ES6学习笔记

# 琐碎知识点

## 删除对象的属性

delete obj.address;

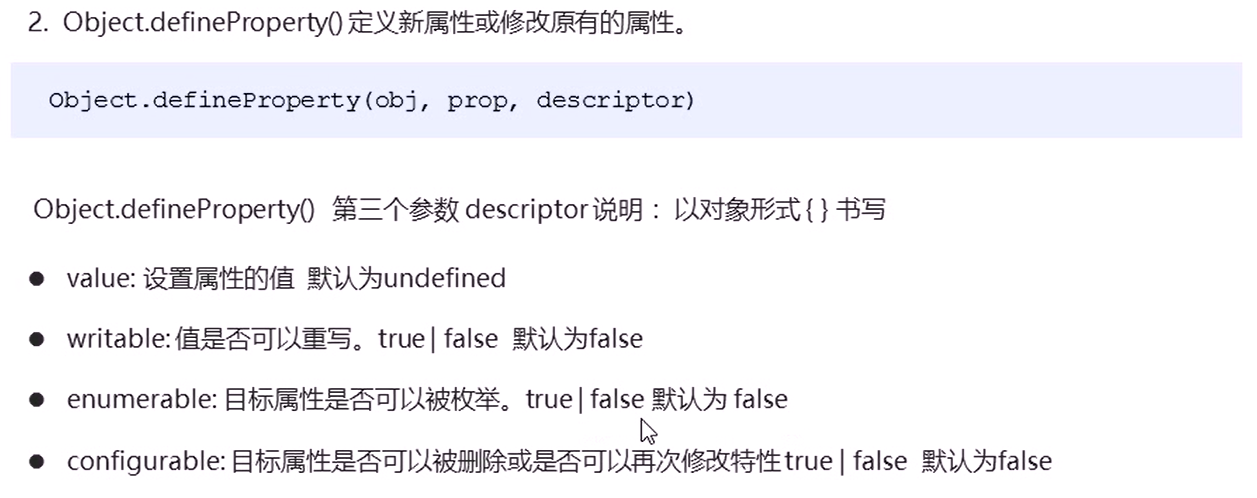
## 遍历对象的属性

console.log(Object.keys(obj));

## 添加对象属性的另一种方式

Object.defineProperty(obj, “property”, {value:”xxx”, writable:false, enumberable:false, configurable:false}

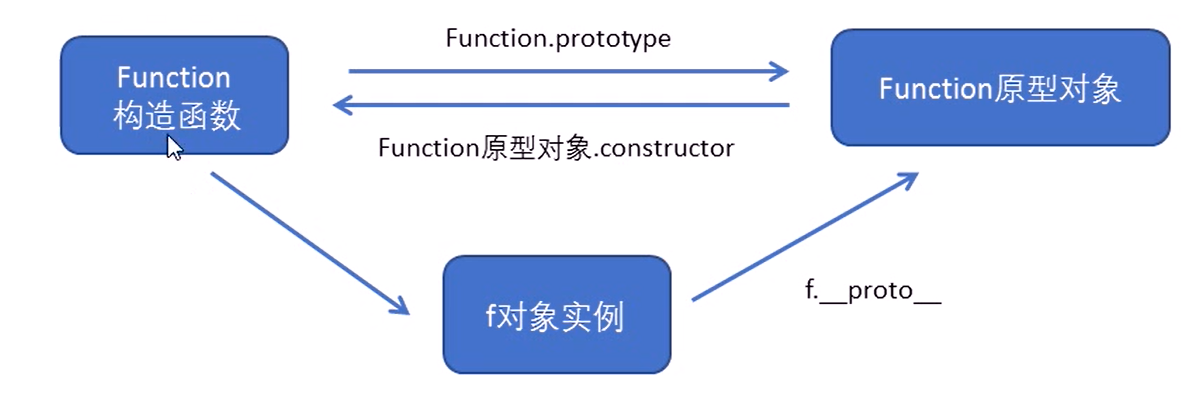
* value:属性的值，默认为undefined，如果属性存在，则该方式是修改属性的值，若不存在，则添加该属性；
* writable：属性值是否可以被修改，默认为false；
* enumerable：属性是否可以被枚举，即上面的遍历对象的属性，默认为false；
* configurable：属性是否可以被删除，删除方式见上，默认为false；



## instanceof

instanceof 运算符通过检测构造函数的 prototype 属性是否出现在某个实例对象的原型链中来判断对象是否是某个类（构造函数）的实例对象。

函数也是对象，js中万物皆对象；



## typeof

判断变量/对象的所属类型。typeof运算符把类型信息当做字符串返回，对于原始值(除了null)都可以正确判断，对于类实例(除了函数)都显示为object；

虽然typeof null的结果是object，但是null并不是对象，这是JS一个存在历史悠久的bug，在JS最初版本使用的是32位系统，为了考虑性能等原因，JS使用低位存储变量信息，000开头表示object，而null表示为全零，所以null被错误的判定为object，内容参考于：

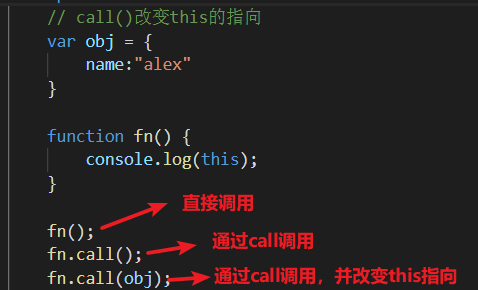
作者：冬玉  
链接：https://juejin.im/post/6873999377430642702

