<script type='application/javascript' src=''></script>

text/javascript 过时了

null转换成boolean是false

{} []转换成boolean是true

变量名以字母、下划线或者$开头

document.getElementById("myDIV").classList.add("mystyle");

document.getElementById("myDIV").classList.remove("mystyle");

myDiv.classList.contains('myCssClass'); //return true or false

全局变量：函数外声明的变量，称为全部变量

局部变量：函数内部使用var声明的变量，称为局部变量

在JS中，只有函数作用域，没有块级作用域！也就是说，if/for等有{}的结构体，并不能具备自己的作用域。

## foreach

let array=[2, 4, 23, {}]

for(let index **in** array) {

console.log(index,array[index]);

};

0 2

1 4

2 23

3 {}

for(let v **of** array) {

console.log(v);

};

2

4

23

{}

## Number

var x=0.2+0.1; //0.30000000000000004

0.4-0.3; //0.10000000000000003

整数最大2^53-1 9007199254740992

不能精确表示十进制小数

2.445.toFixed(2); //IE 2.45 chrome 2.44

判断obj是否为一个整数

function isInteger(obj) {

return Math.floor(obj) === obj

}

避免小数精度问题，要将小数转换成整数进行运算

## 定时器

每隔1000ms执行一次

setInterval(function(){

console.log("alf"):

}, 1000);

停1000毫秒执行，只执行一次

setTimeout(function(){

console.log("asd");

}, 1000);

window的方法,内部的this指向window

var time1=setInterval(function(){}, 1000);

clearInterval(time1);

time1是1,2,3这种值，代表一个定时器的编号

clearTimeout(time2)

setTimeout(func1, 1000, 'hello')

第三个及以后的参数将作为参数传递给func1

## json

{

"name" : "liu",

"age" : 19

}

json属性名必须加双引号

string🡪json

JSON.parse()

json🡪string

JSON.stringify()

var obj={

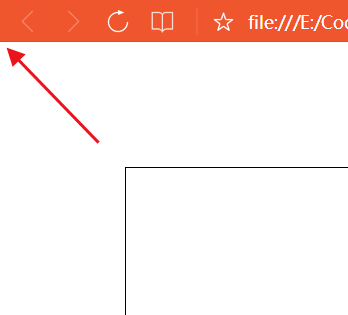
"name" : "liu",

"age" : 12

};

JSON.stringify(obj);

## Event事件

* clientX,clientY：鼠标相对于浏览器窗口可视文档区域的左上角的位置；可视区域不包括工具栏和滚动条。就是相对于这个点的位置
* pageX,pageY (x,y)：鼠标相对于文档左上角的位置，而不是可视区域。因为有可能有滚动条，所以pageX>clientX。经测试page=x，但x, y不推荐使用
* offsetX,offsetY：鼠标相对于srcElement左上角的位置
* layerX,layerY：鼠标相对于最近的有定位的元素左上角的距离
* screenX、screenY：鼠标相对于显示器左上角的位置
* type：事件的类型，如click；
* srcElement/target：事件源，就是发生事件的元素；

jquery只能用target

* button：声明被按下的鼠标键，整数，1代表左键，2代表右键，4代表中键，如果按下多个键，酒把这些值加起来，所以3就代表左右键同时按下；（firefox中 0代表左键，1代表中间键，2代表右键）(没看出来)
* altKey,ctrlKey,shiftKey等：返回一个布尔值；
* keyCode：返回keydown何keyup事件发生的时候按键的代码，以及keypress 事件的Unicode字符；(firefox2不支持 event.keycode，可以用 event.which替代 )
* fromElement,toElement：前者指mouseover事件中鼠标移动过的文档元素，后者指代mouseout事件中鼠标移动到的文档元素；
* cancelBubble：一个布尔属性，把它设置为true的时候，将停止事件进一步起泡到包容层次的元素；(e.cancelBubble = true; 相当于 e.stopPropagation();)
* returnValue：一个布尔属性，设置为false的时候可以组织浏览器执行默认的事件动作；(e.returnValue = false; 相当于 e.preventDefault();)
* attachEvent(),detachEvent()/addEventListener(),removeEventListener：为制定 DOM对象事件类型注册多个事件处理函数的方法，它们有两个参数，第一个是事件类型，第二个是事件处理函数。在attachEvent()事件执行的时候，this关键字指向的是window对象，而不是发生事件的那个元素。attachEvent()在IE11以下版本使用

firefox里的event跟IE里的不同，IE里的是全局变量，随时可用；firefox里的要用参数引导才能用，是运行时的临时变量。

在IE/Opera中是window.event，在Firefox中是event；而事件的对象，在IE中是 window.event.srcElement，在Firefox中是event.target，Opera中两者都可用。

