Simi Docs / JavaScript

Source: https://github.com/saidake/simi-docs/tree/release/simi-docs-1.6.0

Edited by: Craig Brown
Version: 1.0.0
Date: Feb 2, 2025

JavaScript / Concept

JS 位置

<script type="text/javascript " language="JavaScript "></script> 插入 head 或 body 标签中（type 属性错误不执行内部代码，language 可省）

<script src="name.js"> </script> js 文件引入 （出错不执行后方代码，但不影响其他 script 执行） （不同 script 数据互通，重名覆盖）

<input type="button" onclick="alert('fafa');" /> 行内引号内插入 js 事件代码

script{ display: block; } css 可以将 script 的隐藏内容在页面显示出来

标识符

由字母 下划线 数字 $组成 不能数字开头 [(+ - / 句首为这五个符号时必须在前句加分号

//content /* content */ JS 语句后可省略分号 ；

模块规范

CommonJs 规范： 针对服务端 nodejs，不适用于浏览器环境，

nodejs v13.2 版本 识别 js 文件时， .mjs 结尾视为 ES6 模块 .cjs 结尾视为 CommonJS 模块

可以在包的 package.json 文件中增加 "type": "module"信息。nodejs 则将整个包都视为 ES6 模块来加载运行。

AMD 规范： 针对浏览器，也采用 require()语句加载模块，但是不同于 CommonJS

require(['moname'], ()=>{ moname.test(1,2); }); 加载的模块数组，成功之后的回调函数

JavaScript / Core

变量常量

var a=999,**b**=a/10 **var a**=0b11001 **var a**=-017 **var a**=-0xD8 **var a**=2e-5 **var a**=4.4E-10 **number** 数字（4byte）【进制表示时没有符号位，需要开头加 -，转换成十进制计算输出】

特殊值：

NaN（不是一个数字） **Infinity**（无穷大） **-Infinity**（无穷小）

var a='z\u' **var a**='g"kk' **var a**=`dong${obj.name }\ fafa` **string** （字符 ASCII 码比较 'a'>'B'）【字符串赋值或拼接会在堆中不断占用空间，\后可以换行】

在字符串中，

任何字符加反斜杠还表示字符本身，如字符串"\u"就被解释为 u 本身

var a=true **var a**=false **boolean**（true: **非 0** false: **0 == ""** =/= **null ==undefined** =/= **NaN** ）

var a=undefined **undeined** 未定义（没有值的变量，函数的默认返回值）

var a=null **null** 空对象（必须手动设置，主动赋值）

var a= new Person() **object** 对象（数组和对象都是地址赋值，能用地址直接修改内部值） 【var a=new Number() new String() new Boolean() --> 基本类型变成 对象 】

var a=function(){} **function** 函数地址的类型为 function 【最外部定义的变量和函数都是 window 的成员】

const a=5 **const per**= new Person() **常量**，只能被初始化一次

var arr = [] **var arr** = [10] **var arr** = [1，2] **var arr** =[a, b] **var arr** = [{name:"小明" }] **var arr** = [function(){}, function(){}]

var arr = new Array() **var arr** = new Array(10) **var arr** = new Array(1，2) **数组**，可以跨越长度追加元素（单个数字 10 表示初始数组空间，表示为 empty； 连续数字 1，2 表示给赋值数组 ）

var arr={ 0:10, 1:20, 2:30, length:3 } **伪数组**，长度不可变，不能使用数组内置方法【 对象和数组都可以使用 obj[ind] 的形式访问成员 】

let arr={time:string,value:number}[] let arr= [{time:string,value:number}] 类型放在外面数组可以为空，放在里面数组必须至少有一个元素

栈存储 boolean、number、string、undefined、null，对象指针，数组指针 （每个小单元大小基本相等）
堆存储 object

类型判断 typeof NaN=="number" true NaN instanceof Number==false 99 instanceof Number==false
字符串拼接 1+-'9' = -8 1+-'9gg' = NaN 'dong' + 'ff' + num [1,2]+999=1,2999 一边是字符串就会强制拼接成字符串（数组自动转为字符串，每个拼接成员都会调用 toString 方法）
大量拼接字符串会占用很多空间，避免使用。
真判断 {} [] [0,0] {}==true var str=[]; str.push(input.value) str.join("|") 用数组防止重复开辟空间
假判断 [1]==[1] [0]==true []==true [1,2]==true !!false NaN==NaN （数组默认取第一个值判断）

类型转换

自动转换： 3/2 = 1.5 3.5/2 = 1.75 4/2 =2 -' 1 ' =1 -' 1a ' = NaN
判断类型 **typeof** name **typeof**(name) 获取变量基本类型（typeof 能够获得的类型： undefined、object、function、boolean、string、number）
Object.prototype.toString.call(obj) 返回字符串 对象的基本类型和内置类型 [Object Array]
b.**constructor**==Number 判断类型（b=9 普通变量也可以判断，但是不能直接使用数字）
a **instanceof** Object 判断，所属父类（只能判断对象，基础类型 99，'aaa'无法判断）
isNaN(name) 判断，不是数字为 true