## 函数的定义方式

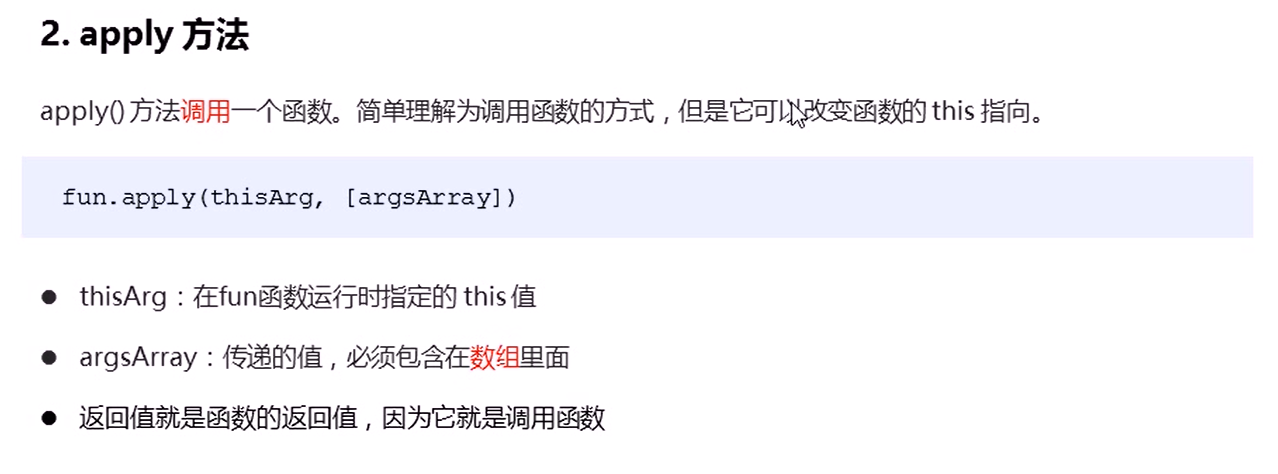


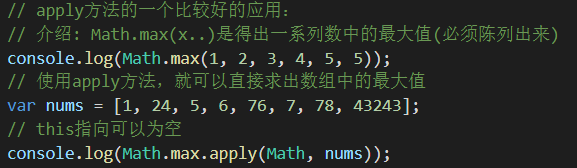
## 改变函数的this指向

* call方法；

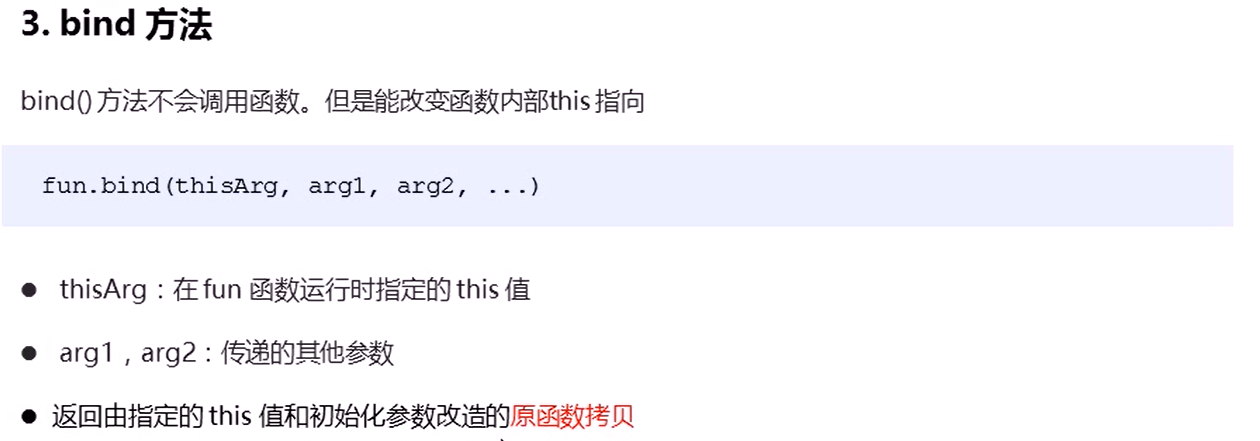


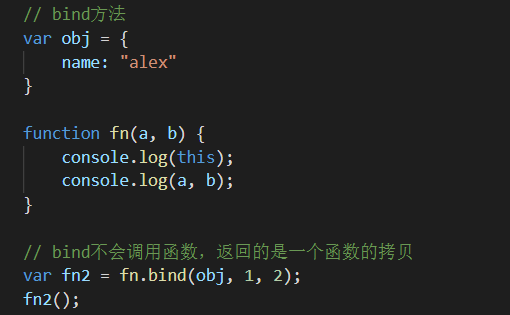
* apply方法；





* bind方法



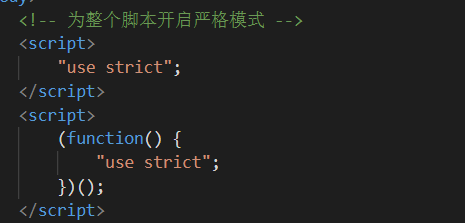


## 在script标签中声明的变量归属

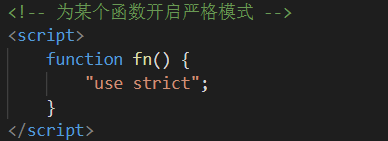
归属于window对象；

# 严格模式

* 为整个脚本开启严格模式：

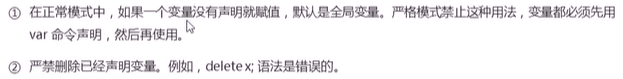


* 为整个函数开启严格模式：

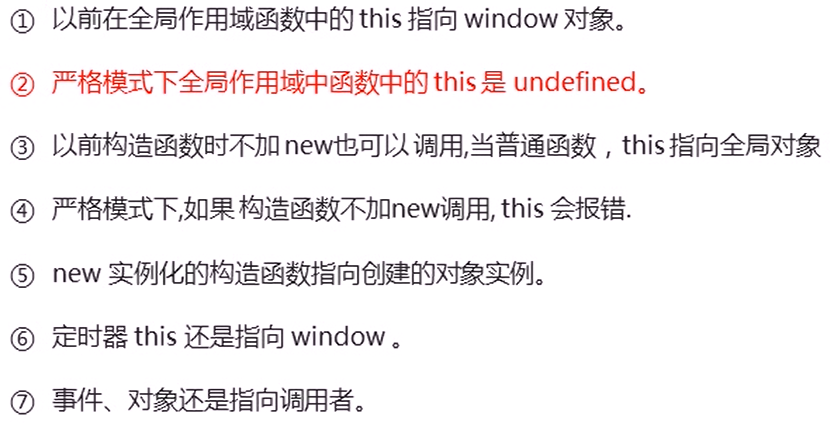


## 严格模式的规范

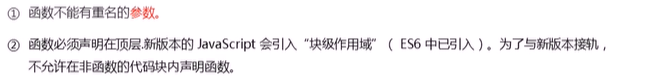
### 变量



### this指向

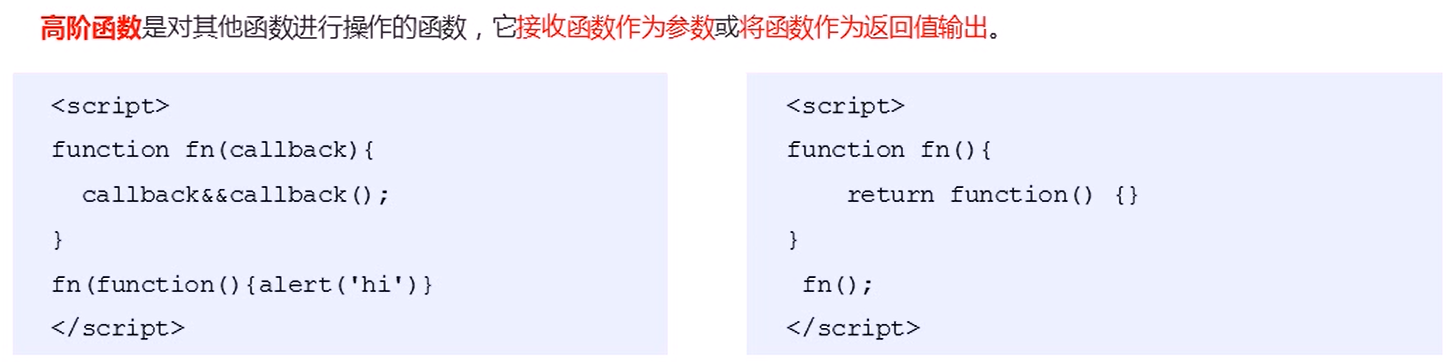


### 函数变化



函数声明不允许在非函数代码块内声明，例如if语句块、for语句块内；

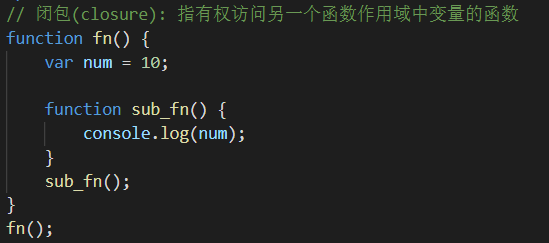