var evt = evt || window.event; // firefox下window.event为null, IE下event为null

事件先捕获后冒泡

捕获：从根节点（body）到div

冒泡：从div到根节点

e.stopPropagation()阻止冒泡

* && 操作符特点：只要一个是false就返回，都是true，返回最后一个
* || 操作符特点：只要一个是true就返回，都是false，返回最后一个。

在js逻辑运算中，0、""、null、false、undefined、NaN都会被判断为false

&&和||都是：当运算到某一个变量就得出最终结果之后，就返回哪个变量。

注意：返回的不是True或false，是变量的值

* 立即执行函数( function(){…} )()和( function (){…} () )两种写法都行

通过定义一个匿名函数，创建了一个新的函数作用域，相当于创建了一个“私有”的命名空间，该命名空间的变量和方法，不会破坏污染全局的命名空间

**let** 所声明的变量，只在let命令所在的代码块内有效

for (let i = 0; i < 10; i++) {

// 以后for循环用let吧

}

var b=[];

for(var i=0; i<10; i++){

b[i]=function(){

console.log(i);

}

}

b[6]()输出10

var a = [];

for (let i = 0; i < 10; i++) {

a[i] = function () {

console.log(i);

};

}

a[6]()输出6

和这样写效果一样

var b=[];

for(var i=0; i<10; i++){

(function(m){

b[i]=function(){

console.log(m);

}

})(i);

}

var t1;

let t2;

console.log(t1);

console.log(t2);

输出

undefined

undefined

console.log(t1);

报错

Uncaught ReferenceError: t1 is not defined at ttt.html:13

console.log(t1);

console.log(t2);

var t1;//var如果没赋值，定义在前面或者后面没有关系

let t2;//let定义在后边会报错

输出

undefined

ttt.html:14 Uncaught ReferenceError: t2 is not defined at ttt.html:14

<h1 id="nihao">

<p>hello world</p>

</h1>

var a=document.getElementById('nihao');

* console.log(a)

<h1 id="nihao">

<p>hello world</p>

</h1>

* a.innerText

"hello world"

a.innerText='<em>你好世界</em>'

改变内容

* a.innerHTML

"<p>hello world</p>"

* parentNode

返回父元素

节点类型

* 元素节点
* 属性节点
* 文本节点
* 注释节点

<h1 id="a">

<p>hello world</p>

</h1>

* a.childNodes

输出这个数组NodeList(3) [text, p, text]

0:text

1:p

2:text

length:3

\_\_proto\_\_:NodeList

* a.nextSibling
* a.previousSibling

a里 两个文本节点（空文本节点）一个元素节点

以上三个方法返回节点

* a.parentElement
* a.children
* a.firstElementChild
* a.lastElementChild
* a.nextElementSibling
* a.previousElementSibling

以上六个方法返回元素节点

javascript对象是基于原型prototye-based的，而java C++是基于类class-based的

js通过一个prototype对象创建更多的对象

{

f1: function(){

console.log('aaa')

}

}

和

{

f1(){

console.log('aaa');

}

}

是一种写法

## prototype 原型

函数（除了一些内建函数）都有一个名为prototype（原型）的属性，这个属性是一个指针，指向一个对象，而这个对象的用途是，包含可以有特定类型的所有实例共享的属性和方法。prototype是通过调用构造函数而创建的对象实例的原型对象。

hasOwnProperty()判断指定属性是否为自有属性；in操作符对原型属性和自有属性都返回true。

var obj = {a: 1};

obj.hasOwnProperty("a"); // true

obj.hasOwnProperty("toString"); // false

"a" in obj; // true

"toString" in obj; // true

示例：鉴别原型属性

function hasPrototypeProperty(obj, name){

return name in obj && !obj.hasOwnProperty(name);

}

\_\_proto\_\_

对象具有属性\_\_proto\_\_，可称为隐式原型，指向构造该对象的构造函数的原型，这也保证了实例能够访问在构造函数原型中定义的属性和方法。

var Boo = {name: "Boo"};

function Foo(){}

Foo.prototype = Boo;

var f = new Foo();

Object.getPrototypeOf(obj)

使用原型对象的好处是可以让所有对象实例共享它所包含的属性和方法

obj.isPrototypeOf(obj) 方法检查某个对象是否是另一个对象的原型对象

function F1(){

}

var obj1=new F1();

* F1.prototype

返回 一个对象

{

constructor:ƒ F1()

\_\_proto\_\_:Object

}

这就是典型的，一个由构造函数创建的对象的prototype。除了自带一个constructor属性之外就是个空对象。现在我们可以往这个prototype里面加东西A。以后所有F1构造出来的对象都有东西A

