域，或在相同的块中，通过 `let` 重新声明一个 `var` 变量是不允许的：

```
var x = 10;        // 允许
let x = 6;        // 不允许

{
  var x = 10;    // 允许
  let x = 6;    // 不允许
}
```

在相同的作用域，或在相同的块中，通过 `let` 重新声明一个 `let` 变量是不允许的：

```
let x = 10;        // 允许
let x = 6;        // 不允许

{
  let x = 10;    // 允许
  let x = 6;    // 不允许
}
```

在相同的作用域，或在相同的块中，通过 `var` 重新声明一个 `let` 变量是不允许的：

```
let x = 10;        // 允许
var x = 6;        // 不允许

{
  let x = 10;    // 允许
  var x = 6;    // 不允许
}
```

在不同的作用域或块中，通过 `let` 重新声明变量是允许的：

```
let x = 6;        // 允许

{
  let x = 7;    // 允许
}

{
  let x = 8;    // 允许
}
```

4.4.3 提升（Hoisting）

通过 `var` 声明的变量会*提升*到顶端。

```
// 在此处，您可以使用 carName
carName = "leisure";

var carName;
```

通过 `let` 定义的变量不会被提升到顶端。

```
// 在此处，您不可以使用 carName
carName = "leisure";

let carName;
```

4.5 const

通过 `const` 定义的变量与 `let` 变量类似，但不能重新赋值。

`const` 变量必须在声明时赋值。

4.5.1 不可更改原始值

```
const PI = 3.141592653589793;
PI = 3.14;      // 会出错
```

4.5.2 可更改常量对象

```
// 您可以创建 const 对象：
const car = {type:"porsche", model:"911", color:"Black"};

// 您可以更改属性：
car.color = "White";

// 您可以添加属性：
car.owner = "Bill";
```

4.5.3 常量数组可以更改

```
// 您可以创建常量数组：
const cars = ["Audi", "BMW", "porsche"];

// 您可以更改元素：
cars[0] = "Honda";

// 您可以添加元素：
cars.push("Volvo");
```

4.5.4 提升（Hoisting）

通过 `var` 声明的变量会*提升*到顶端。

```
// 在此处，您可以使用 carName
carName = "leisure";

var carName;
```

通过 `const` 定义的变量不会被提升到顶端。

```
// 在此处，您不可以使用 carName
carName = "leisure";

const carName;
```

4.6 window

在 HTML 中，全局作用域是 window。所有全局变量均属于 window 对象。

除非有意为之，否则请勿创建全局变量。

您的全局变量（或函数）能够覆盖 window 变量（或函数）。

任何函数，包括 window 对象，能够覆盖您的全局变量和函数。

```
var carName = "porsche";

// 此处的代码能够使用 window.carName
window.carName;
```

5 数据类型

字符串值，数值，布尔值，数组，对象。

```
// 数字
var length = 7;
// 字符串
var lastName = "Gates";
 // 数组
var cars = ["Porsche", "Volvo", "BMW"];
// 对象
var x = {firstName:"Bill", lastName:"Gates"};
// 布尔值
var isRight = true;
```

5.1 typeof 运算符

```
typeof false; // boolean
typeof ""; // string
typeof 0; // number
typeof NaN; // number
typeOf Infinity; // number
typeof undefined; // undefined
typeof x; // undefined
typeof null; // object
typeof {name:'Bill', age:62}; // object
typeof [1,2,3,4]; // object
typeof function myFunc(){}; //function
```

6 函数

6.1 函数定义

```
function name(p1, p2, p3) {
    // 要执行的代码
}
```

6.2 () 运算符调用函数

`toCelsius` 引用的是函数对象，而 `toCelsius()` 引用的是函数结果。

访问没有 () 的函数将返回函数定义。

```
function toCelsius(fahrenheit) {
    return (5/9) * (fahrenheit-32);
}

document.getElementById("demo").innerHTML = toCelsius;
// 结果：function toCelsius(f) { return (5/9) * (f-32); }
```

7 对象

7.1 对象的定义

```
var person = {
  firstName: "Bill",
  lastName: "Gates",
  id: 678,
  fullName: function() {
    return this.firstName + " " + this.lastName;
  }
};
```

7.2 访问对象属性

```
var person = {
    firstName:"Bill",
    lastName:"Gates",
    age:50,
    eyeColor:"blue"
};

// 方式一
person.lastName;

// 方式二
person["lastName"];
```

7.3 添加新属性

```
var person = {
    firstName:"Bill",
    lastName:"Gates",
    age:50,
    eyeColor:"blue"
};

// 直接赋值
person.nationality = "English";
```