# 高阶函数

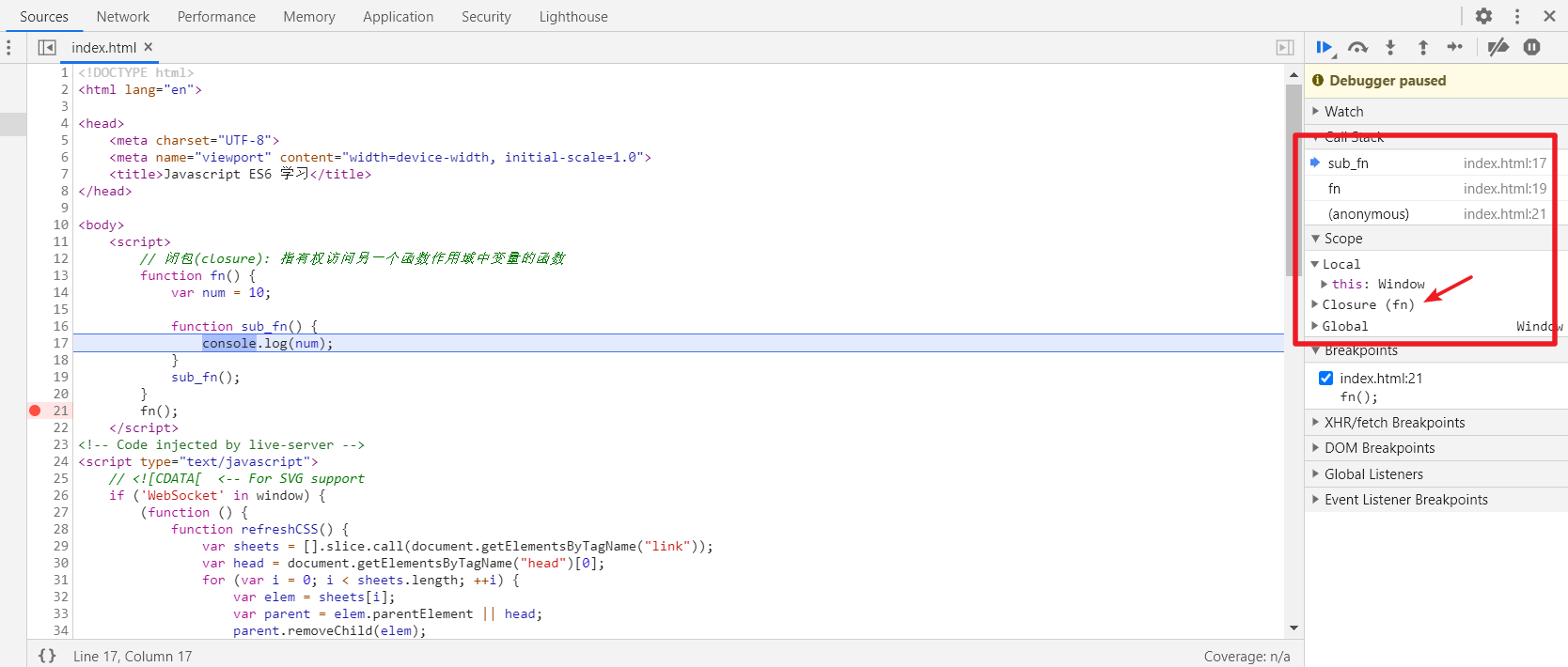


应用场景：回调函数；

# 闭包







闭包的主要作用：延伸了变量的作用范围；

## 应用案例

### 点击列表项，打印当前li的索引号

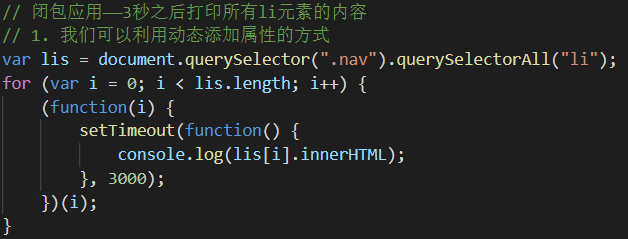
html结构: ul.nav>li\*4



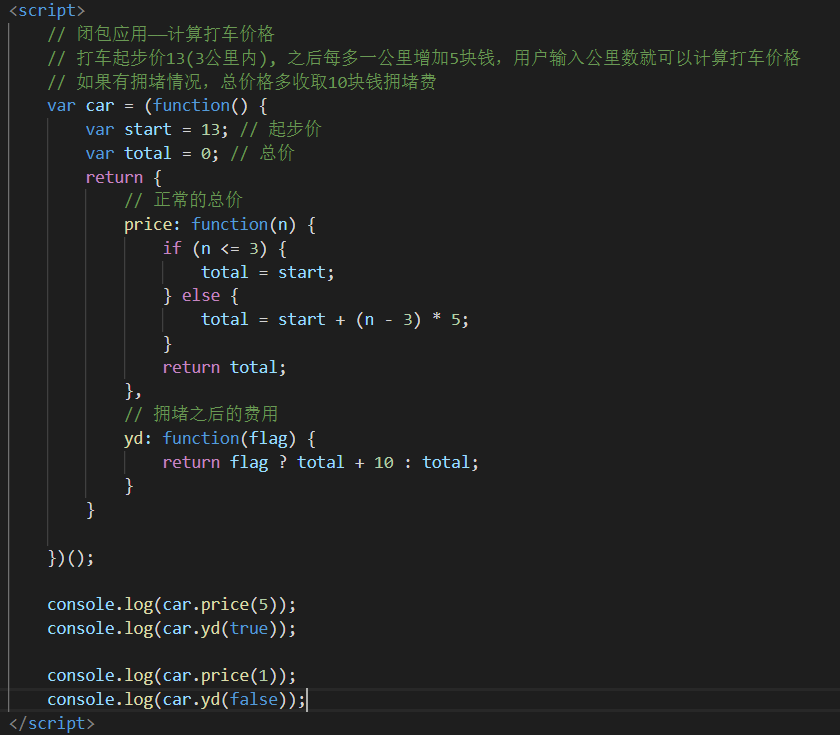
此处使用了立即执行函数；

### 3秒之后，打印li中的内容

html代码和上面保持一致；



### 打车计费

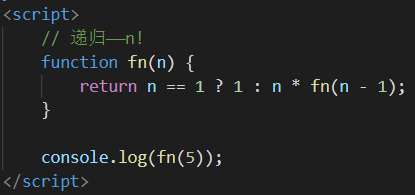


# 递归

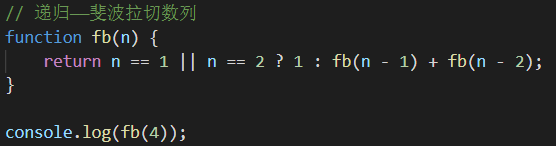
在函数内部自己调用自己；必须要有退出条件；

## 案例

### n!



### 斐波拉切数列



### 利用递归遍历json数据



// 递归——遍历数据

        var data = [{

            id: 1,

            name: "家电",

            goods: [{

                id: 11,

                gname: "冰箱",

                goods: [{

                    id: 111,

                    name: "海尔冰箱"