-->number parseInt('10g' ,2); parseInt('1111', 2) -->15 转 10 进制整数，转 2 进制 （直接 去掉小数点后和字母后的全部， 包含小数点和原字母 或 开头为字母输出 NAN）
parseFloat('10g'); 转小数，空字符串和 null 都会转为 NAN （直接 去掉字母后的全部， 包含原字母 或 开头为字母输出 NAN）
Number('10g'); 转数字 含字母就为 NaN（严格） // Number({})==NaN Number([])==0
-->string String(val) 允许 undefined null // String(undefined)=== 'undefined' String(null)==='null' String(false)==='false' String(1)==='1'
obj.toString(16) 数字转换 16 进制字符串，undefined null 报错 （Object.prototype 内置函数）
-->boolean Boolean(val); !!val 等同 if() 括号内做的事

语句运算符

for(var i=0;i<3;i++); for 中第一条语句可以定义变量 （var 定义可在外部使用，let const 定义只能在 for 内部使用）
for(var ind **in** arr **)** { arr[ind] } 索引遍历 数组/对象/字符串 的键或索引（ES5，遍历对象：遍历的键是无序的可枚举属性，包括从原型上继承的成员）（遍历数组：数组遍历不一定按次序访问元素，用 for of）
for(var val **of** obj **)** { obj[i] } 值遍历 数组/可迭代对象/字符串 （ES6，可迭代对象包括 Array，Map，Set，String，TypedArray 对象等，new 对象和普通 Object 不能迭代）
throw "msg" throw err 终止整个 script 内代码程序执行，报错并把 err 变量或字符串打印到控制台 【var err=new Error("error message")】
const result=delete per.name; 释放地址的内部成员（删除对象成员时，不会影响原型的成员）（删除数组成员时，不会影响其他索引值，当前索引变 empty）（delete null; 可以无限删除，没有次数限制）

a === b a !== b 严格判断（限制类型）

JavaScript / Functionality

作用域

函数外---全局变量 （关闭页面释放） 【if，for 等语句没有作用域】

函数内---局部变量（函数执行结束释放）

块级作用域: let，const 只在{ }内使用 （var 定义后可以在外部使用）

访问规则: 函数内---->函数外 函数外--||-->函数内

隐式全局变量: a=100 定义后任何地方都可以访问 // var a = b = c =100; ---> var a=100; b=100; c=100;

预解析: 提前解析代码，声明提前到当前作用域的最上面---赋值，调用不提前（变量提升到最上方，会被同名函数覆盖） // 提前 var func; let const 重名报错

严格模式

"use strict" ES5 表达式

不允许使用未声明的变量，不允许删除变量或对象或函数，不允许变量重名。

不允许使用八进制，不允许使用转义字符。

不允许对只读属性赋值，禁止 this 关键字指向全局对象。

函数

function Person(a=1, b=2) { } 不可重载：同名直接覆盖， 可却省：缺省并且没有值的变量会显示为 undefined

可多参：多余参数不报错 （数组和对象传入的是地址，函数内部可以直接修改内部值 ）

(function( ){ }) **( )**; (function( ){ } **( )**); 自调用（函数中外部 window 成员被重名时，可以使用 window 指定：window.document，script 中 this 指向 window）

匿名函数 **function( ){ }**和 函数名 都等同地址，指向在内存中的函数代码（可以调用函数，作形参，返回值）

闭包 在父函数内定义的子函数可以访问在父函数作用域中声明和定义的变量，当父函数返回子函数时，子函数正在使用的变量不会被释放（let var 都可以产生闭包）

内置：arguments[0] 函数内部使用，实参伪数组（长度不可变）

Person.name 函数名--只读 Person.length 形参个数 Person.caller 调用者

Person.**call**(other, **a, b**) Person.**apply**(other, **[a, b]**) 调用函数并传参（临时改变函数内 this 地址指向，other=null 或无为 window）

var Person=function(a,b){}.**bind**(other, **1**) 返回一个新函数地址（newfun 的 this 指向永久改变为 other，指定的形参 a 固定不可变，未指定的形参的 b 接受参数 1）

防抖： 短时间内大量调用，只会执行一次函数 （定义中间函数利用闭包 timer，中间函数调用时，如果 timer 存在清空定时器，每次都会赋值新的定时器延迟执行）

节流： 函数执行一次后，时间段内暂时失效，过了时间段后再重新激活 （定义中间函数利用闭包 flag，中间函数调用时，如果 flag 为假终结函数，每次都会赋值 flag 为假并设置定时器结束执行并赋值真）