7.4 删除属性

```
var person = {
    firstName:"Bill",
    lastName:"Gates",
    age:50,
    eyeColor:"blue"
};

// 删除属性
delete person.eyeColor;
delete person["eyeColor"];
```

7.5 显示对象

```
const person = {
  name: "Bill",
  age: 19,
  city: "Seattle"
};

// 使用 Object.values()
// 通过使用 Object.values()，任何 JavaScript 对象都可以被转换为数组
const myArray = Object.values(person);// Bill,19,Seattle
```

7.6 Getter 和 Setter

```
var person = {
  firstName: "Bill",
  lastName : "Gates",
  language : "en",
  get getLang() {
    return this.language;
  },
  set setLang(lang) {
    this.language = lang;
  }
};

person.lang;
person.lang = "zh";
```

7.7 为对象添加方法

```
var person = {
  firstName: "Bill",
  lastName : "Gates",
  language : "en"
}

person.name = function () {
    return this.firstName + " " + this.lastName;
};
```

7.8 对象构造器

`this` 关键词

在构造器函数中，`this` 是没有值的。它是新对象的替代物。 当一个新对象被创建时，this 的值会成为这个新对象。

请注意 `this` 并不是变量。它是关键词。您无法改变 this 的值。

```
function Person(first, last, age, eye) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eye;
    this.name = function() {
        return this.firstName + " " + this.lastName;
    };
}

// 通过 new 关键词调用构造器函数可以创建相同类型的对象
var myFather = new Person("Bill", "Gates", 62, "blue");
var myMother = new Person("Steve", "Jobs", 56, "green");
```

内建 JavaScript 构造器。

JavaScript 提供用于原始对象的构造器。

`Math()` 对象不再此列。Math 是全局对象。`new` 关键词不可用于 Math。

```
var x1 = new Object();    // 一个新的 Object 对象
var x2 = new String();    // 一个新的 String 对象
var x3 = new Number();    // 一个新的 Number 对象
var x4 = new Boolean();   // 一个新的 Boolean 对象
var x5 = new Array();     // 一个新的 Array 对象
var x6 = new RegExp();    // 一个新的 RegExp 对象
var x7 = new Function();  // 一个新的 Function 对象
var x8 = new Date();      // 一个新的 Date 对象
```

正如以上所见，JavaScript 提供原始数据类型字符串、数字和布尔的对象版本。但是并无理由创建复杂的对象。原始值快得多！

请使用对象字面量 `{}` 代替 `new Object()`。

请使用字符串字面量 `""` 代替 `new String()`。

请使用数值字面量代替 `Number()`。

请使用布尔字面量代替 `new Boolean()`。

请使用数组字面量 `[]` 代替 `new Array()`。

请使用模式字面量代替 `new RexExp()`。

请使用函数表达式 `() {}` 代替 `new Function()`。

```
var x1 = {};            // 新对象
var x2 = "";            // 新的原始字符串
var x3 = 0;             // 新的原始数值
var x4 = false;         // 新的原始逻辑值
var x5 = [];            // 新的数组对象
var x6 = /()/           // 新的正则表达式对象
var x7 = function(){};  // 新的函数对象
```

7.9 原型对象

所有 JavaScript 对象都从原型继承属性和方法。

日期对象继承自 Date.prototype。数组对象继承自 Array.prototype。Person 对象继承自 Person.prototype。

Object.prototype 位于原型继承链的顶端：

日期对象、数组对象和 Person 对象都继承自 Object.prototype。

JavaScript prototype 属性允许您为对象构造器添加新属性。

```
function Person(first, last, age, eyecolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eyecolor;
}
Person.prototype.nationality = "English";
```

JavaScript prototype 属性也允许您为对象构造器添加新方法。

```
function Person(first, last, age, eyecolor) {
    this.firstName = first;
    this.lastName = last;
    this.age = age;
    this.eyeColor = eyecolor;
}
Person.prototype.name = function() {
    return this.firstName + " " + this.lastName;
};
```

7.10 对象控制

ECMAScript 5 (2009) 向 JavaScript 添加了许多新的对象方法。

管理对象

```
// 以现有对象为原型创建对象
Object.create();

// 添加或更改对象属性
Object.defineProperty(object, property, descriptor);

// 添加或更改对象属性
Object.defineProperties(object, descriptors);

// 访问属性
Object.getOwnPropertyDescriptor(object, property);

// 以数组返回所有属性
Object.getOwnPropertyNames(object);

// 访问原型
Object.getPrototypeOf(object);

// 以数组返回可枚举属性
Object.keys(object);
```

保护对象

```
// 防止向对象添加属性
Object.preventExtensions(object);

// 如果属性可以添加到对象，则返回 true
Object.isExtensible(object);

// 防止更改对象属性（不是值）
Object.seal(object);

// 如果对象被密封，则返回 true
Object.isSealed(object);
```

```
// 防止对对象进行任何更改
Object.freeze(object);

// 如果对象被冻结，则返回 true
Object.isFrozen(object);
```