                }, {

                    id: 112,

                    name: "美的冰箱"

                }]

            }, {

                id: 12,

                gname: "洗衣机"

            }]

        }, {

            id: 2,

            name: "服饰"

        }]

        // 我们想要输入id，就可以返回数据对象

        // 利用forEach遍历每一个对象

        function getObj(json, id) {

            var o = {};

            json.forEach(function(item) {

                if (item.id == id) {

                    // console.log(item);

                    o = item;

                    // 我们想要得到里层的数据 11, 12 可以利用递归函数

                } else if (item.goods && item.goods.length > 0) {

                    o = getObj(item.goods, id);

                }

            });

            return o;

        }

        console.log(getObj(data, 1));

        console.log(getObj(data, 2));

        console.log(getObj(data, 11));

        console.log(getObj(data, 12));

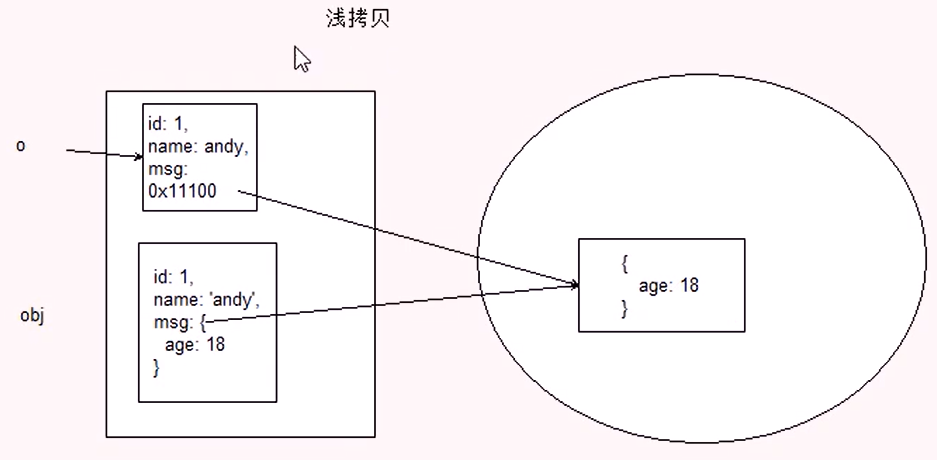
        console.log(getObj(data, 111));

    console.log(getObj(data, 112));

# 浅拷贝和深拷贝

浅拷贝只是拷贝一层，更深层次对象级别的只拷贝引用；深拷贝是拷贝多层，每一级别的数据都会拷贝；

## 浅拷贝



var obj = {

            id: 1,

            name: "alex",

            msg: {

                age: 18

            }

        }

        var o = {};

        // ES6以前的浅拷贝方式

        for (var k in obj) {

            // k是属性值，obj[k]是属性值

            // console.log(k, obj[k]);

            o[k] = obj[k];

        }

        // ES6新增的浅拷贝方法

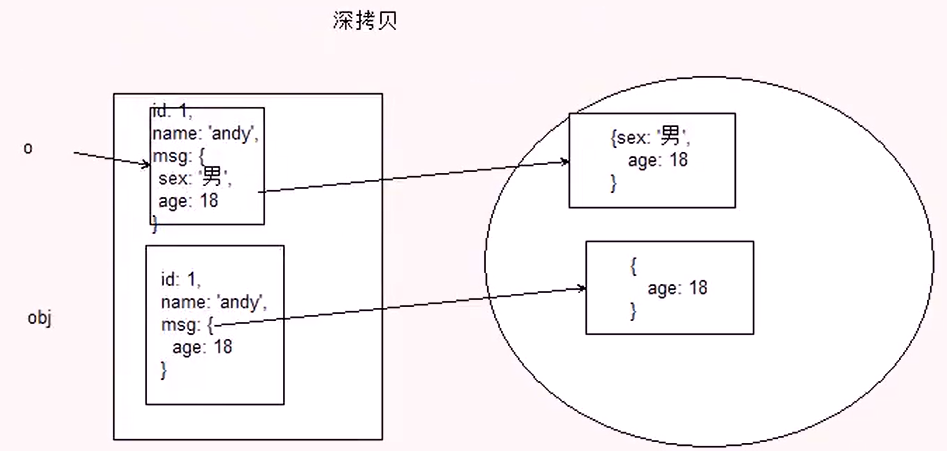
        Object.assign(o, obj);

        obj.msg.age = 20;

        console.log(obj);

    console.log(o);

## 深拷贝



var obj = {

            id: 1,

            name: "alex",

            msg: {

                age: 18

            },

            color: ["pink", "skyblue"]

        }

        var o = {};

        // 深度拷贝，封装函数

        function deepCopy(newObj, oldObj) {

            for (var k in oldObj) {

                // 判断属性值属于哪种数据类型

                // 获取属性值 k->属性, oldObj[k]->属性值

                var item = oldObj[k];

                // 判断值是否是数组<数组也是对象>

                if (item instanceof Array) {

                    newObj[k] = [];

                    deepCopy(newObj[k], item);

                } else if (item instanceof Object) {

                    // 判断值是否是对象

                    newObj[k] = {};

                    deepCopy(newObj[k], item);

                } else {

                    // 属于简单数据类型

                    newObj[k] = item;

                }

            }

        }

        deepCopy(o, obj);

        // 修改数据值，查看拷贝后的数据是否发生改变

        obj.msg.age = 20;

        console.log(obj);

    console.log(o);

## 循环

### for-in

循环数组项：

for (var index in nums) {

    }

循环数组项可以遍历数组或者对象：

遍历数组时，index是数组项的下标，使用nums[index]才能得到数组中的每项数据；

循环对象时，index是对象的属性，使用nums[index]才能得到对象中属性对应的值；

### for-of

与for-in想对应的是，for(let item of arr)中，item是数组中的元素而不是下标；

# 正则表达式

正则表达式(Regular Expression)是用于匹配字符串中字符组合的模式。在JavaScript中，正则表达式也是对象。

正则表通常被用来检索、替换那些符合某个模式（规则）的文本，例如验证表单：用户名表单只能输入英文字母、数字或者下划线，昵称输入框中可以输入中文(匹配)。此外，正则表达式还常用于过滤掉页面内容中的一些敏感词(替换)，或从字符串中获取我们想要的特定部分(提取)等。