* obj1.\_\_proto\_\_

返回 上边那个对象

对象.\_\_proto\_\_=构造器.prototype

* obj1.\_\_proto\_\_.\_\_proto\_\_

返回null

* obj1.constructor

返回函数F1

实际上返回的是obj1.\_\_proto\_\_的属性

* 对象没有prototype，函数的\_\_proto\_\_没有意义

F1.\_\_proto\_\_

返回ƒ () { [native code] }

出问题了。函数的\_\_proto\_\_也是有意义的

F1.\_\_proto\_\_=Function.prototype

而这个Function.prototype 是一个function

基于原型创建多个对象

* 定义原型对象
* 定义对象的构造函数
* 将构造函数关联到原型
* 实例化对象

var person={

name: "aaa",

age: 12

}

function Person(a, b){

this.name=a;

this.age=b;

}

Person.prototype=person;

var p1=new Person('nihao', 23);

这个new看起来违背了基于原型的思想，来自原型的Object.create登场

var p2 = Object.create(person);

p2.name="nihao";

p2.age=23;

这样写又重复了，于是

function cons(a, b){

var obj=Object.create(person);

obj.name=a;

obj.age=2;

return obj

}

var p3=cons("nihao", 23);

变量和函数**提升**

javascript引擎进入作用域时对代码进行两轮处理

1.

* 声明并初始化函数参数
* 声明局部变量(undefined)
* 声明并初始化函数

2.执行代码

所以会出现

情况1

var a=11;

function f1(){

console.log(a);

var a;

}

输出undefined，因为里边的a在第一轮处理时就声明，但并未赋值

情况2

var a=11;

function f1(aa){

console.log(aa);//11

var aa;//这行相当于啥也没干。如果是var aa=22, 则arguments[0]也变成22

console.log(aa);//11

}

f1(a);

* 任何数据类型+字符串都=字符串
* 1+1+"a"+1+1="2a11" 从左向右执行
* 1 \* "3" = 3
* "6" - "4" = 2
* 1/0 Infinity
* 0/0 NaN
* NaN==NaN false

Undefined===Undefined true

null===null true

Undefined==null true

Undefined===null false

* undefined, NaN, null, "", 0, false转换成布尔值都是false
* window.prompt 可提示用户进行输入的对话框
* typeof null 🡪 object

typeof [213, 3] 🡪 object

* 沿着函数作用域链向上查找变量
* 沿着对象原型链向上查找属性

js执行会阻塞DOM树的解析和渲染

css加载不会阻塞DOM树的解析

css加载会阻塞DOM树的渲染

css加载会阻塞后面js语句的执行

## 变量类型

* 基本数据类型 number string boolean undefined null

保存在栈内存中

var a= 1;

操作的是变量实际保存的值。

基本类型变量的复制：在栈中创建一个新值，然后把值复制到为新变量分配的位置上。

* 引用数据类型：object array function

保存在堆内存中

也就是，变量中保存的实际上的只是一个指针，这个指针指向堆中的一个位置，该位置保存着对象。访问方式是按引用访问。

操作时，需要先从栈中读取内存地址，然后再延指针找到保存在堆内存中的值再操作。

引用类型变量的复制：复制的是存储在栈中的指针，将指针复制到栈中未新变量分配的空间中，而这个指针副本和原指针指向存储在堆中的同一个对象；复制操作结束后，两个变量实际上将引用同一个对象。改变其中的一个变量的值，将影响另一个变量。

声明object

var aaa={name: "xiaolin", age: 18}

var aaa=new Object();

aaa.age=18;

* symbol

JavaScript的第七种数据类型

* 对象的属性名现在可以有两种类型，一种是原来的字符串，另一种就是新增的 Symbol 类型。
* 凡是属性名属于 Symbol 类型，就都是独一无二的，可以保证不会与其他属性名产生冲突。

let s=**Symbol**(); //通过Symbol()函数生成symbol，注意没有new

let s1=Symbol();

typeof s; // 'symbol'

s==s1; //false

Symbol函数可以接受一个字符串作为参数，表示对 Symbol 实例的描述，主要是为了在控制台显示，或者转为字符串时，比较容易区分

let s2 = Symbol('foo');

想读取的话显式toString()

s2.toString() // 'Symbol(foo)'

作为属性名

a[s] = 'Hello!';

let a = { [s]: 'Hello!' };

let a={ [s](param){console.log(param)} }; //结合增强对象写法的花式写法

这玩意可以当枚举使？

const log = {};

log.levels = {

DEBUG: Symbol('debug'),

INFO: Symbol('info'),

WARN: Symbol('warn')

};

console.log(log.levels.DEBUG, 'debug message');

console.log(log.levels.INFO, 'info message');

Symbol 作为属性名，该属性不会出现在for...in、for...of循环中，也不会被Object.keys()、Object.getOwnPropertyNames()、JSON.stringify()返回。但是，它也不是私有属性，有一个Object.getOwnPropertySymbols方法，可以获取指定对象的所有 Symbol 属性名

Object.getOwnPropertySymbols(a); //[Symbol()]

Object.getOwnPropertyNames(a); //["rq"]