ES5 允许更改以下属性元数据

```
writable : true      // 属性值可更改
enumerable : true    // 属性可枚举
configurable : true  // 属性可重新配置

writable : false     // 属性值不可更改
enumerable : false   // 属性不可枚举
configurable : false // 属性不可重新配置
```

8 事件

```
<p id="demo"></p>
<button onclick="displayDate()">时间是？</button>
<script>
function displayDate() {
    document.getElementById("demo").innerHTML = Date();
}
</script>
```

常见的事件

事件	描述
onchange	HTML 元素已被改变
onclick	用户点击了 HTML 元素
onmouseover	用户把鼠标移动到 HTML 元素上
onmouseout	用户把鼠标移开 HTML 元素
onkeydown	用户按下键盘按键
onload	浏览器已经完成页面加载

9 字符串

9.1 字符串方法

```
var str = "leisure";
// 字符串长度
str.length;

// 首次初见的索引值
str.indexOf("s"); // 3
// 作为检索起始位置的第二个参数
str.indexOf("s", 2);

// 最后出现的索引值
str.lastIndexOf("e"); // 6
// 从尾到头，从下标5检索到字符串的起点
str.lastIndexOf("e", 5);

// search() 方法搜索特定值的字符串，并返回匹配的位置
// 支持正则表达式
str.search("e");

// 提取字符串[1, 5)
str.slice(1,5);
// 提取字符串(1, 5]
str.slice(-5,-1); // isur
// 提取字符串[1, str.length)
str.slice(1);

// 提取字符串[1, 5)
str.substring(1,5);
// 提取字符串[1, str.length)
str.substring(1);

// 提取字符串[1, 5)
str.substr(1,5);
// 提取字符串[1, str.length)
str.substr(1);
// 提取字符串(1, str.length)
str.substr(-5); //isure

// replace() 方法用另一个值替换在字符串中指定的值
// 默认地，replace() 只替换首个匹配
// 对大小写敏感
str.replace("e", "t");
// 如需执行大小写不敏感的替换，请使用正则表达式 /i（大小写不敏感）
str.replace(/E/i, "t");
// 如需替换所有匹配，请使用正则表达式的 g 标志（用于全局搜索）
str.replace(/e/g, "t");

// 字符串转换为大写
str.toUpperCase();
// 字符串转换为小写
str.toLowerCase();

// 连接两个或多个字符串
str.concat(" ", "coming");// leisure coming

// 删除字符串两端的空白符
str.trim();
```

```
// 返回指定下标的字符串
str.charAt(0); // l
// 返回指定下标的字符串 unicode 编码
str.charCodeAt(0);

// 可以通过 split() 将字符串转换为数组
str.split(""); // [l,e,i,s,u,r,e]

// match() 方法根据正则表达式在字符串中搜索匹配项，并将匹配项作为 Array 对象返回
// g修饰符全局搜索；i修饰符不区分大小写
str.match(/e/gi); // [e,e]

// 如果字符串包含指定值，includes() 方法返回 true
str.includes("e"); // ture
// 检查字符串是否包含 "e"，从位置 3 开始搜索
str.includes("e", 3); // ture

// 如果字符串以指定值开头，则 startsWith() 方法返回 true，否则返回 false
str.startsWith("e"); // false
// 3为开始的位置
str.startsWith("e", 3); // false

// 如果字符串以指定值结尾，则 endsWith() 方法返回 true，否则返回 false
str.endsWith("e"); // true
// 3为要搜索的长度
str.endsWith("e", 3); // false
```

9.2 字符串模板

模板字面量使用反引号【``】； 不是双引号【""】和单引号【''】来定义字符串。

```
// 通过使用模板字面量，您可以在字符串中同时使用单引号和双引号
let str1 = `He's often called "Johnny"`;

// 模板字面量允许多行字符串
let str2 =
    `The quick
    brown fox
    jumps over
    the lazy dog`;

// 插值
let firstName = "Bill";
let lastName = "Gates";
let str3 = `Welcome ${firstName}, ${lastName}!`;
```

10 数字

JavaScript 只有一种数值类型。

JavaScript 数值始终是 64 位的浮点数。

```
var x = 3.14;
var x = 123e5; // 12300000
var x = 123e+5; // 12300000
var x = 123e-5; // 0.00123
```