其他语言也会使用正则表达式，本阶段我们主要是利用JavaScript正则表达式完成表单验证。

# JSON

简单转换：

* var obj = JSON.parse(str); 将JSON字符串转换成JS对象；
* var str = JSON.stringify(obj); 将JS对象转换成JSON对象；

# 正则表达式

正则表达式不需要加引号；

## 创建

// 1. 利用RegExp对象来创建正则表达式

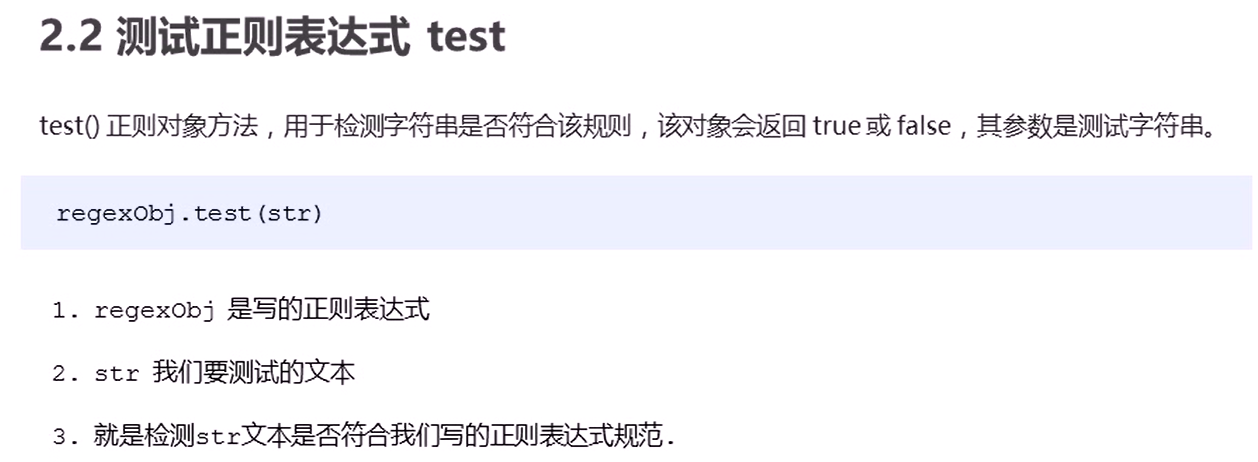
        var regexp = new RegExp(/123/);

        console.log(regexp);

        // 2. 利用字面量创建正则表达式

    var rg = /123/;

## 测试字符串是否符合



## 语法

### 边界符

* ^ 表示匹配行首的文本以谁开始；
* $ 表示匹配行尾的文本以谁结束；

// 字符串是否包含abc

var rg1 = /abc/;

// 字符串必须以abc开头

var rg2 = /^abc/;

// 字符串必须以abc结尾

var rg3 = /abc$/;

// 字符串必须为abc

var rg4 = /^abc$/;

### 字符类

字符类: []表示有一系列字符可供选择，只要匹配其中一个就可以了；

// 只要包含有 a 或者 b 或者 c 都返回true

        var rg1 = /[abc]/;

        // 三选一，只有是 a 或者 b 或者 c 这三个字母才返回true

        var rg2 = /^[abc]$/;

        // 26个小写英文字母任选一个字母, [-]表示范围符

        var rg3 = /^[a-z]$/;

        // 26个英文字母, 大写小写字母都可以, 还可以是数字、下划线、短横线(选其中一个)

        var rg4 = /^[a-zA-Z0-9\_-]$/;

        // 如果中括号里面有^表示取反，注意和边界符相区分

    var rg4 = /^[^a-zA-Z0-9\_-]$/;

### 量词符

// 量词符: 用来设定某个模式出现的次数

        // 简单理解: 就是让下面的a这个字符重复多少次

        var reg1 = /^a$/;

        // \* 相当于 >= 0, 可以出现0次或者很多次

        var reg2 = /^a\*$/;

        // + 相当于 >= 1, 可以出现1次或者很多次

        var reg3 = /^a+$/;

        // ? 相当于 0 || 1, 可以出现0次或者1次

        var reg4 = /^a?$/;

        // {3} 重复3次

        var reg5 = /^a{3}$/

        // {3,} 重复大于等于3次

        var reg6 = /^a{3,}$/

        // {3,6} 重复大于等于3次 并且 小于6次

    var reg7 = /^a{3,6}$/

### 量词重复某个模式的次数

// 量词是设定某个模式出现的次数(,之间不能有空格)

// 用户名表单验证正则表达式

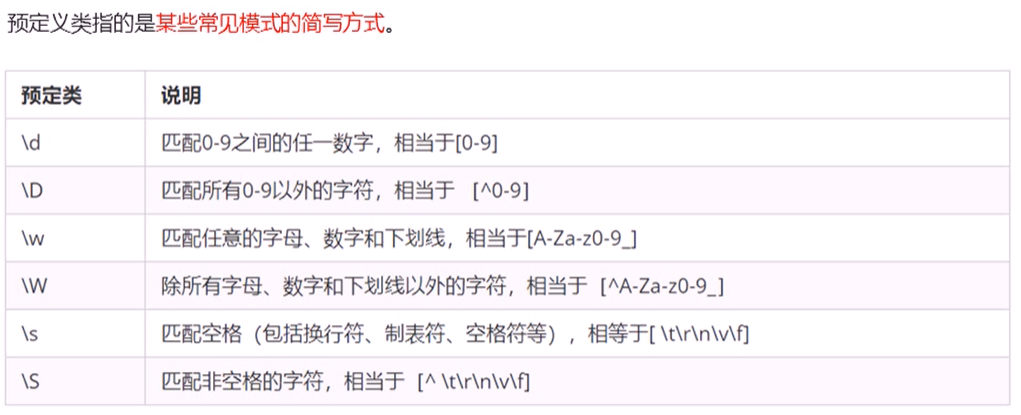
    var reg = /^[a-zA-Z0-9\_-]{6,16}$/;

### 优先级——小括号

var reg1 = /^abc{3}$/; // 匹配abccc

var reg2 = /^(abc){3}$/; // 匹配abcabcabc

### 预定义类



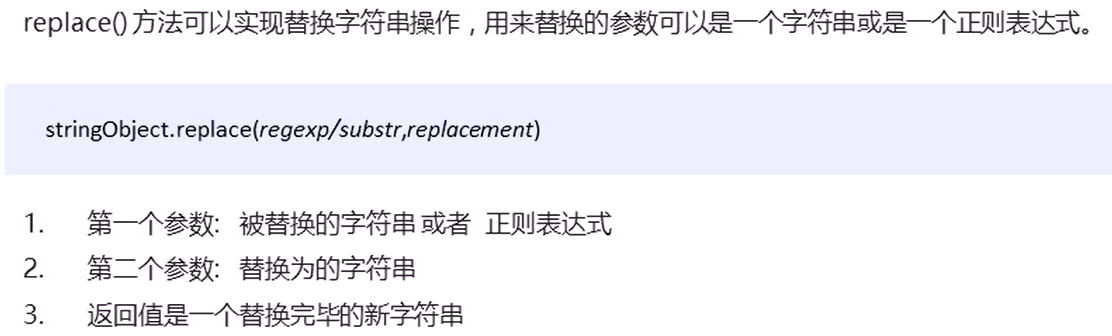
### 预定义类案例——座机号验证(正则表达式的或)