类与对象

function Person( ){ let sta=99; this.a =99; this.b=function(){ this.a } **}** 内部定义成员，成员的函数可以访问 this 数据，内部定义变量相当于闭包的静态成员

var obj = new Person() **var obj** = new Person **var obj={**"name":"小明", suiban:function(e){} **}** 不传递参数创建时可省略括号( )， { }是 object 创建对象

obj.mem=99 **obj.["mem"]=99** **obj.mem()** **obj.["mem"]()** 可点追加属性，可存 DOM 对象

创建对象过程 创建一个新对象 var person={}

__proto__ 指向构造函数的原型对象

构造函数作用域赋值给新对象（使 this 指向新对象）

执行构造函数代码 this 追加属性

返回新对象（创建完毕所有成员就都被赋值了，也都能互相访问）

原型链　　prototype（新函数内置的属性）　constructor（原型对象内置的属性）　__proto__（实例对象内置属性）

Person.prototype　{ constructor: Person，__proto__: Object.prototype }　**per**　Person { __proto__: Person.prototype }

Object 没有父类 Object.prototype.__proto__===null （prototype 内定义的函数能访问 this 成员，能输出调用者内部的 this.a 数据）

构造器　作用：当无法访问父函数时 Person.prototype，通过对象访问 prototype （初始化原型，必须重置 constructor）

Person.prototype == per.__proto__　Person.__proto__===Function.prototype　构造函数的父函数是 Function，prototype 也可以储存公共成员，和父类原型对象的地址，父类一些信息

Person.prototype.constructor ===　per._ _proto__.constructor ===　Person　constructor 内部添加变量直接可以通过 Person.xx 访问，但定义的对象或者内部成员都无法访问

继承　替换　function **Son**(a, b, c){　Person.call(this, a, b)　} 继承父类内部成员

Son.prototype=Object.create(Person.prototype) 开启公共属性　Son.prototype.constructor=Son 开启构造器（不能继承父类原型成员 Father.prototype）

修改构造函数　Son.prototype=new Father()　父类内部成员变成了公共成员，会导致同步更改 Son.prototype===Father.prototype，默认访问__proto__间接实现了 constructor （ 无法实现多继承，无法向父类传参 ）

函数对象　函数是 Function 对象，对象不一定是函数

函数是 Function 类型的　var fun=new Function("a","b","return a+b");　fun(1, 2);　效率低，字符串线解析

属性名　obj['1'] === obj[1]　obj[true] === obj['true']　obj[undefined]===obj['undefined']　自动调用 String 转换键名为字符串，symbol 类型作为键不会重复，

浅拷贝：地址赋值（对象数组是地址复制，同步修改）

深拷贝：复制成员构建新对象或数组（同时避免复制成员时出现地址赋值）

DOM 原型链：divObj　HTMLDivElement　HTMLElement　Element　Node　EventTarget　Object　null

JavaScript / ES6

变量常量

{ let a=100　const b=20　}　let 和 const 变量只在 { } 块作用域内部有效 外部无法访问（ES6 都要使用严格模式）

let 和 const 定义的变量常量再次定义会报错（var 会覆盖）

let 和 const 定义不会被预解析

展开符　let [arg1,arg2,...arg3] = [1, 2, 3, 4];　arg3 = ['3','4']　let { x, y, ...z } = { x:1, y:2, a:3, b:4 };　z = { a:3, b:4 }　解构赋值展开符只能在最后方

function fun(arr, ..res){　}　fun(1,2,3,4,)　res=[2,3,4]　将多余的参数转换成数组

newArray=[...newArray, [...arr]]　[...new Set(newArray)]　展开只能放在最前方，不能去重，可以防止地址赋值

成员简写　**let obj=　{ aa(){},**　bb: function(){}　**}**　对象的 键值对匿名函数成员可以合并为函数

动态键　let global='left'　let obj={ ...otherobj,　[global]:99　}　const { [global]: newa } = {　left:99 }　动态键必须搭配冒号

解构语法

const { a: { ins: **newa=be+lala** }, **b=99, [global]: c,　...z** } = {　a:{ left:99 },　b: 77,　c: 80,　d: 99, e:100　}　对象解构定义时，在左侧获取转移的值　（传递 null 会覆盖默认值）

当值为 undefined 时取默认值 99

支持动态键但必须搭配冒号

展开符获取剩余键值对对象

let newa=0, b, z;
({ a: { ins: newa=88 }, b=99, [global]: c, ...z } = { a:{ left:99 }, b: 77, c: 80, d: 99, e:100 }) 对象解构赋值时，需要外部添加括号，并提前定义变量

const [a, b=99, ...c] = [1, 2, 3, 4, 5, 6]; 数组解构定义时，在左侧获取转移的值
当值为 undefined 时取默认值 99
展开符获取剩余键值对对象

