目录

[1. 表达式还是语句？ 1](#_Toc23338889)

[2. 属性是否存在：in 运算符 2](#_Toc23338890)

[3. With语句 2](#_Toc23338891)

[4. 函数名的提升 3](#_Toc23338892)

[5. 立即调用的函数表达式 4](#_Toc23338893)

[6. 数字开头的对象属性读取 5](#_Toc23338894)

[7. 对象的相加 7](#_Toc23338895)

[8. map和forEach的区别 7](#_Toc23338896)

[9. 对象拷贝 8](#_Toc23338897)

[10. 异步任务顺序执行和如何正确合理使用资源（并行和串行结合） 8](#_Toc23338898)

[1. 并行与串行的结合， 8](#_Toc23338899)

[2. 防抖动debounce函数 9](#_Toc23338900)

[3. 节流 10](#_Toc23338901)

[11. null、undefined、NAN（not a number） 10](#_Toc23338902)

[12. 如何在页面上同时使用 jQuery 和其他框架？$冲突问题？ 11](#_Toc23338903)

# 表达式还是语句？

对象采用大括号表示，这导致了一个问题：如果行首是一个大括号，它到底是表达式还是语句？

{ foo: 123 }

JavaScript 引擎读到上面这行代码，会发现可能有两种含义。第一种可能是，这是一个表达式，表示一个包含foo属性的对象；第二种可能是，这是一个语句，表示一个代码区块，里面有一个标签foo，指向表达式123。

为了避免这种歧义，JavaScript 引擎的做法是，如果遇到这种情况，无法确定是对象还是代码块，一律解释为代码块。

{ console.log(123) } // 123

上面的语句是一个代码块，而且只有解释为代码块，才能执行。

如果要解释为对象，最好在大括号前加上圆括号。因为圆括号的里面，只能是表达式，所以确保大括号只能解释为对象。

({ foo: 123 }) // 正确

({ console.log(123) }) // 报错

这种差异在eval语句（作用是对字符串求值）中反映得最明显。

eval('{foo: 123}') // 123

eval('({foo: 123})') // {foo: 123}

上面代码中，如果没有圆括号，eval将其理解为一个代码块；加上圆括号以后，就理解成一个对象。

# 属性是否存在：in 运算符

in运算符用于检查对象是否包含某个属性（注意，检查的是键名，不是键值），如果包含就返回true，否则返回false。它的左边是一个字符串，表示属性名，右边是一个对象。

var obj = { p: 1 };

'p' in obj // true

'toString' in obj // true

in运算符的一个问题是，它不能识别哪些属性是对象自身的，哪些属性是继承的。就像上面代码中，对象obj本身并没有toString属性，但是in运算符会返回true，因为这个属性是继承的。

这时，可以使用对象的hasOwnProperty方法判断一下，是否为对象自身的属性。

var obj = {};

if ('toString' in obj) {

console.log(obj.hasOwnProperty('toString')) // false

}

# With语句

with (对象) {

语句;

}

它的作用是操作同一个对象的多个属性时，提供一些书写的方便。

// 例一

var obj = {

p1: 1,

p2: 2,

};

with (obj) {

p1 = 4;

p2 = 5;

}

// 等同于

obj.p1 = 4;

obj.p2 = 5;

// 例二

with (document.links[0]){

console.log(href);

console.log(title);

console.log(style);

}

// 等同于

console.log(document.links[0].href);

console.log(document.links[0].title);

console.log(document.links[0].style);

注意，如果with区块内部有变量的赋值操作，必须是当前对象已经存在的属性，否则会创造一个当前作用域的全局变量。

var obj = {};

with (obj) {

p1 = 4;

p2 = 5;

}

obj.p1 // undefined

p1 // 4

建议不要使用with语句

# 函数名的提升

如果同时采用function命令和赋值语句声明同一个函数，最后总是采用赋值语句的定义。

var f = function () {

console.log('1');

}

function f() {

console.log('2');

}

f() // 1

function y() {

a = 2;

}

y();

console.log(a);

函数y()在刚被调用的时候相当于自定义了一个全局变量，注意一定是先调用y()才会有a:

function y() {

a = 2;

}

// y();

console.log(a);

//ReferenceError: a is not defined

去掉使用y()则不会定义a；

//函数传递方式，数组

var obj = [1, 2, 3];

function f(o) {

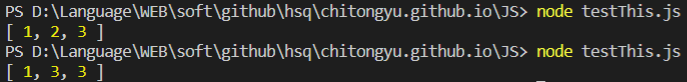
// o = [2, 3, 4];//2、整个替换不会影响

o[1] = 3;//1、修改数组的部分，会影响原来的值

}

f(obj);

console.log(obj);