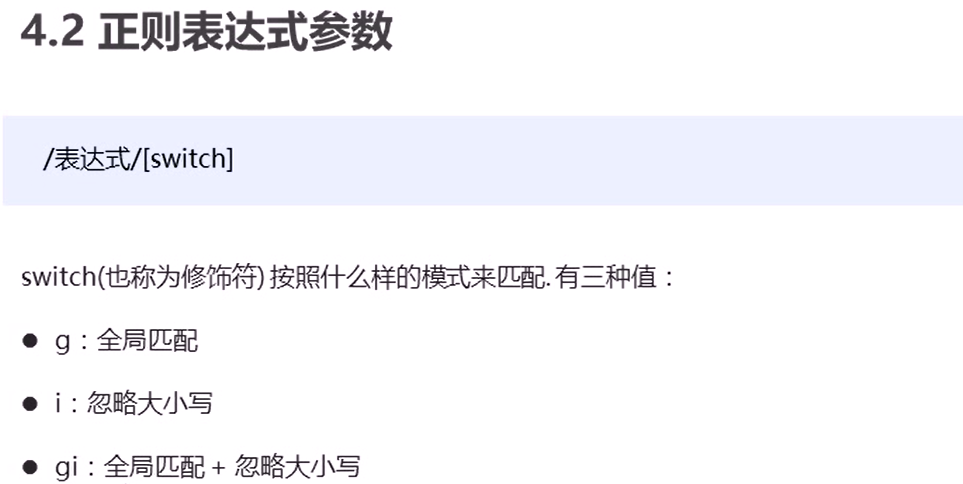
// 座机号验证 全国座机号两种格式: 010-12345678 或者 0530-1234567

// 正则表达式中的或者符号: |

    var reg = /^\d{3}-\d{8}|\d{4}-\d{7}$/;

### 正则替换





<textarea name="" id="msg" cols="30" rows="10"></textarea><button>提交</button>

    <div></div>

    <script>

        // // 正则表达式中的替换

        // var str = "Tom 和 Jerry";

        // // var newStr = str.replace("和", "and");

        // var newStr = str.replace(/和/, "and");

        // console.log(newStr);

        var text = document.querySelector("#msg");

        var btn = document.querySelector("button");

        var div = document.querySelector("div");

        btn.onclick = function() {

            div.innerHTML = text.value.replace(/激情/g, "\*\*");

        }

    </script>

# ES6

## let关键字

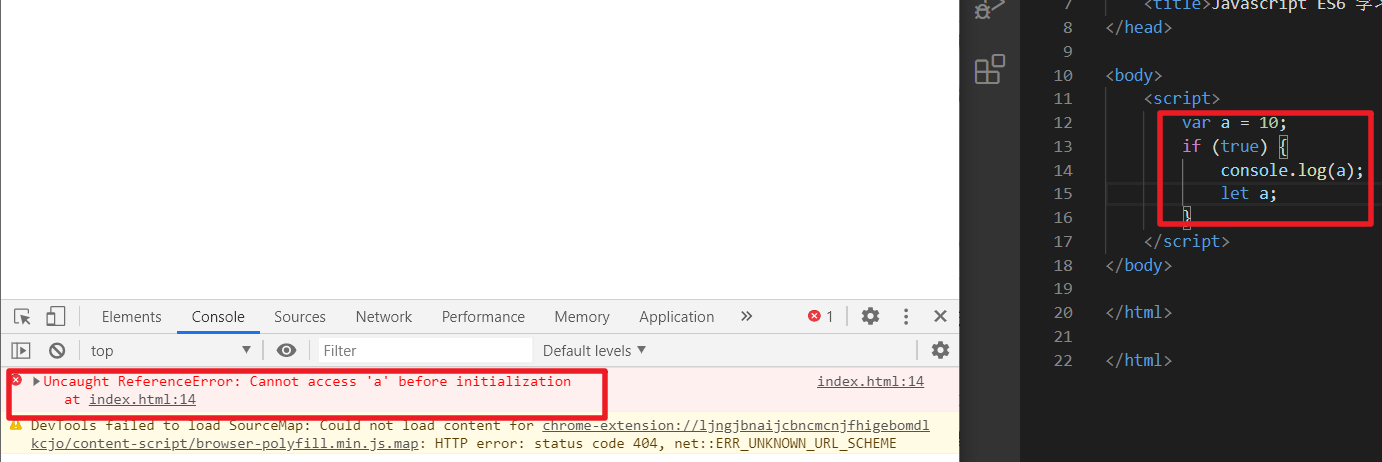


在大括号中，使用let关键字声明的变量才具有块级作用域，var关键字声明的变量是不具备这个特点的；

使用let关键字可以防止循环变量变成全局变量；

使用let关键字声明的变量没有变量提升，即变量只有声明后才能使用；

使用let关键字声明的变量具有暂时性死区的特性，即在块级作用域中申明的变量会绑定在当前作用域中，不管与外部的变量(是否同名)产生联系;



## const关键字

作用：声明常量，常量就是值(内存地址)不能变化的量；

const关键字具有块级作用域特性；

const关键字申明常量时必须赋初始值；

const关键字赋值以后不能更改(地址)；

const i = 1;

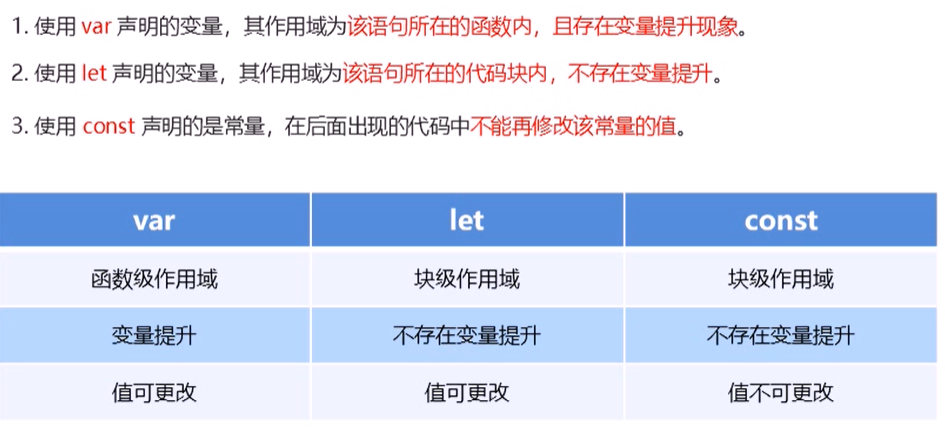
i = 2; // 普通数据类型，修改会报错

const arr = [100, 200];

arr[0] = "kkk"; // 对于复杂数据类型，因为地址没变，所以不报错

arr = [1, 2]; // 地址改变了，报错

## var、let、const关键字的区别



这篇博文讲得比较清晰：<https://blog.csdn.net/er_fan/article/details/84139372>

### 易混点——变量提升以及全局变量的定义

var temp = new Date()

function fn() {

  console.log('msg1', temp) // msg2 undefined

  var temp = '123'

  console.log('msg2', temp) // msg2 123

}

console.log('msg3', temp) // msg3 Tue Nov 17 2020 17:01:15 GMT+0800 (中国标准时间)

fn()