let a=0, b, c;
[a, b=99, ...c] = [1, 2, 3, 4, 5, 6]; 数组解构赋值时，需要提前定义变量

异步函数

async function a(){ async 异步函数，返回值是 promise 对象，返回其他值时会自动用 Promise.resolve()包装（异步函数同步执行，只有遇到 await 才会等待）
    return new Promise()
}
async function a(){ await 只能在 async 函数内使用（await 只能让内部代码等待，外部同步代码依旧执行）
    await new Promise((resolve,reject)=>{ reject('error') }) console.log(err) 遇到 Promise 对象时，等待 pending 过去，获取 fulfilled 状态的结果值
} 非 Promise 对象时，相当于 return 1，但不会终止函数
a().catch(console.log) 等待获得 rejected 状态时将不会执行后方代码，返回 rejected 状态新 promise 对象，可以在外部 catch 捕捉错误信息

async function a(){
    try { async 内部的 try 可以捕捉 await 的 rejected 状态，并继续执行后方代码
        await new Promise((resolve,reject)=>{ reject('error') })
    }catch(err){
        console.log(err)
    }
}

匿名函数

(v1, v2) => { a++,b++ } v => a++ v =>(v+2) v =>{ a++ } 箭头函数没有构造器，不能创建对象 （只有函数的大括号有函数作用域， 大括号对象或普通语句 没有函数作用域，this 即为外部对象）
function Person(){ this.fun=()=>this } per.fun() -->Person 箭头函数的 this 是固定的，指向定义时的作用域
per.newfun=()=>{} per.bfun() --> window
obj={ pro:func(){ return this } } obj.pro.func() --> window
const func =(a,b) => (c, d) => { } func (1,2) ('x','x') 多参时，从右向左编译，第一个函数调用返回 第二个函数的地址

错误捕捉

try{ throw "a"; throw err;
}catch(err){ console.log(err) catch 捕捉 try 内部抛出的错误并处理错误消息
}finally { console.log(err) } finally 无论 catch 有没有捕捉到错误都会执行的代码，不接受任何参数

Promise

new Promise((resolve, reject)=>{ resolve(data) reject(err) }) 创建时内部代码会立即执行，首次必须调用 resolve 或 reject（没有调用 resolve，reject 时，后方 then 不会被执行）

{<rejected>: "resultData"} == reject(resultData) Promise 对象有 **pending** **fulfilled** **rejected** 三种状态，并伴随有结果数据

pending 可能变为 fulfilled，而 fulfilled 和 rejected 状态不可变

Promise.**resolve**(1) Promise.resolve(thenobj)===pobj 返回一个 fulfilled 的 promise 对象，结果数据为括号内的参数

Promise.**reject**(1) Promise.reject(thenobj)===pobj 返回一个 rejected 的 promise 对象，结果数据为括号内的参数

Promise.**all**([p1, p2, p3]) {<fulfilled>: ["data1", "data2", "data3"] } 将多个 Promise 对象包装成一个新的 Promise 对象，只要有一个失败则返回最先 rejected 状态的值，成功的时候返回 fulfilled 结果数组。

Promise.**race**([p1, p2, p3]) {<fulfilled>: ["data1"] } 返回最快的结果，无论 fulfilled 或 rejected

pobj.**then**(data=> p2, err=>null).**then**(data=>null) then 接收数据为 promise 对象时，**pending** 状态会持续等待，**fulfilled** 和 **reject** 状态，分别执行 resolve 和 reject 处理函数获取结果数据（then 不会等待内部异步代码）

resolve 和 reject 处理函数都可以向后方传递 promise 对象 （没有返回 promise 对象时，默认返回 fulfilled 状态的 return 值）

then 内部出现 throw 'err' 或 非致命错误时，会将抛出的错误作为参数传递给后方的失败处理函数

pobj. **then**(data=> p2).**catch**(err=>null).**then**(data=> null) catch 作为 reject 处理函数，接收前方没有 reject 处理函数的 rejected 状态（rejected 状态没有遇到 reject 处理函数时回向后传递）

没有触发 reject 处理函数时，默认原样返回

pobj. **then**(data=> p2).**finally**(data=>null).**then**(data=> null) finally 不接收任何参数，等待前方 promise 结束后 执行内部代码，并将其原样返回