10.1 运算

加

```
var x = "100";
var y = "10";
var z = x + y; //10010
```

减

```
var x = "100";
var y = "10";
var z = x - y; // 90
```

乘

```
var x = "100";
var y = "10";
var z = x * y; // 1000
```

除

```
var x = "100";
var y = "10";
var z = x / y; // 10
```

10.2 NaN

`NaN` 属于 JavaScript 保留词，指示某个数不是合法数。

```
// 尝试用一个非数字字符串进行除法会得到 NaN（Not a Number）
var x = 100 / "Apple";  // NaN
isNaN(x); // true
```

10.3 十六进制

```
var myNumber = 128;
myNumber.toString(16); // 80
myNumber.toString(8); // 200
myNumber.toString(2); // 10000000
```

10.4 数字方法

```
var x = 123;

// 转字符串
x.toString(); // 123

// toExponential() 返回字符串值，它包含已被【四舍五入】并使用指数计数法的数字
// 参数定义小数点后的字符数：
x.toExponential(2); // 123.00e+0

// toFixed() 返回字符串值，它包含了指定位数小数的数字【四舍五入】
x.toFixed(2); // 123.00

// toPrecision() 返回字符串值，它包含了指定长度的数字【四舍五入】
x.toPrecision(2); // 1.2e+2
(9.656).toPrecision(2); // 9.7

// valueOf() 以数值返回数值
x.valueof(); // 123


// 全局方法可用于所有 JavaScript 数据类型
// Number() 可用于把 JavaScript 变量转换为数值
Number(true); // 1
Number(new Date()); // 1404568027739
Number(new Date("2019-04-15")); // 1506729600000
Number(10); // 10
Number("10 20"); // NaN

// parseInt() 解析一段字符串并返回数值。允许空格。只返回首个数字
parseInt("10"); // 10
parseInt("10.33"); // 10
parseInt("10 20 30"); // 10
parseInt("10 years"); // 10
parseInt("years 10"); // NaN

//parseFloat() 解析一段字符串并返回数值。允许空格。只返回首个数字
parseFloat("10"); // 10
parseFloat("10.33"); // 10.33
parseFloat("10 20 30"); // 10
parseFloat("10 years"); // 10
parseFloat("years 10"); // NaN

// 返回 JavaScript 中可能的最大数
Number.MAX_VALUE;
// 返回 JavaScript 中可能的最小数
Number.MIN_VALUE;
// 表示负的无穷大（溢出返回）
```

```
Number.NEGATIVE_INFINITY;
// 表示无穷大（溢出返回）
Number.POSITIVE_INFINITY;
// 表示非数字值（"Not-a-Number"）
Number.NaN;
```

11 数组

JavaScript 变量可以是对象。数组是特殊类型的对象。

正因如此，您可以在相同数组中存放不同类型的变量。

您可以在数组保存对象。您可以在数组中保存函数。你甚至可以在数组中保存数组。

```
var myArray = [
    Date.now,
    function getName() {
        return "leisure";
    },
    11,
    "leisure"
];
```

11.1 数组的属性和方法

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];

// 数组长度
fruits.length;

// 向数组添加新元素的最佳方法是使用 push() 方法
fruits.push("Lemon");

// 也可以使用 length 属性向数组添加新元素
fruits[fruits.length] = "Lemon";

// 添加最高索引的元素可在数组中创建未定义的“洞”
var fruits = ["Banana", "Orange"];
fruits[3] = "Lemon"; //  ["Banana", "Orange", undefined, "Lemon"]

// 如何识别数组
Array.isArray(fruits); // true
fruits instanceof Array // true

// 把数组转换为字符串
fruits.toString(); // Banana,Orange,Apple,Mango

// join() 方法也可将所有数组元素结合为一个字符串
fruits.join("-"); // Banana-Orange-Apple-Mango

// pop() 方法从数组中删除最后一个元素
```

```
// pop() 方法返回“被弹出”的值
fruits.pop(); // Mango

// push() 方法（在数组结尾处）向数组添加一个新的元素
// push() 方法返回新数组的长度
fruits.push("Kiwi"); // 5

// shift() 方法会删除首个数组元素，并把所有其他元素“位移”到更低的索引
// shift() 方法返回被“位移出”的字符串
fruits.shift(); // Banana

// unshift() 方法（在开头）向数组添加新元素，并“反向位移”旧元素
// unshift() 方法返回新数组的长度
fruits.shift("Lemon"); // 5

// 既然 JavaScript 数组属于对象，其中的元素就可以使用 JavaScript delete 运算符来删除
delete fruits[0]; // 把 fruits 中的首个元素改为 undefined


// splice() 方法可用于向数组添加新项
// 第一个参数（2）定义了应添加新元素的位置（拼接）
// 第二个参数（0）定义应删除多少元素
// 其余参数（“Lemon”，“Kiwi”）定义要添加的新元素
// splice() 方法返回一个包含已删除项的数组
fruits.splice(2, 2, "Lemon", "Kiwi");
// 使用 splice() 来删除元素
fruits.splice(0, 1);