Reflect.ownKeys(a); //["rq", Symbol()] 这个可以返回所有，包括Symbol的属性名

Symbol.for方法它接受一个字符串作为参数，然后搜索有没有以该参数作为名称的 Symbol 值。如果有，就返回这个 Symbol 值，否则就新建并返回一个以该字符串为名称的 Symbol 值。

let s1 = Symbol.**for**('foo');

let s2 = Symbol.for('foo');

s1 === s2 // true

## 类型检查

**typeof** true; //'boolean'

typeof null; //'object'

typeof undefined; //'undefined'

typeof (()=>{}); //'function'

typeof Symbol(); //'symbol'

* object instanceof constructor

instanceof运算符用来检测 constructor.prototype 是否存在于 object 的原型链上

非object不能用instance检测

let x1=3;

let x2=new Number(3);

typeof x2; //"object"

x1 instanceof Object; //false

x2 **instanceof** Number; //true

x2.\_\_proto\_\_ instanceof Object; //true

Number instanceof Object; //true 不是因为number是object的实例，而是因为Number是一个函数，函数是Object的实例

[] instanceof Object; //true 总之右边如果是Object，则总为true。

理解这句

F1.\_\_proto\_\_==Function.\_\_proto\_\_ == Function.prototype

Function.\_\_proto\_\_.\_\_proto\_\_==Function.prototype.\_\_proto\_\_==Object.prototype

F1.\_\_proto\_\_.\_\_proto\_\_== Object.prototype

Function instanceof Object; //true

Object instanceof Function; //true

let F1=()=>{};

F1 instanceof F1; //false 右边F1的prototype是一个对象。左边F1找的是\_\_proto\_\_,是一个函数，等于Function.prototype，所以不再原型链上

F1 instanceof Function; //true

js 在底层存储变量的时候，会在变量的机器码的低位1-3位存储其类型信息

* 000：对象
* 010：浮点数
* 100：字符串
* 110：布尔
* 1：整数

对于 undefined 和 null 来说，较为特殊

* null：所有机器码均为0
* undefined：用 -2^30 整数来表示

所以，typeof 在判断 null 的时候就出现问题了，由于 null 的所有机器码均为0，因此直接被当做了对象来看待。

**Object.prototype.toString**.call(1) // "[object Number]"

Object.prototype.toString.call('hi') // "[object String]"

Object.prototype.toString.call({a:'hi'}) // "[object Object]"

Object.prototype.toString.call([1,'a']) // "[object Array]"

Object.prototype.toString.call(true) // "[object Boolean]"

Object.prototype.toString.call(() => {}) // "[object Function]"

Object.prototype.toString.call(null) // "[object Null]"

Object.prototype.toString.call(undefined) // "[object Undefined]"

Object.prototype.toString.call(Symbol(1)) // "[object Symbol]"

## 类型转换

var m= Number('456');

var m2=parseInt('123abc234feaw');//123

var m3=Boolean('{}'); //true

## window.onload

window.onload=function(){

alert('hello');

}

$(window).load(function(){});

$(document).ready(function(){

})

$(function(){}) //上一种方法的简写

等待文档加载完毕后执行

等待document只需要整个DOM结构加载完，等待window要等待图片等等

ducument.querySelector()

返回选中的第一个

document.querySelectorAll()

返回一个数组

## Date

GMT Greenwich Mean Time 格林尼治标准时间/世界时

UTC Coordinated Universal Time 协调世界时/世界协调时间

UTC是经过平均太阳时(以格林威治时间GMT为准)、地轴运动修正后的新时标以及以「秒」为单位的国际原子时所综合精算而成的时间，计算过程相当严谨精密，因此若以「世界标准时间」的角度来说，UTC比GMT更加精准。其误差值必须保持在0.9秒以内，若大于0.9秒则由位于巴黎的国际地球自转事务中央局发布闰秒，使UTC与地球自转周期一致。

大多数情况，UTC和GMT一样

地球二十四个时区，经度15度，差1小时。北京东八区

北京时间=GMT时间+8小时

如果GMT 2019.05.13 0:00，那么北京时间2019.05.13 8:00, 东十一区2019.05.13 11:00, 西十一区2019.05.12 13:00，按这样算，东十二区是5月13号中午12点，西十二区是5月12号中午12点，真的就是这样。其实，东西十二区是一个时区，加起来15度，时刻相同但是日期不同。中间有条线叫日界线，从东到西日期加一天，从西到东日期减一天。

var aaa=new Date();

console.log(aaa);