声明类

```
class Person {                                  class 不能像普通函数一样被调用，类名值是一个地址，能够被 return 和赋值
                                                class 不会被预解析
                                                class 包含一些函数内置属性 name, call, apply, bind，但类不能被 call 等调用
    constructor(name, age){                     构造器函数===Person.prototype.constructor===per.constructor，默认返回 this 实例对象（构造函数执行时，内部属性已经都被赋值了）
        console.log(new.target)                 返回执行当前构造函数的类地址，会显示子类地址，能访问子类普通函数成员或静态成员（赋值成员只有调用构造函数时才能访问）
        this.name=name;
        this.age=age;
        this.func =this.func.bind(this);        锁死 this，达到箭头函数一样的效果
    }
    #eyes='zz'                                  私有成员，只能在内部使用 this.#eyes，也可以设置 get eyes 和 set eyes 方法，但会让其在外部能够被访问 per.eyes
    static info='xx'                            静态成员，能通过类名 Person.info 直接访问，但无法被继承
    func(){                                     类的方法默认添加到 Person.prototype 中，Object.assign 方法可以向类添加多个方法，可以覆盖构造函数代码，但无法改变构造函数的默认返回值（会导致实例所有属性丢失）
        console.log("inside");                  内部有 this 时，func 作为普通函数被单独解构出去调用时会出错
    }
    [GlobalName]=999                            外部变量定义成员名称
    get name (){                                数据拦截，获取 age 时触发
        return this.age;
    }
    set age(value){                             数据拦截，设置 age 时触发，获取即将赋值的值
        this.age =value;
    }
```

```
}
class Son extends Person{
    constructor(name,age){
        super(name);                    调用父类构造函数，必须在 this 之前使用
        this.age=age;
        super.func()                    调用父类普通函数，不能用于获取父类被赋值成员，必须在 super()之后使用 （赋值成员只有调用构造函数时才能访问）
    }
}
```

暴露模块

```
export var a= 'xxx'
export {kkk as b, zzz as c, d}
export function e(){}
    普通暴露成员，export 只能这三种写法，可多个 export（as 前的变量无法接收）
export default 777
export default { a:'dong',  b:999 }
export default [a:'dong',  b:999 ]
export default function(){}
    默认暴露成员（多个会报错，不能使用 var let  as）

module.exports = 777
module.exports = { a:'dong',  b:999  }
module.exports = function(){}        CommonJS 规范，暴露所有成员（module.exports 多个会覆盖）

exports.a= 888;        exports.default = 'feng';                    CommonJS 规范，，自动转换为 module.exports （ {a:'d', default:'feng'} ）
```

导入模块

```
import test from './test'          test.b    test()        导入 default 成员          (没有 export default 为 undefined，不能导入 module.exports 或 exports.default)
import test, { a,b,c,d as xxx }  from './test'            导入 default 和 其他成员   （as 重命名 ）
import * as test from './test'                            导入所有成员               (等同 require 结果)

import styles './index.css'                                          导入 css 文件      (不需要* as，导入的 default 已经包含了所有的类了)
import logo from '../../assets/logo.png'      <img src={ logo }>     导入图片           (类型为 Object，但不是 Image 的对象)

var test=require('./test')      导入模块所有成员（可以是 export 或 exports 或 module.exports）   【默认导入顺序 xxx.js，xxx.json，xxx.node，package.json 内 main 指定的入口文件，index.js，node_modules， 全局 node_modules 】
require('./x')();               可以直接执行 module.exports 导出的函数：

require.context(  './',  false,  /.js$/  )          1. 读取文件的路径 2. 是否遍历文件的子目录 3. 匹配文件的正则表达式 （webpack 内使用）
require(['moname'], ()=>{  moname.test(1,2);  });     AMD 规范，加载的模块数组，成功之后的回调函数
返回函数对象：     resolve：     是一个函数，它返回请求被解析后得到的模块 id。
                   keys：        也是一个函数，它返回一个数组，由所有可能被上下文模块处理的请求组成。
                   id：          是上下文模块里面所包含的模块 id. 它可能在你使用 module.hot.accept 的时候被用到
```

JavaScript / build-in libraries

输出

console.log(a, b, c);　　把内容输出在浏览器控制台中
console.dir( );　　可以显示对象的结构
console.time('name');　　开始一个计时器
console.timeEnd('name');　　结束一个计时器

Number

Number("33aa")　　转数字　含字母就为 NaN（严格）　　不属于 Number 对象时，new 的时候也会自动转换【Number({})==NaN　Number([])==0】
Number.MAX_VALUE　**Number**.MIN_VALUE　　数字类型最大值最小值
toFixed(2)　　四舍五入，保留 2 位小数

String

readonly length: number;　　字符串长度

fromCharCode(...codes: number[]): string;　　用 ASCII 码返回字符串（静态）【字符串可以比较 - ASCII 码存储：　48（0）　65（A）　90（Z）　97（a）　122（z）　len=75 】
charCodeAt(index: number): number;　　返回指定索引字符的 ASCII 编码
includes(searchString: string, position?: number): boolean;　判断某个成员是否存在
concat(...strings: string[]): string;　　和原字符串拼接 返回新的字符串（不修改原字符串）

indexOf(searchString: string, position?: number): number;　　[从 2 开始]　找该字符串的位置　返回正向索引，没找到返回-1
lastIndexOf(searchString: string, position?: number): number;　　从后往前找　返回正向索引，没找到返回-1

replace(searchValue: { [Symbol.replace](string: string, replaceValue: string): string; }, replaceValue: string): string;　　只会替换一个，返回一个结果字符串，通配符可以全部替换
replace(searchValue: { [Symbol.replace](string: string, replacer: (substring: string, ...args: any[]) => string): string; }, replacer: (substring: string, ...args: any[]) => string): string;　　replacer 函数将老字符串 substring 替换为返回值 string（不修改原字符串）