// concat() 方法通过合并现有数组来创建一个新数组
fruits.concat(["Lemon", "Kiwi"]); //["Banana", "Orange", "Apple",
"Mango","Lemon", "Kiwi"]
// 合并多个数组
fruits.concat(["Lemon", "Kiwi"], ["Cecilie", "Lone"]);

// slice() 方法用数组的某个片段切出新数组
// slice() 方法创建新数组。它不会从源数组中删除任何元素
fruits.slice(3); // ["Mango"]
// slice() 可接受两个参数，比如 (1, 3)
fruits.slice(1, 3); // ["Orange", "Apple"]

// sort() 方法以字母顺序对数组进行排序
fruits.sort();
// reverse() 方法反转数组中的元素
fruits.reverse();

var points = [40, 100, 1, 5, 25, 10];

// 【比值函数】的目的是定义另一种排序顺序
// 当 sort() 函数比较两个值时，会将值发送到比较函数，并根据所返回的值（负、零或正值）对这些值
进行排序。
// 以随机顺序排序数组
points.sort(function(a, b){return 0.5 - Math.random()});

// 升序排序
points.sort(function(a, b){return a - b});

// 降序排序
points.sort(function(a, b){return b - a});
```

```
// 使用 Math.max.apply 来查找数组中的最高值
Math.max.apply([1, 2, 3]);
Math.max(1, 2, 3);

// 使用 Math.min.apply 来查找数组中的最低值
Math.min.apply([1, 2, 3]);
Math.min(1, 2, 3);

// 即使对象拥有不同数据类型的属性，sort() 方法仍可用于对数组进行排序
// 解决方法是通过比较函数来对比属性值
var cars = [
    {type:"Volvo", year:2016},
    {type:"Saab", year:2001},
    {type:"BMW", year:2010}
];
// 比较数字
cars.sort(function(a, b){return a.year - b.year});
// 比较字符串
cars.sort(function(a, b){
      var x = a.type.toLowerCase();
      var y = b.type.toLowerCase();
      if (x < y) {return -1;}
      if (x > y) {return 1;}
      return 0;
});

var numbers = [45, 4, 9, 16, 25];
// forEach() 方法为每个数组元素调用一次函数（回调函数）
numbers.forEach(function myFunction(value, index, array){
    console.log(value,index,array);
});
numbers.forEach(function myFunction(value){
   console.log(value);
});

// map() 方法通过对每个数组元素执行函数来创建新数组
// map() 方法不会对没有值的数组元素执行函数
// map() 方法不会更改原始数组
numbers.map(function myFunction(value, index, array){
     return value * 2;
});
numbers.map(function myFunction(value){
     return value * 2;
});

// filter() 方法创建一个包含通过测试的数组元素的新数组
numbers.filter(function myFunction(value, index, array){
      return value > 18;
});
numbers.filter(function myFunction(value){
      return value > 18;
});

// reduce() 方法在每个数组元素上运行函数，以生成（减少它）单个值
// reduce() 方法在数组中从左到右工作。另请参阅 reduceRight()
// reduce() 方法不会减少原始数组
numbers.reduce(function myFunction(total,value,index,array){
```

```
      return total + value;
});
numbers.reduce(function myFunction(total,value){
      return total + value;
});

// reduce() 方法能够接受一个初始值
numbers.reduce(function myFunction(total,value){
      return total + value;
}, 100);


// reduceRight() 方法在每个数组元素上运行函数，以生成（减少它）单个值
// reduceRight() 方法在数组中从右到左工作。另请参阅 reduce()
// reduceRight() 方法不会减少原始数组。
numbers.reduceRight(function myFunction(total,value,index,array){
      return total + value;
});
numbers.reduceRight(function myFunction(total,value){
      return total + value;
});

// reduceRight() 方法能够接受一个初始值
numbers.reduceRight(function myFunction(total,value){
      return total + value;
}, 100);


// every() 方法检查所有数组值是否通过测试
numbers.every(function myFunction(value, index, array){
      return value > 18;
}); // false
numbers.every(function myFunction(value){
      return value > 18;
}); // false

// some() 方法检查某些数组值是否通过了测试
numbers.some(function myFunction(value, index, array){
      return value > 18;
}); // true
numbers.some(function myFunction(value){
      return value > 18;
}); // true

var fruits = ["Apple", "Orange", "Apple", "Mango"];
// indexOf() 方法在数组中搜索元素值并返回其位置
fruits.indexOf("Apple"); // 0
fruits.indexOf("Apple", 0); // 0