// 输出顺序 msg3 -> msg1 -> msg2

ES5以前若要在函数中定义全局变量，则不能用var申明变量直接使用变量名就是定义全局变量，在函数中使用var申明的变量，存在变量提升的现象，即JS解析器在执行代码之前先预解析代码， 获取所有被声明的变量和函数声明再一行一行的执行代码，所以上述代码的fn()函数等价于：

function fn() {

  var temp

  console.log('msg1', temp) // msg2 undefined

  temp = '123'

  console.log('msg2', temp) // msg2 123

}

这就是变量提升，同时注意，ES5之前只存在两种作用域——全局作用域和函数作用域，在上述代码fn()函数中申明的temp函数作用域变量会覆盖全局作用域中的temp变量（**注意**：for循环中使用var声明的变量在外部也可以使用，这是因为for循环不是函数，没有函数作用域，那么在当前区域当然可以使用咯）。

### let声明变量——暂时性死区

var tmp = 123;

if (true) {

  tmp = 'abc'; // Uncaught ReferenceError: Cannot access 'tmp' before initialization

  let tmp;

}

使用let关键字声明的变量具有块级作用域的概念，类似的在函数中使用var声明的变量具有函数作用域，它们都可以屏蔽外部的“全局变量”，与var声明变量的预解析不同的是，let声明方式不能在声明之前使用该变量，否则会报错，这就是暂时性死区的概念。

### const声明常量时必须赋值(初始化)

const PI // Uncaught SyntaxError: Missing initializer in const declaration

### let、const声明变量不能重名

与var声明变量(后者覆盖前者)不同的是，使用let、const声明变量，变量名不能相同，否则会报错：

var num = 10

var num = 20

console.log(num) // 20

let message = "123"

const message = "456" // Uncaught SyntaxError: Identifier 'message' has already been declared

## 解构赋值

### 数组解构

// 数组解构允许我们按照一一对应的关系从数组中提取值，然后将值赋值给变量

let arr = [1, 2, 3];

// 等号左边是解构，不是数组

let [a, b, c] = arr; // a=1, b=2, c=3

特殊情况：

// 若等号右方值的个数不足，那么等号左方对应位置没有值的变量会被赋值为undefined

let [num1, num2, num3] = [1, 2]

console.log(num1, num2, num3) // 1 2 undefined

// 若等号右方是字符串，那么在执行结构赋值操作时，字符串会默认转换为数组，再进行结构操作

let [a, b, c, d, e] = 'world'

console.log(a, b, c, d, e) // w o r l d

// 可以设置默认值，如果对应变量被赋值为undefined，那么该变量会被赋值为默认值

let [n1, n2, n3 = 4] = [1, 2]

console.log(n1, n2, n3) // 1 2 4

### 对象解构

// 对象解构允许我们使用与对象属性名相同的匹配方式给变量赋值(与顺序无关)

let { a, b } = { b: '李四', a: '张三' }

console.log(a, b) // 张三 李四

还有另一种更灵活的写法：

// 变量名称也可以改变(注意别名的位置！！！)

let { a: name1, b: name2 } = { b: '李四', a: '张三' }

console.log(name1, name2) // 张三 李四

特殊情况：

// 如果在等号右边的对象中找不到属性，那么值会被赋值为undefined

let { a: name1, c: name3 } = { b: '李四', a: '张三' }

console.log(name1, name3) // 张三 undefined

// 对应于数组解构中字符串隐式转换成数组，在对象解构中也会隐式转换成对象

let {toString: s} = 123; // 123 隐式转换成了Number对象

console.log(s) // ƒ toString() { [native code] }->Number.prototype.toString()方法

难点：

例一：

// 在数组解构中，如果等号右边对应位置没有元素，那么会被赋值为undefined，

// 在对象解构中，是根据属性名去寻找元素的，与位置无关

const node = {

  loc: {

    start: {

      line: 1,

      column: 5

    }

  }

};

// 对象解构，会根据对应结构去解构元素

let { loc, loc: { start }, loc: { start: { line } } } = node;

console.log(loc, start, line) // {start: {…}} start: {line: 1, column: 5} 1

例二：

const obj = {name: '日用品', params:{ contain: '500ml', price: '￥6' } }

const {name, params, params: { contain: size, price } } = obj

console.log(name, params, size, price) // 日用品 {contain: "500ml", price: "￥6"} 500ml ￥6

可理解为，给对应位置的变量赋值，变量为等式左边单个属性or属性的值，例如上述代码的name，size；

## 箭头函数

// 箭头函数是用来简化函数定义语法的

        const fn = () => {

            console.log("hello");

        }

     fn();

在箭头函数中，如果函数体只有一句代码并且代码的执行结果就是函数的返回值，函数体大括号可以省略(**可以优化理解为只要函数体只有一句代码，那么就可以省略函数体的大括号，只是返回值为undefined罢了**)；

const sum = (num1, num2) => num1 + num2;

        console.log(sum(1, 2));

在箭头函数中，如果函数形参只有一个，那么可以省略小括号；

const println = str => console.log(str);

        println("hello, ES6");

箭头函数不绑定this，箭头函数没有自己的this关键字，如果在箭头函数中使用this关键字，this将指向箭头函数定义位置所在的this；

function fn() {

            console.log(this);

            return () => console.log(this);

        }

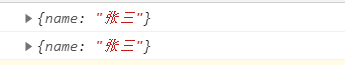
        const obj = {

            name: "张三"

        }

        const return\_fn = fn.call(obj);

     return\_fn();



const obj = {

            name: "张三",

            say: () => console.log(this)

        }

     obj.say();



因为箭头函数不绑定this，在定义对象的时候没有块级作用域，所以this指向window对象(对于普通函数this则指向obj，因为obj调用了它)；

### 箭头函数补充点

* 函数体内的this对象，就是定义时所在的对象，而不是使用时所在的对象；
* 不可以当作构造函数，也就是说，不可以使用new命令，否则会抛出一个错误；
* 不可以使用arguments对象，该对象在函数体内不存在。如果要用，可以用 rest 参数代替；
* 不可以使用yield命令，因此箭头函数不能用作 Generator 函数；

上面四点中，**第一点**尤其值得注意。this对象的指向是可变的，但是在箭头函数中，它是固定的；

## 剩余参数

剩余参数语法允许我们将一个不定数量的参数表示为一个数组；

// 剩余参数，将不定数量的参数表示为一个数组，语法: "...arr"

        const sum = (...args) => {

            let total = 0;

            // // forEach语法

            // args.forEach(function (item, index, arr) {

            //     console.log(item);

            //     console.log(index);

            //     console.log(arr);

            // });

            args.forEach(item => total += item);

            return total;