# 立即调用的函数表达式

// 语句

function f() {}

// 表达式

var f = function f() {}

为了避免解析上的歧义，JavaScript 引擎规定，如果function关键字出现在行首，一律解释成语句。因此，JavaScript 引擎看到行首是function关键字之后，认为这一段都是函数的定义，不应该以圆括号结尾，所以就报错了。

解决方法就是不要让function出现在行首，让引擎将其理解成一个表达式。最简单的处理，就是将其放在一个圆括号里面。

(function(){ /\* code \*/ }());

// 或者

(function(){ /\* code \*/ })();

上面两种写法都是以圆括号开头，引擎就会认为后面跟的是一个表示式，而不是函数定义语句，所以就避免了错误。这就叫做“立即调用的函数表达式”（Immediately-Invoked Function Expression），简称 IIFE。

# 数字开头的对象属性读取

//数字开头的对象属性只能使用方括号访问

var obj = {};

obj['1p'] = 1;

obj.1p = 1;//报错，这个和数组一样，array[0]

console.log(obj);

清空数组的一个有效方法，就是将length属性设为0。

var arr = [ 'a', 'b', 'c' ];

arr.length = 0;

arr // []

上面代码将数组的键分别设为字符串和小数，结果都不影响length属性。因为，length属性的值就是等于最大的数字键加1，而这个数组没有整数键，所以length属性保持为0。

如果数组的键名是添加超出范围的数值，该键名会自动转为字符串。

var arr = [];

arr[-1] = 'a';

arr[Math.pow(2, 32)] = 'b';

arr.length // 0

arr[-1] // "a"

arr[4294967296] // "b"

上面代码中，我们为数组arr添加了两个不合法的数字键，结果length属性没有发生变化。这些数字键都变成了字符串键名。最后两行之所以会取到值，是因为取键值时，数字键名会默认转为字符串。

var arr = [];

arr[-1] = 'a';

arr[Math.pow(2, 32)] = 'b';

// arr.length // 0

// arr[-1] // "a"

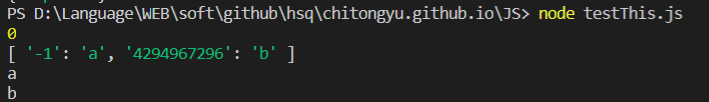
// arr[4294967296] // "b"

console.log(arr.length);

console.log(arr);

console.log(arr[-1]);

console.log(arr[4294967296]);



之所以能去到-1和字符串的值，就是因为arr其实就是一个对象，length只计算了 “键名”为数字的部分，但是实际上arr还是保存了所有付值的属性。

典型的“类似数组的对象”是函数的arguments对象，以及大多数 DOM 元素集，还有字符串。

// arguments对象

function args() { return arguments }

var arrayLike = args('a', 'b');

arrayLike[0] // 'a'

arrayLike.length // 2

arrayLike instanceof Array // false

// DOM元素集

var elts = document.getElementsByTagName('h3');

elts.length // 3

elts instanceof Array // false

// 字符串

'abc'[1] // 'b'

'abc'.length // 3

'abc' instanceof Array // false

上面代码包含三个例子，它们都不是数组（instanceof运算符返回false），但是看上去都非常像数组。

数组的slice方法可以将“类似数组的对象”变成真正的数组。

var arr = Array.prototype.slice.call(arrayLike);

var arr = [];

arr[-1] = 'a';

arr[Math.pow(2, 32)] = 'b';

arr[0] = 1;

arr[1] = 2;

var newArray = Array.prototype.slice.apply(arr);

console.log(newArray);



只会转换是数字的部分；



arr[0]为空的话，依然可以转换，但是缺少的部分使用<1 empty item>替代。

# 对象的相加

var obj = { p: 1 };

obj // "[object Object]"

上面代码中，对象obj转成原始类型的值是[object Object]。

对象转成原始类型的值，规则如下。

首先，自动调用对象的valueOf方法。

var obj = { p: 1 };

obj.valueOf().toString() // "[object Object]"

首先，自动调用对象的valueOf方法。一般来说，对象的valueOf方法总是返回对象自身，这时再自动调用对象的toString方法，将其转为字符串。

# map和forEach的区别

arr = [1,3,4].map(item => {

return item + 1

})

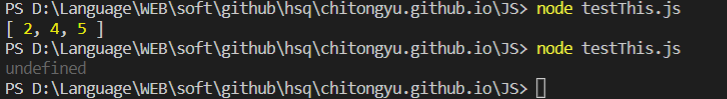
console.log(arr);

arr = [1,3,4].forEach(item => {

return item + 1

})

console.log(arr);



都能遍历数组，但是map可以返回一个数组，而forEach只能是单纯的遍历数组，不能返回数组。

# 对象拷贝

function copyObject(orig) {

return Object.create(

Object.getPrototypeOf(orig),

Object.getOwnPropertyDescriptors(orig)

);