substring(5, [10])　substring(10, 5)　返回　**[5, 10)** 之间的字符串，索引从 0 开始
slice(5,[10])　　返回　**[5,10)**　的字符串　　（省略为 5 开始后的全部）　【slice 不支持第一个数大】
substr(5, [n])　　返回　**[5, 5+n-1]**　包含 5 开始的 n 个字符　（省略为 5 开始后的全部）
split('|', [4])　　通过删除 '|'　把原字符串分割　返回字符串数组（前面 4 个）　【支持字符串参数都支持正则　split(/\s+/)，匹配到的内容删除　/(\s)+/）匹配不删除】
match(/\s/)　　返回一个匹配数组　（全局模式会返回所有匹配的情况）
startsWith('str')　　是否以 str 开头　【str 内不能含有空格，空格无法比对】
endsWith('str')　　是否以 str 结尾
toLowerCase() = toLocaleLowerCase()　返回英文小写　　"DDD%#%(#%_TTT"　--->　"ddd%#%(#%_ttt"
toUpperCase() = toLocaleUpperCase()　返回英文大写
trim()　　返回去除字符串两端的空格的新字符串（不修改原字符串）【无返回值】
repeat(2)　　返回一个重复当前字符串 2 次的新字符串
charAt(2) ==str[2]　　返回指定索引的字符（0 至 length-1）　超出为空
padStart(10, "9")　　用所给定的字符串从开头填充到目标长度，默认填充空格，填充字符串超出部分截断（长度小于当前字符串时，返回原字符串）　//　"999".padStart(7,"abc")　==>　'abca999'

Array

Array.isArray(arr)　　判断是不是数组 返回 Boolean （静态，[] instanceof Array==true）　【数组的[]内可以装表达式 a[a.length-1-2]】【JSON 和 concat 都可以重构数组】

Array.from(new Set(arr) | {'0':'2','1':'4','2':'5', length: 3 }, val=>newVal)　　Set 集合，伪数组 变普通数组，并提供遍历（伪数组必须有 length 属性）

length　　数组的长度（清空数组 arr.length=0，[1,2,3] 长度为 3）

includes("tt")　　判断某个成员是否存在

concat(A, B, C)　　返回 当前与多个数组连接后的新数组 （不修改原数组）

indexOf("tt",[2])　　返回 该元素的索引 没有返回-1【 arr.indexOf()!=-1 】

lastIndexOf("tt")　　返回 该元素的索引 没有返回-1

unshift('aa',99,'kk')　数组前方加一个或多个元素 返回之后数组的长度　// [1,2,3] ---> ['aa',99,'kk', 1,2,3]

shift()　　数组前方删除第一个元素　　返回删除的这个值

push('aa',99,'kk');　　数组后方加一个或多个元素 返回之后数组的长度

push.apply([2,3]，[4,5])　　将后方数组元素追加到前方

pop()　　数组后方删除最后一个元素 返回删除的这个值【没有为 undefined】

forEach(**(val, ind,** [arr] **) =>{** **},** [obj])　　遍历数组（函数内 this 指向 obj，arr 为当前遍历的数组，ind 无法修改，splice 会删除后一个元素）【无返回值，区分遍历 renturn 和外部函数 return】

map(**(val, ind,** [arr] **) =>{** return val*2 **},** [obj])　　无序遍历数组，返回 由每次 return 的值 组成的新数组（没有 return 会添加 undefined，this 指向 obj，arr 为当前遍历的数组）

find(**(val, ind,** [arr] **) =>{** return val>2 **},** [obj])　　遍历数组，返回第一个满足条件的值

findIndex(**(val, ind,** [arr] **) =>{** return val>2 **},** [obj])　　返回满足条件的下标

filter(**(val, ind,** [arr] **)=>{** return val>20; **},** [obj])　　遍历数组，返回一个满足 return 条件的新数组【arr=arr.filter() 右侧比左侧先执行，长度为 0 不会执行遍历操作】

findIndex(**(val, ind,** [arr] **)=>{** return val>20; **},** [obj])　　返回一个满足条件的元素索引

reduce(**(prev, val,ind,** [arr] **){** return prev+val **},** [init]);　　有初始值 prev=init 从 0 开始遍历，无初始值 prev=arr[0]从 1 开始遍历（prev 表示上一次遍历的返回值，自定义返回值）