        }

### 剩余参数与数组解构配合使用

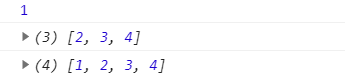
let [a, ...b] = [1, 2, 3, 4];

        let [...c] = [1, 2, 3, 4];

        console.log(a);

        console.log(b);

    console.log(c);



## 扩展运算符

扩展运算符可以将数组拆分成以逗号分隔的参数序列；

  var arr = [1,2,3];

// ...arr <==> 1,2,3

     console.log(...arr); // <==> console.log(1, 2, 3);

扩展运算符可以运用于数组的合并；

// 扩展运算符运用之一——合并数组

        // 方法一

        var arr1 = [1, 2, 3];

        var arr2 = [4, 5, 6];

        var arr3 = [...arr1, ...arr2];

        console.log(arr3);

        // 方法二

        var arr4 = [1, 2, 3];

        var arr5 = [4, 5, 6];

        arr4.push(...arr5);

        console.log(arr4);

        // 数组的push方法可以向数组中插入一些列以数字分割数字序列

        arr4.push(7, 8);

     console.log(arr4);

扩展运算符还可以将伪数组转换成真正的数组；

var lis = document.getElementsByTagName("li");

        console.log(lis);

        var lis\_arr = [...lis];

     console.log(lis\_arr);



## Array新增方法

### Array.from()

Array.from方法也可像扩展运算符一样将伪数组转换成为数组；

var arr1 = {

            "0": "1",

            "1": "2",

            "2": "3",

            length: 3

        }

        var arr2 = Array.from(arr1);

    console.log(arr2);



Array.from还有第二个可选参数，在对每一项进行转换时，可以对每项进行相应的处理；

  var arr1 = {

            "0": "1",

            "1": "2",

            "2": "3",

            length: 3

        }

        var arr2 = Array.from(arr1, item => item \* 2); // 每项乘以二

    console.log(arr2);



### Array.find()

查找数组中第一个满足条件的元素并返回，如果没有查到，则返回undefined；

  const arr = [{

            id: 1,

            name: "张三"

        }, {

            id: 2,

            name: "李四"

        }];

        // 每次查找一个元素都会调用传入find方法的函数，如果满足条件则返回该元素，并退出查找

        var target = arr.find((item, index) => item.id === 2);

    console.log(target);



### Array.findIndex()

查找数组中第一个满足条件的元素并返回，如果没有查到，则返回-1；用法和Array.find()方法保持一致；

### Array.includes()

查找数组中是否包含给定的值，返回布尔值；

console.log([1, 2, 3].includes(2)); // true

console.log([1, 2, 3].includes(4)); // false

## 模板字符串

用反引号定义``，并具有以下特点：

模板字符串解析变量，通过${var}的方式；

模板字符串可以换行，在模板字符串中换行，输出也是换行的；

模板字符串可以调用函数，同样通过${fn()}方式调用，在此处显示调用函数后的返回值；

const id = "201731062329";

        var getName = () => "吕登名";

        var info = `学号:${id},  姓名:${getName()}`;

        var html = `

            <div>学号:${id}</div>

            <div>姓名:${getName()}</div>

        `

        console.log(info);

    console.log(html);



## String新增方法

### String.startsWith()和String.endsWith()

startsWith：判断字符串是否以某一字符串开头，返回布尔值；

endsWith：判断字符串是否以某一字符串结尾，返回布尔值；

### String.repeat()

repeat：将原字符串重复n次，返回一个新的字符串；

## Set数据结构

ES6提供了新的数据结构Set，它类似于数组，但是成员的值都是唯一的，没有重复的值；

### 创建

Set本身是一个构造函数，用来生成Set数据结构；Set函数可以接受一个数组作为参数，用来初始化；

const set1 = new Set();

        console.log(set1.size);

        const set2 = new Set([1, 2, 3]);

    console.log(set2.size);

### 利用Set数据结构给数组去重

const set = new Set([1, 2, 2, 3, 4, 4]);

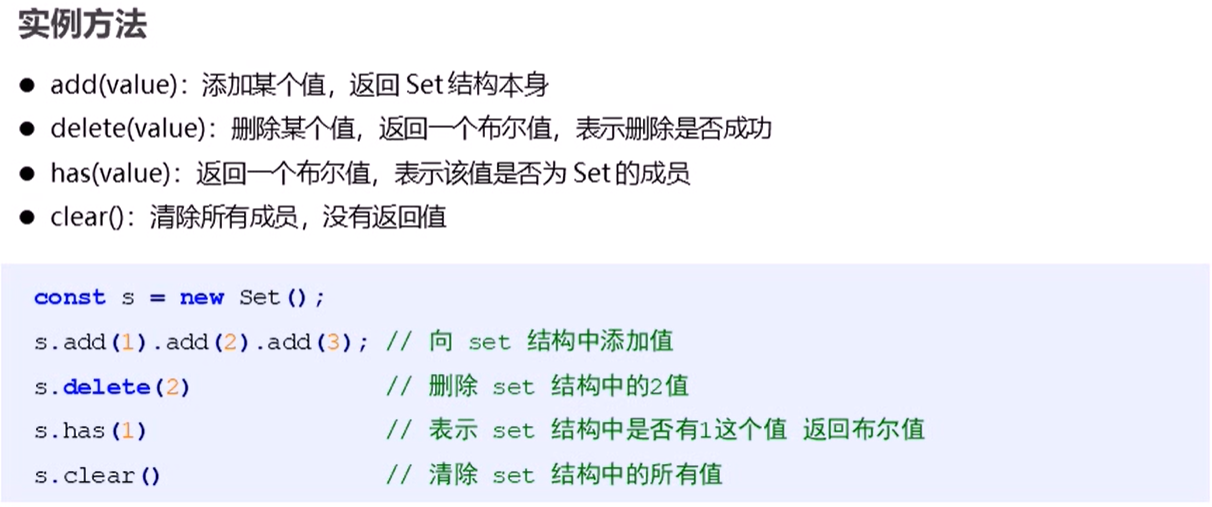
        console.log(set.size);

        var arr = [...set];

    console.log(arr);



### Set结构的内置方法



var set = new Set();

        // 用add方法添加数据

        set.add(1).add(2).add(3);

        console.log(set);

        // 用delete方法删除数据

        const res = set.delete(1);

        console.log(set);

        console.log(`删除结果:${res}`);

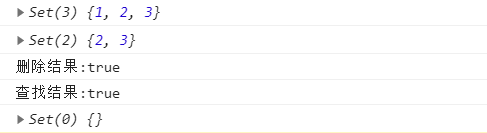
        // 用has方法查找数据

        console.log(`查找结果:${set.has(2)}`);

        // 用clear方法清空Set

        set.clear();

    console.log(set);

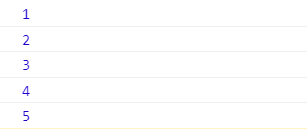


### 遍历Set元素

方式和数组一样，使用forEach方法；

const set = new Set([1, 2, 3, 4, 5]);

set.forEach(item => console.log(item));



# 完结时间：2020年8月8日22:33:49