Mon May 13 2019 11:24:06 GMT+0800 (中国标准时间)

getDate() 1 ~ 31 13

getDay() 0 ~ 6 周日是1！ 1

getMonth() 0 ~ 11 4

getFullYear() 2019

getHours() 0 ~ 23 11

getMinutes() 0 ~ 59 24

getSeconds() 0 ~ 59 6

getMilliseconds() 0 ~ 999 963，也就是当前时间24分06秒963毫秒

getTime() 自1970.01.01的毫秒数 1557718435471

getTimezoneOffset() 与GMT的分钟差 -480

aaa.toDateString() "Mon May 13 2019"

aaa.toTimeString() "11:40:35 GMT+0800 (中国标准时间)"

aaa.toUTCString() "Mon, 13 May 2019 03:40:35 GMT"

aaa.toLocaleString() "2019/5/13 上午11:40:35"

aaa.toLocaleTimeString() "上午11:40:35"

aaa.toLocaleDateString() "2019/5/13"

## Array

var a = Array(); // []

var a = Array(3); // [undefined,undefined,undefined]

var a = Array(3,11,8); // [ 3,11,8 ]

* concat() 连接数组

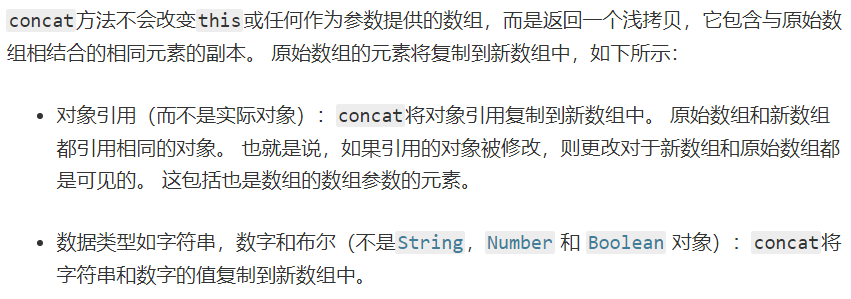
let a=[1, 2, 3]

a.concat(4, 5);

a.concat([6, 7])

a.concat([8], ['12', 're']) //会把数组拆成一项一项的连

连接的时候是浅拷贝，他们的成员都是对原成员的引用。



* join() 把数组的所有元素放入一个字符串。元素通过指定的分隔符进行分隔。
* pop() 删除并返回最后一个元素
* push() 向末尾添加一个或更多元素，返回新的长度。
* reverse() 颠倒顺序
* slice() 从某个已有的数组返回选定的元素
* sort() 排序

字符串不需要写排序函数，数字需要

function sortNumber(a,b){

return a - b

}

var arr = []

arr[0] = "10"

arr[1] = "5"

arr[2] = "40"

document.write(arr.sort(sortNumber))

* splice() 删除元素，并向数组添加新元素。
* toString()
* shift() 删除并返回第一个元素
* unshift() 向数组的开头添加一个或更多元素，并返回新的长度。
* valueOf() 返回数组对象的原始值

字符串🡪数组

**Array.from**('hello') // ["h", "e", "l", "l", "o"]

类数组🡪数组

function(x, y){

return Array.from(arguments);

}

# String

## 立即执行函数

var mmm = [];

for (var i = 0; i < 6; i++) {

mmm[i] = function() {

console.log(i)

}

}

这样写，mmm[3](100)也输出6，因为这个函数里边的i相当于是找全局的i

for (var i = 0; i < 6; i++) {

mmm[i] = function(i) {

console.log(i)

}

}

这样写，mmm[3](100)输出100

for (var i = 0; i < 6; i++) {

mmm[i] = (function() {

console.log(i)

})()

}

这样写，直接打印012345，而且mmm=[undefined, undefined, undefined, undefined, undefined, undefined]

## 深拷贝

function deepClone(obj){

let objClone = Array.isArray(obj)?[]:{};

if(obj && typeof obj==="object"){

for(key in obj){

if(obj.hasOwnProperty(key)){

//判断ojb子元素是否为对象，如果是，递归复制

if(obj[key]&&typeof obj[key] ==="object"){

objClone[key] = deepClone(obj[key]);

}else{

//如果不是，简单复制

objClone[key] = obj[key];

}

}

}

}

return objClone;

}

.concat(), .slice()方法都是浅拷贝，string，number，boolean复制一份副本，object，array，function指向同一个对象。

## defer async

<script src="path/to/myModule.js" **defer**></script>

<script src="path/to/myModule.js" **async**></script>

defer要等到整个页面在内存中正常渲染结束（DOM 结构完全生成，以及其他脚本执行完成），才会执行；async一旦下载完，渲染引擎就会中断渲染，执行这个脚本以后，再继续渲染。一句话，defer是“渲染完再执行”，async是“下载完就执行”。另外，如果有多个defer脚本，会按照它们在页面出现的顺序加载，而多个async脚本是不能保证加载顺序的。

## 装箱

* 基本类型转换成对应的引用类型称为装箱
* 引用类型转换为对应的基本类型称为拆箱

每当读取一个基本类型的时候，后台就会创建一个对应的基本包装类型对象，从而能够调用一些方法来操作这些数据。这就是**装箱**

var str='nihao';

str.substring(2); //'hao'

相当于这个过程

var n1 = new Number(23);

var n2 = n1.toFixed(2); //"23.00"

n1 = null;

而Number的prototype有以下方法

* toExponential: ƒ toExponential()
* toFixed: ƒ toFixed()
* toPrecision: ƒ toPrecision()
* toString: ƒ toString()
* valueOf: ƒ valueOf()

所以number基本类型也能使用这些方法

相比起来string基本类型可以使用的方法就有很多

* substr: ƒ substr()
* substring: ƒ substring()
* toUpperCase: ƒ toUpperCase()
* trim: ƒ trim()
* ...