不能在中途直接通过地址 prev 操作堆内的数据，直接 return prev.concat()

every(**(val, ind,** [arr] **)=>{** return val>20; **},** [obj])　　所有值满足 return 条件，返回 true【所有 val 满足条件时 函数终止，可以用 if 多条件判断】

some(**(val, ind,** [arr] **)=>{** return val>20; **},** [obj])　　有一个值满足 return 条件，返回 true【从前向后遍历，当前 val 满足条件时 函数终止】

sort()　　升序排列**当前数组**，返回当前数组 【不能对负数排序（-1 -2 -3）　必须传入排序函数 function(a,b){ return a-b } sort 中参数大于 0，交换 a b 顺序，a 在后面，sort()中参数小于 0，a b 顺序不变，a 在前面】

join("|")　　返回 元素间用"|"连接成字的符串（数组只有一个元素时，仅返回这个元素的字符串）

reverse()　　反转**当前数组**，返回当前数组

slice(5, 10)　　返回 [5,10) 的元素组成的数组，如果是负数，则表示从数组尾部开始算起，-1 指最后一个元素，-2 指倒数第二个元素（超出范围的部分不会返回，1, 1 相同索引范围返回空数组）

splice(2, 0, 99, 88, 77) 包括索引 2 开始　删除**当前数组**内 0 个元素　在索引 2 前方插入元素 99 88 77（省略删除后方全部）返回被删除的元素组成的数组

fill('s', 0, 5)　　填充数组，[0, 5）范围内的值，返回填充后的数组 【new Array(10).fill(0)】

Math

Math.PI　　π

Math.E　　e

Math.abs(-9)　　绝对值（null 时为 0）

Math.random()　　返回一个 [0, 1）的伪随机小数　　表示任意范围：乘范围大小+最小值　　（每次调用都是不同的值）

Math.**ceil**(3.5)　　向大取整

Math.floor(3.5)　　向小取整-4.5　-5

Math.round(3.5)　　四舍五入　　Math.round(5.5)==6　　Math.round(-5.5)==-5

Math.max(3, 4, 5)　最大值

Math.min(3, 4, 5)　 最小值

Math.pow(x, y)　　返回 x 的 y 次方

Math.sqrt(x)　　根号 x

Math.sin(33)　　正弦

Object

Object.keys(obj);　　获取 全部由对象的键 组成的数组　（不会获取 Symbol 键）

Object.getOwnPropertyDescriptor(obj, 'name')　　获取指定属性描述符对象（相当于 definedProperty 的属性描述参数）

Object.getOwnPropertyNames(obj)　　获取 全部由对象的 String 键 组成的数组　　（默认都会转换为 String，除了 Symbol）

Object.getOwnPropertySymbols(obj)　　获取　全部由对象的 Symbol 键 组成的数组

Object.create(Person.prototype, {name:{value: 888 }})　返回一个实例对象，指定了__proto__，并可以追加一些属性，实现继承（Son.prototype=Object.create，类似于 new，但 new 有继承缺点）

Object.assign(a, b)　　将后方所有对象的键值 移动到 a 上　（键重名覆盖）

Object.freeze(obj)　　冻结一个对象，不能修改其成员

Object.defineProperty(obj, key, {　　为对象定义新属性或者修改现有属性，返回修改后的新对象（第三个参数为属性描述符）

　　value: 'xx'　　属性值

　　set:　val=>{ temp=val }　　setget 均为只读，重名会覆盖普通属性【设置 obj.key 触发 set，获取 obj.key 触发 get】

　　get:　()=>{ return temp }

　　writable: false,　　是否 value 能够被赋值改变（再次赋值时会失败，不会报错）

　　enumerable: false,　　是否该属性才会出现在对象的枚举属性中

　　configurable: false,　　是否该属性的描述符能够被改变，同时该属性也能从对应的对象上被删除（删除属性时会失败，不会报错）

})

Object.getOwnPropertyDescriptor(obj, "name")　　获取指定对象的自身属性描述符，没有此属性会返回 undefined　// {value: '999', writable: true, enumerable: true, configurable: true}

Reflect.ownKeys(obj)　　返回全部由对象的键 组成的数组　（会获取 Symbol 键，并不会去获取那些继承的键）

obj.hasOwnProperty(key)　　是否含有这个键（不判断原型对象 false）

Person.prototype.isPrototypeOf(duck);　　验证是否是自己的对象

Date

Date 对象　　var dt = new Date(datastr);　　当前服务器的时间"2017-08-12" "2017/08/12" 日期之间不能计算　（ 一般月份 31 天，4，6，9，11 是 30 天。　2 月 闰年是 29 天,一般年份是 28 天 ）

　　Date(毫秒)　毫秒数变日期

　　Date.now()　获取当前 毫秒数时间戳　// 1618972237921

　　Date.parse(date)　　获取时间戳

　　　　valueof()　　获取时间戳

　　　　getTime()　　获取时间戳

　　　　setYear()　getYear()

　　　　getFullYear()　　年份　　//2020

　　　　getMonth()+1　　月份　　(0-11 代表一年的月份，需要加 1)

　　　　getDate()　　一月的第几天　　//16

　　　　getHours()　　小时

getMinutes()　分钟
getSeconds()　秒数
getDay()　星期几（0-6）
toString()　转换字符串（输出 dt 默认调用）
toDateString()　Mon　Dec 17 2018
toGMTString()　标准时间格式
toLocaleDateString()　12/17/2018
toTimeString()　21:37:30
toLocaleTimeString()　9:36:51 PM