// lastIndexOf()从数组结尾开始搜索
fruits.lastIndexOf("Apple"); // 0
fruits.lastIndexOf("Apple", 3); // 0

var numbers = [45, 4, 9, 16, 25];
// find() 方法返回通过测试函数的第一个数组元素的值
numbers.find(function myFunction(value, index, array){
      return value > 18;
}); // 45
```

```
numbers.find(function myFunction(value){
      return value > 18;
}); // 45


// findIndex() 方法返回通过测试函数的第一个数组元素的索引
numbers.findIndex(function myFunction(value, index, array){
      return value > 18;
}); // 0
numbers.findIndex(function myFunction(value){
      return value > 18;
}); // 0
```

12 日期

```
new Date();
new Date(year,month,day,hours,minutes,seconds,milliseconds);
new Date(milliseconds);
new Date(dateString);

new Date().toString();
new Date().toUTCString();
new Date().toDateString();
```

12.1 日期格式

```
ISO 日期: YYYY-MM-DDTHH:MM:SS
短日期: MM/DD/YYYY
长日期: MMM DD YYYY
```

12.2 日期获取方法

常用的方法

方法名	描述
getDate()	以数值返回天（1-31）
getDay()	以数值获取周名（0-6）
getFullYear()	获取四位的年（yyyy）
getHours()	获取小时（0-23）
getMilliseconds()	获取毫秒（0-999）
getMinutes()	获取分（0-59）
getMonth()	获取月（0-11）
getSeconds()	获取秒（0-59）
getTime()	获取时间（从 1970 年 1 月 1 日至今）

```
var d = new Date();
d.getTime();
d.getFullYear();
d.getMonth();
d.getMonth();
d.getDate();
d.getHours();
d.getMinutes();
d.getSeconds();
d.getMilliseconds();
d.getDay();
d.getDay();
```

12.3 日期设置方法

常用的方法

方法名	描述
setDate()	以数值（1-31）设置日
setFullYear()	设置年（可选月和日）
setHours()	设置小时（0-23）
setMilliseconds()	设置毫秒（0-999）
setMinutes()	设置分（0-59）
setMonth()	设置月（0-11）
setSeconds()	设置秒（0-59）
setTime()	设置时间（从 1970 年 1 月 1 日至今的毫秒数）

13 数学

13.1 属性

```
Math.E          // 返回欧拉指数（Euler's number）
Math.PI         // 返回圆周率（PI）
Math.SQRT2      // 返回 2 的平方根
Math.SQRT1_2    // 返回 1/2 的平方根
Math.LN2        // 返回 2 的自然对数
Math.LN10       // 返回 10 的自然对数
Math.LOG2E      // 返回以 2 为底的 e 的对数（约等于 1.414）
Math.LOG10E     // 返回以 10 为底的 e 的对数（约等于 0.434）
```

13.2 Math方法

https://www.w3school.com.cn/js/js_math.asp

方法	描述
abs(x)	返回 x 的绝对值
acos(x)	返回 x 的反余弦值，以弧度计
asin(x)	返回 x 的反正弦值，以弧度计
atan(x)	以介于 -PI/2 与 PI/2 弧度之间的数值来返回 x 的反正切值。
atan2(y,x)	返回从 x 轴到点 (x,y) 的角度
ceil(x)	对 x 进行上舍入
cos(x)	返回 x 的余弦
exp(x)	返回 Ex 的值
floor(x)	对 x 进行下舍入
log(x)	返回 x 的自然对数（底为e）
max(x,y,z,...,n)	返回最高值
min(x,y,z,...,n)	返回最低值
pow(x,y)	返回 x 的 y 次幂
random()	返回 0 ~ 1 之间的随机数
round(x)	把 x 四舍五入为最接近的整数
sin(x)	返回 x（x 以角度计）的正弦
sqrt(x)	返回 x 的平方根
tan(x)	返回角的正切

13.3 随机数

`Math.random()` 返回 0（包括） 至 1（不包括） 之间的随机数。

`Math.random()` 与 `Math.floor()` 一起使用用于返回随机整数。

```
// 返回 0 至 9 之间的数
Math.floor(Math.random() * 10);

// [min, max)
function getRndInteger(min, max) {
    return Math.floor(Math.random() * (max - min) ) + min;
}

// [min, max]
function getRndInteger(min, max) {
    return Math.floor(Math.random() * (max - min + 1) ) + min;
}
```

14 循环

`For/In`

```
// 语句遍历对象的属性。
var person = {fname:"Bill", lname:"Gates", age:62};
for (let key in person) {
    console.log(person[key]);
}

// 语句也可以遍历数组的属性
const numbers = [45, 4, 9, 16, 25];
for (let index in numbers) {
   console.log(person[index]);
}
```

`for/of`语句循环遍历可迭代对象的值如：数组、字符串、映射、节点列表等 。