**拆箱**

* 手动拆箱

numObj.valueOf();

strObj.toString();

* 隐式拆箱或自动拆箱

拆箱过程内部调用了ToPrimitive抽象操作

该操作接收两个参数，要转变的对象和PreferredType被期待转成的类型

PreferredType不是必须的，默认number

先调用valueOf方法看有没有返回值，如果没有则调用toString方法，如果toString也没有返回值则报错TypeError

let obj={

valueOf: ()=>{return 'invoke valueOf first'},

toString: ()=>{return 'invoke toString first'}

}

console.log(obj+' aaaa');//invoke valueOf first aaaa

Number(obj); //Uncaught TypeError: Cannot convert object to primitive value

如果使用String(obj)方法，PreferredType参数会传入string，则先调用toString()方法，后调用valueOf()，都不返回基本类型时报错

String(obj)

String(obj)

显示"invoke toString first"

但是不能理解为String(x)返回x.toString()

let obj={

valueOf: ()=>{return 23},

toString: ()=>{return {nihao:3}}

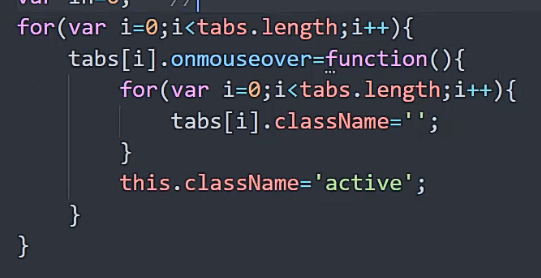
}

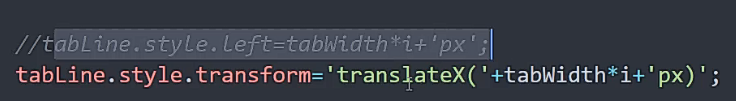
String(obj)

"23"

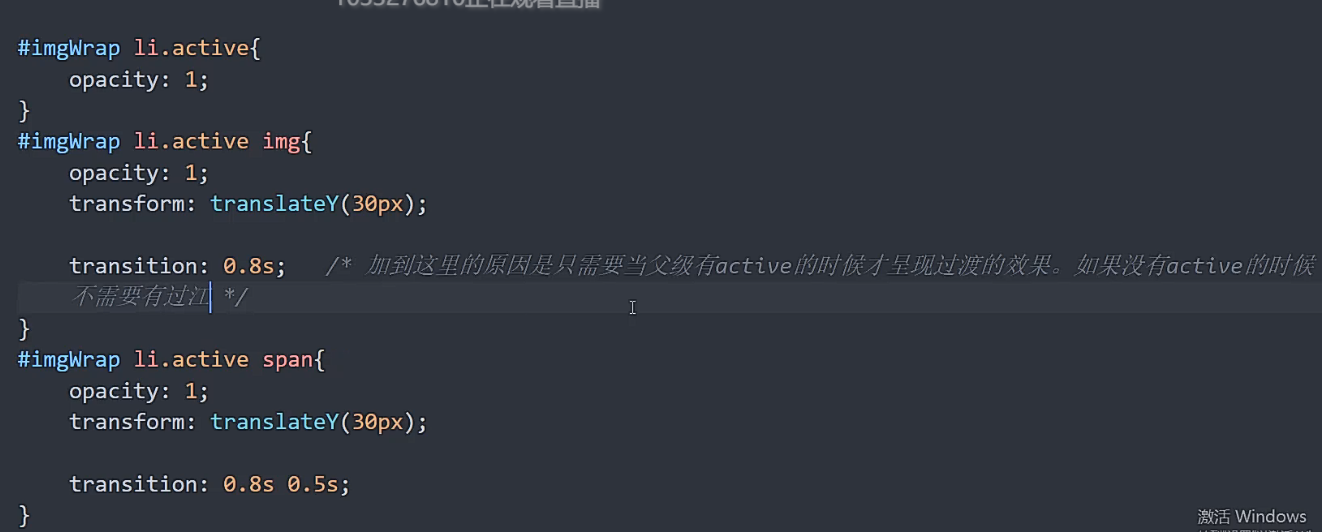
obj.toString()