Image

Var img=new Image();　==　document.createElement('img')
img.onload=function(){}　图片加载完毕事件
img.src = imgurl;
img.setAttribute('crossOrigin', 'anonymous');　防止跨域违规（只能在 Image 创建的 dom 中使用）

JSON

JSON 数据　（成对的键值，键和值都用双引号，json 文件内不能写注释）　var j= {"name":"先", "age":"10" };
JSON.parse('str')　**j 串**转换成 数组或对象　【当转换的是对象时，后面不能接数组方法　JSON.parse().forEach()】
JSON.stringify(json, (key, val)=>{}, [' name'],　1 ）　数组或对象 转换成**j 串，**[' name'] 为最后保留的键，　1 为控制输出时字符串里面的间距【函数会丢失，Date 会变成字符串，RegExp 和 Error 对象会变成{}】

JSON.stringify(obj, array) 将数组内的值当作键　在 obj 中找到含有此键的数据，转换成 j 串

Error

name　错误名　（Uncaught errname: errmsg）
message　显示信息

RegExp

var tobj=/\d{5}/;　字面量创建
var tobj=new RegExp(/\d{5}/gi);　var tobj=new RegExp("\\d{5}")　new 创建
/\d{5}/.**test**(str)　验证是否匹配，返回 Boolean
/\d{5}/.**exec**(str)　返回一个匹配数组　（全局模式/g　不会重复匹配，需要多次执行直到为 null，再执行则重新开始匹配）
// ['2022/1/14', index: 0, input: '2022/1/14 22:05', groups: undefined]
RegExp.**$3**　返回当前第 3 组正则表达式匹配结果　tobj.test　str.match 都会触发

Symbol

let sym1 = Symbol('xx'),　sym 2 = Symbol('xx');　sym1 =x= sym2　symbol 类型具有唯一性，同样传入的参数 描述信息但值不相等　（ Symbol 不能使用 new ）（Symbol 不能自动被转换为字符串，拼接字符串会报错）
let obj = {　[sym1]: 'val'　};　obj[sym1]　对象使用和访问 symbol 属性时都只能通过[]，不能使用 obj.sym1 访问　【 for·in，object.keys() 无法访问到 symbol 键值 】
let sym1 = Symbol.for('name'),　sym2= Symbol.for('name');　sym1 === sym2　使用同一个全局 symbol 值，无值新建，有值返回
Symbol.keyFor(sym1) === sym2　获取全局 Symbol 值

Map（ES6）

let ma = new Map(　[　['m1', 95], ['m2', 75], ['m3', 85]　]　);　map 是一组键值对的结构，具有极快的查找速度（不允许重复项）
size　获取大小
set('m4', 67)　添加新的键值对（同名覆盖）
get('m1')　获取一个键的值
has('m1');　是否存在键
delete('m1');　删除键

Set（ES6）

let se= new Set([1, 1, 2])　有序集合，自动去掉完全相同的数据，数据类型可不同　Set 的键和值相同　支持 for-in，for-of，不支持数组内置

let se= new Set(　[...a, ...b])　并集　[1,2,3]　[4,3,2] ==> {1, 2, 3, 4}

let se= new Set(　[...a].filter(x => b.has(x))　)　交集　[1,2,3]　[4,3,2] ==> {2, 3}

let se= new Set(　[...a].filter(x => !b.has(x))　)　不包含　[1,2,3]　[4,3,2] ==> {1}

let arr= [...se]　将集合变成普通数组

- size　获取大小
- add(1)　添加数据，已经存在会不添加
- delete(2)　删除数据
- has(1)　判断数据是否存在
- clear()　清空集合
- forEach()　遍历 set
- keys()　返回一个键名的遍历器　【 支持 for-of 遍历，for-in 无效：　for(let val of set.values())　==>　[1, 1]　[2, 2] 】
- values()　返回一个键值的遍历器　【 for(let val of set.values())　==>　1　2】
- entries()　返回一个键值对的遍历器

Uint8Array

new Uint8Array();　数组类型表示一个 8 位无符号整型数组，创建时内容被初始化为 0。创建完后，可以以对象的方式或使用数组下标索引的方式引用数组中的元素。

new Uint8Array(23);

new Uint8Array(typedArray);

new Uint8Array(object);

new Uint8Array(buffer, [byteOffset]　, [length]　);　8 位无符号整数值的类型化数组。内容将初始化为 0。如果无法分配请求数目的字节，则将引发异常。

BigInt

BigInt.asIntN()

将 BigInt 值转换为一个 -2^width - 1 与 2^(width-1) - 1 之间的有符号整数。

BigInt.asIntN(width, bigint);

BigInt.asUintN()

将一个 BigInt 值转换为 0 与 2^width-1 之间的无符号整数。

BigInt.asUintN(width, bigint);

typeof bobj === 'bigint'