```
// 遍历数组
const cars = ["BMW", "Volvo", "Mini"];
for (let x of cars) {
  console.log(x);
}

// 遍历字符串
let language = "JavaScript";
for (let x of language) {
    console.log(x);
}
```

15 constructor属性

`constructor` 属性返回所有 JavaScript 变量的构造函数。

```
// function String()  {[native code]}
"Bill".constructor;

// function Number()  {[native code]}
(3.14).constructor;

// function Boolean() {[native code]}
false.constructor;

// function Array()   {[native code]}
[1,2,3,4].constructor;

// function Object()  {[native code]}
{name:'Bill',age:19}.constructor;
```

```
// function Date()    {[native code]}
new Date().constructor;

// function Function(){[native code]}
function () {}.constructor;
```

16 正则表达式

在 JavaScript 中，RegExp 对象是带有预定义属性和方法的正则表达式对象。

```
// test() 是一个正则表达式方法
// 它通过模式来搜索字符串，然后根据结果返回 true 或 false
/e/.test("leisure"); // true

// exec() 方法是一个正则表达式方法
// 它通过指定的模式（pattern）搜索字符串，并返回已找到的文本
// 如果未找到匹配，则返回 null
/e/.exec("leisure"); // e
```

17 异常

```
try {
     // 供测试的代码块
}
 catch(err) {
     // 处理错误的代码块
}
finally {
     // 无论结果如何都执行的代码块
}
```

throw 语句

`throw` 语句允许您创建自定义错误。

从技术上讲您能够*抛出异常（抛出错误）*。

异常可以是 JavaScript 字符串、数字、布尔或对象。

```
throw "Too big";    // 抛出文本
throw 500;          //抛出数字
```

当发生错误时，*JavaScript 将抛出异常。

JavaScript 实际上会创建带有两个属性的 *Error 对象*：`name` 和 `message`。

属性	描述
name	设置或返回错误名
message	设置或返回错误消息（一条字符串）

error 的 name 属性可返回六个不同的值：

错误名	描述
EvalError	已在 eval() 函数中发生的错误
RangeError	已发生超出数字范围的错误
ReferenceError	已发生非法引用
SyntaxError	已发生语法错误
TypeError	已发生类型错误
URIError	在 encodeURI() 中已发生的错误

18 Hoisting

Hoisting 是 JavaScript 将所有声明提升到当前作用域顶部的默认行为（提升到当前脚本或当前函数的顶部）。

JavaScript 只【提升声明】，而非初始化。

19 严格模式

在脚本开头进行声明，拥有全局作用域（脚本中的所有代码均以严格模式来执行）。

```
"use strict";

function myFunction() {
     y = 3.14;   // 这会引发错误，因为 y 尚未声明
}
```

在函数中声明严格模式，拥有局部作用域（只有函数中的代码以严格模式执行）。

```
x = 3.14;       // 这不会引发错误

function  myFunction() {
    "use strict";
     y = 3.14;   // 这会引发错误
}
```

严格模式执行其他规则：https://www.w3school.com.cn/js/js_strict.asp

20 this

JavaScript `this` 关键词指的是它所属的对象。

它拥有不同的值，具体取决于它的使用位置：

- 在对象的方法中，`this` 指的是所有者对象。
- 单独的情况下，`this` 指的是全局对象。
- 在函数中，`this` 指的是全局对象。
- 在函数中，严格模式下，`this` 是 undefined。
- 在事件中，`this` 指的是接收事件的元素。

像 `call()` 和 `apply()` 这样的方法可以将 this 引用到任何对象。

在对象方法中，`this` 指的是此方法的“拥有者”。

```
var person = {
  firstName: "Bill",
  lastName : "Gates",
  id       : 678,
  fullName : function() {
    return this.firstName + " " + this.lastName;
  }
};
```

单独的 this

在单独使用时，拥有者是全局对象，全局对象是 `[object Window]`。

```
var x = this;
```

严格模式中，全局对象是 `[object Window]`。

```
"use strict";
var x = this;
```

函数中的 this

在 JavaScript 函数中，函数的拥有者默认绑定 `this`。

因此，在函数中，`this` 指的是全局对象 `[object Window]`。

```
function myFunction() {
  return this;
}
```

JavaScript 严格模式不允许默认绑定。

因此，在函数中使用时，在严格模式下，`this` 是未定义的（`undefined`）。

```
"use strict";
function myFunction() {
  return this;
}
```

事件处理程序中的 this

在 HTML 事件处理程序中，`this` 指的是接收此事件的 HTML 元素。

```
<button onclick="this.style.display='none'">
  点击来删除我！
</button>
```

21 箭头函数