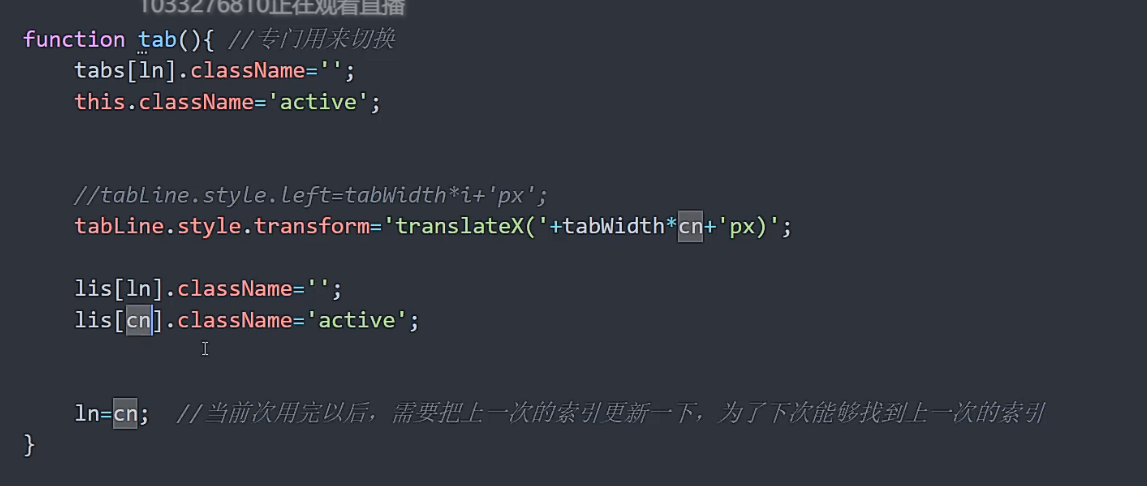
{nihao: 3}





offsetWidth





# ES6

**Promise**

我们先看一个最简单的Promise例子：生成一个0-2之间的随机数，如果小于1，则等待一段时间后返回成功，否则返回失败：

function test(resolve, reject) { //接收两个函数作为参数

var timeOut = Math.random() \* 2;

log('set timeout to: ' + timeOut + ' seconds.');

setTimeout(function () {

if (timeOut < 1) {

log('call resolve()...');

resolve('200 OK');//想当于把'200 OK' return给then当参数

}

else {

log('call reject()...');

reject('timeout in ' + timeOut + ' seconds.'); //return给catch作为参数

}

}, timeOut \* 1000);

}

有了执行函数，我们就可以用一个Promise对象来执行它，并在将来某个时刻获得成功或失败的结果：

var p1 = new Promise(test);

var p2 = p1.then(function (result) {

console.log( result);

}); //then的参数是一个函数，这个函数的参数是resolve的参数，resolve在Promise对象绑定的函数里

var p3 = p2.catch(function (reason) {

console.log( reason);

});

变量p1是一个Promise对象，它负责执行test函数。由于test函数在内部是异步执行的，当test函数遇到resolve(param1), 执行then(function(param1))

Promise对象可以串联起来，所以上述代码可以简化为：

new Promise(test).then(function (result) {

console.log('成功：' + result);

}).catch(function (reason) {

console.log('失败：' + reason);

});

(x) => x + 6

相当于

function(x){

return x + 6;

};

(obj) =>{

const a = 999;

return 99;

}

## 对象的解构赋值

let { first, second, third } = { second: "aaa", third: "bbb" };

first: undefined

second: "aaa"

third: "bbb"

let { foo: baz } = { foo: 'aaa', bar: 'bbb' };

baz: 'aaa'

foo: undefined

迭代器

next()方法返回一个结果对象

{

value: 444, //value返回下一个值

done: false //done是否没有更多数据可以返回。最后一个值返回后再next()，done=true

}

对象字面量

当属性与值的变量同名时。

const name = 'Jane';

const age = 20

// es6

const person = {

name,

age

}

// es5

var person = {

name: name,

age: age

};

计算对象键

var heat = '50%';

var field = 'rock';

var music = {

[field]: heat

}

console.log(music); // Object {rock: "50%"}

## unicode

'\u0061' 'a'

a的ascii码97, 16进制61

'\u5F62' "形"

但是，这种表示法只限于在\u0000~\uFFFF之间的字符，超过FFFF，需要连续两个\u

"\uD842\uDFB7" "𠮷" 55362 57271

Unicode应该可以避免这种的编码，前4位是其他字符的编码吧，比如\uD842是？

ES6之前: \u20BB7 "₻7"

ES6: \u{20BB7} "𠮷" 134071

D83D 55327 DE80 56960 1F680 128640

'\u{1F680}' === '\uD83D\uDE80' //true

## 遍历字符串

let text = String.fromCodePoint(0x20BB7); // "𠮷"

for (let i = 0; i < text.length; i++) {

console.log(text[i]);

}

text.length=2 打印 � �

for (let i of text) {

console.log(i);

}

打印 "𠮷"

for of可以识别大于）0xFFFF的码点，传统for循环不能。当然，传统的字就没有超过0xFFFF的

'你好世界'.length 4

'𠮷𠮷𠮷𠮷'.length 8

## 模板字符串

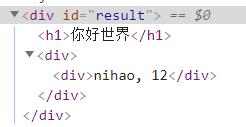
var aaa=12;

let m=document.createElement('div');

m.innerHTML=`<div>nihao, ${aaa}</div>`

document.querySelector('#result').appendChild(m);

最终的结构是



## 默认参数

function aaa(x, y=6){

}

aaa(3);

aaa()

aaa({})

aaa({x:'fefe'})

aaa({x:'fefe', y:123})

参数和解构赋值一同使用

function aaa({x, y=3}){

console.log(x, y);

}

error

undefined 3

fefe 3

fefe 123

function aaa({x, y=3}={}){

console.log(x, y);

}

undefined 3

undefined 3

fefe 3

fefe 123

function aaa({x, y}={x: 1, y:3}){

console.log(x, y)

}

1 3

undefined undefined

fefe undefined

fefe 123

function aaa({x, y=3}={x: 3, y:6}){

console.log(x, y);

}

3 6

undefined 3

fefe 3

fefe 123

以上有4种写法

* 只要在参数中用到了对象的解构赋值，想要正确的传递一个参数，就必须记住这个形参的名字，然后传一个对象进去。对象的格式为 { 形参名 : 实参 }
* 不管哪种写法，只要包含了所有的参数，就按传进去的为准，不管默认的是啥。比如只要有{x: 12, y:'13'}，x, y就必定是指定的这两个
* 函数接收参数的时候，首先判断是否有默认参数，然后如果传进了参数，再解构赋值
* 一旦发现没有传进参数，就不管结构赋值是什么样的，统一使用默认参数
* 同理，一旦传入了参数，就不管默认参数是什么样的，开始解构赋值。
* 开始解构赋值后，还有可能属性仍然有默认参数。如果传进来的这个参数没有这个属性，而且这个属性有默认参数，则还可以使用默认参数。
* 以写法4为例，如果aaa()，也就是不传参数，则使用默认参数{x: 3, y:6}；如果传了参数，但属性不全，比如传了个{x:'fefe'}，则y=3；如果传个{}，则x=undefined, y=3
* 只要函数参数使用了默认值，解构赋值，扩展运算符，函数内部就不能'use strict'
* 因为函数运行时先执行参数，后函数体，但是执行参数时不知道函数体有没有使用严格模式，所以矛盾。
* 解决方法：

1. 全局'use strict'

2. 用一个立即执行函数包起来

aaa=(function(){

'use strict'

return function(...ar){

//...

return ar;

}

})()

## rest 参数

let m=(var1, ...var2)=>var2;

是一个真正的数组

而arguments不是

## 箭头函数

let m= nihao => nihao;

等同于

let m=function(nihao){

return nihao;

}

箭头函数参数的解构赋值

let m=({var1, var2}) => var1+var2

* 箭头函数体内的this对象，就是定义时所在的对象，而不是使用时所在的对象。
* 不能当做构造函数
* 没有arguments对象

## 数组

扩展运算符spread ...

相当于rest参数逆运算，把数组转换成逗号分割的参数序列

var aaa=(var1, var2) => console.log(var1, var2);

aaa(...[12, 'qq']); //用作参数时，放在一个数组前面

[...[1, '2'], 3] //[1, 2, 3]

[...document.querySelectorAll('div')] //也可以转换NodeList等任何实现了Iterator的对象

也可以跟表达式

...(x > 0 ? ['a'] : [])

将字符串转化成数组

[...'hello world'] //["h", "e", "l", "l", "o", " ", "w", "o", "r", "l", "d"]

三种写法效果相同。apply()可以废弃了

Math.max.apply(null, [14, 3, 77])

Math.max(...[14, 3, 77])

Math.max(14, 3, 77);

* 克隆数组

ES5方法

const a1 = [1, 2];

const a2 = a1.concat();

ES6方法

const a1 = [1, 2];

const a2 = [...a1];

或

const [...a2] = a1;

都是浅拷贝

注意，即使是浅拷贝也是拷贝。和a2=a1是有区别的。

## 模块

export

**export** var m1=1;

var m2='nihao';

var m3=(a)=>console.log(a);

export { m2 as NihaoString, m3}

两种export方法，推荐第二种

as 关键词指定别名

<script type="**module**" src="./foo.js"></script>

默认defer属性开启

代码是在模块作用域之中运行，而不是在全局作用域运行。模块内部的顶层变量外部不可见。

模块脚本自动采用严格模式