```
hello = function() {
  return "Hello World!";
}

hello = () => {
  return "Hello World!";
}

// 如果函数只有一个语句，并且该语句返回一个值，则可以去掉括号和 return 关键字
hello = () => "Hello World!";

// 带参数的箭头函数
hello = (val) => "Hello " + val;

// 事实上，如果只有一个参数，您也可以略过括号
hello = val => "Hello " + val;
```

注意：

在常规函数中，关键字 `this` 表示调用该函数的对象，可以是窗口、文档、按钮或其他任何东西。

对于箭头函数，`this` 关键字始终表示【定义】箭头函数的对象

22 类

请使用关键字 `class` 创建类。

请始终添加名为 `constructor()` 的方法。

JavaScript 类*不是*对象。

它只是 JavaScript 对象的模板。

```
class Car {
  constructor(name, year) {
    this.name = name;
    this.year = year;
  }
  age(x) {
    return x - this.year;
  }
}

let myCar1 = new Car("Ford", 2014);
let myCar2 = new Car("Ford", 2014);
```

23 JSON

JSON 格式在语法上与创建 JavaScript 对象的代码相同。

由于这种相似性，JavaScript 程序可以很容易地将 JSON 数据转换成本地的 JavaScript 对象。

```
var text = '{ "employees" : [' +
'{ "firstName":"Bill" , "lastName":"Gates" },' +
'{ "firstName":"Steve" , "lastName":"Jobs" },' +
'{ "firstName":"Alan" , "lastName":"Turing" } ]}';

// 使用 JavaScript 的内建函数 JSON.parse() 来把这个字符串转换为 JavaScript 对象
var obj = JSON.parse(text);
```

23.1 方法

JSON.stringify()

任何 JavaScript 对象都可以使用 JavaScript 函数 `JSON.stringify()` 进行字符串化（转换为字符串）。

`JSON.stringify` 将日期转换为字符串。

`JSON.stringify` 不会对函数进行字符串化

```
// 对象
const person = {
  name: "Bill",
  city: "Seattle",
  today: new Date(),
  age: function () {return 19;}
};

let myString = JSON.stringify(person); //
{"name":"Bill","city":"Seattle","today":"2022-11-09T02:48:09.994Z"}
```

24 调试

24.1 console.log()方法

如果您的浏览器支持调试，那么您可以使用 `console.log()` 在调试窗口中显示 JavaScript 的值

```
a = 5;
b = 6;
c = a + b;
console.log(c);
```

24.2 设置断点

在调试窗口中，您可在 JavaScript 代码中设置断点。

在每个断点中，JavaScript 将停止执行，以使您能够检查 JavaScript 的值。

在检查值之后，您可以恢复代码执行。

24.3 debugger关键字

debugger 关键词会停止 JavaScript 的执行，并调用（如果有）调试函数。

这与在调试器中设置断点的功能是一样的。

如果调试器不可用，`debugger` 语句没有效果。

如果调试器已打开，此代码会在执行第三行之前停止运行。

```
var x = 15 * 5;
debugger;
document.getElementbyId("demo").innerHTML = x;
```

25 Map

Map 对象存有键值对，其中的键可以是任何数据类型。

Map 对象记得键的原始插入顺序。

Map 对象具有表示映射大小的属性。

方法名	描述
new Map()	创建新的 Map 对象。
set()	为 Map 对象中的键设置值。
get()	获取 Map 对象中键的值。
entries()	返回 Map 对象中键/值对的数组。
keys()	返回 Map 对象中键的数组。
values()	返回 Map 对象中值的数组。
clear()	删除 Map 中的所有元素。
delete()	删除由键指定的元素。
has()	如果键存在，则返回 true。
forEach()	为每个键/值对调用回调。

属性	描述
size	获取 Map 对象中某键的值。

能够使用对象作为键是 Map 的一个重要特性。

```
// 创建对象
const apples = {name: 'Apples'};
const bananas = {name: 'Bananas'};
const oranges = {name: 'Oranges'};

// 创建新的 Map
const fruits = new Map();

// Add new Elements to the Map
fruits.set(apples, 500);
fruits.set(bananas, 300);
fruits.set(oranges, 200);
```

26 Set

Set 是唯一值的集合。

每个值在 Set 中只能出现一次。

一个 Set 可以容纳任何数据类型的任何值。

方法名	描述
add()	向 Set 添加新元素。
clear()	从 Set 中删除所有元素。
delete()	删除由其值指定的元素。
entries()	返回 Set 对象中值的数组。
has()	如果值存在则返回 true。
forEach()	为每个元素调用回调。
keys()	返回 Set 对象中值的数组。
values()	与 keys() 相同。
size	返回元素计数。

```
// 创建 Set
const letters = new Set();

// 向 Set 添加一些值
letters.add("a");
letters.add("b");
letters.add("c");
```