}

# 异步任务顺序执行和如何正确合理使用资源（并行和串行结合）

所谓

## 并行与串行的结合，

就是设置一个门槛，每次最多只能并行执行n个异步任务，这样就避免了过分占用系统资源。

var items = [ 1, 2, 3, 4, 5, 6 ];

var results = [];

var running = 0;

var limit = 2;

function async(arg, callback) {

console.log('参数为 ' + arg +' , 1秒后返回结果');

setTimeout(function () { callback(arg \* 2); }, 1000);

}

function final(value) {

console.log('完成: ', value);

}

function launcher() {

while(running < limit && items.length > 0) {

var item = items.shift();

async(item, function(result) {

results.push(result);

running--;

if(items.length > 0) {

launcher();

} else if(running == 0) {

final(results);

}

});

running++;

}

}

launcher();

## 防抖动debounce函数

有时，我们不希望回调函数被频繁调用。比如，用户填入网页输入框的内容，希望通过 Ajax 方法传回服务器，jQuery 的写法如下。

$('textarea').on('keydown', ajaxAction);

这样写有一个很大的缺点，就是如果用户连续击键，就会连续触发keydown事件，造成大量的 Ajax 通信。这是不必要的，而且很可能产生性能问题。正确的做法应该是，设置一个门槛值，表示两次 Ajax 通信的最小间隔时间。如果在间隔时间内，发生新的keydown事件，则不触发 Ajax 通信，并且重新开始计时。如果过了指定时间，没有发生新的keydown事件，再将数据发送出去。

这种做法叫做 debounce（防抖动）。假定两次 Ajax 通信的间隔不得小于2500毫秒，上面的代码可以改写成下面这样。

$('textarea').on('keydown', debounce(ajaxAction, 2500));

function debounce(fn, delay){

var timer = null; // 声明计时器

return function() {

var context = this;

var args = arguments;

clearTimeout(timer);

timer = setTimeout(function () {

fn.apply(context, args)

}, delay);

};

}

上面代码中，只要在2500毫秒之内，用户再次击键，就会取消上一次的定时器，然后再新建一个定时器。这样就保证了回调函数之间的调用间隔，至少是2500毫秒。

<html>

<head>

</head>

<body>

<button type="button" οnclick="fuck()" >这是一个按钮</button>

<script>

let handler = {

time:null,

send:function(){

console.log('I have a message to server')

},

do:function(){

//清除处理好的time,阻止之前的调用被执行

clearTimeout(this.time)

let self = this

this.time = setTimeout(function(){

self.send()

},3000)

}

}

function fuck(){

handler.do()

}

</script>

</body>

</html>

## 节流

节流： 高频事件触发，但在n秒内只会执行一次，所以节流会稀释函数的执行频率。

思路：每次触发事件时都判断当前是否有等待执行的延时函数。

一种实现方式如下：

function throttle(fn, ms) {

let canRun = true; // 通过闭包保存一个标记

return function () {

if (!canRun) return; // 在函数开头判断标记是否为true，不为true则return

canRun = false; // 立即设置为false

setTimeout(() => { // 将外部传入的函数的执行放在setTimeout中

fn.apply(this, arguments);

/\* 最后在setTimeout执行完毕后再把标记设置为true(关键)表示可以执行下一次循环了。

当定时器没有执行的时候标记永远是false，在开头被return掉 \*/

canRun = true;

}, ms);

};

}

// 使用例子

function sayHi(e) {

console.log(e.target.innerWidth, e.target.innerHeight);

}

window.addEventListener('resize', throttle(sayHi, 1000));

# null、undefined、NAN（not a number）

undefined 是声明了变量但未对其初始化时赋予该变量的值，null 则用于表示尚未存在的对象

最后一个特殊值是 NaN，表示非数（Not a Number）。NaN 是个奇怪的特殊值。一般说来，这种情况发生在类型（String、Boolean 等）转换失败时

# 如何在页面上同时使用 jQuery 和其他框架？$冲突问题？

$.noConflict();

jQuery(document).ready(function(){

jQuery("button").click(function(){

jQuery("p").text("jQuery 仍在运行！");

});

});

noConflict() 方法会释放会 $ 标识符的控制，这样其他脚本就可以使用它了

var jq = $.noConflict();

jq(document).ready(function(){

jq("button").click(function(){

jq("p").text("jQuery 仍在运行！");

});

});

noConflict() 可返回对 jQuery 的引用

# js获取手机屏幕宽度、高度

网页可见区域宽：document.body.clientWidth

网页可见区域高：document.body.clientHeight

网页可见区域宽：document.body.offsetWidth (包括边线的宽)

网页可见区域高：document.body.offsetHeight (包括边线的宽)

网页正文全文宽：document.body.scrollWidth

网页正文全文高：document.body.scrollHeight

网页被卷去的高：document.body.scrollTop

网页被卷去的左：document.body.scrollLeft

网页正文部分上：window.screenTop

网页正文部分左：window.screenLeft

屏幕分辨率的高：window.screen.height

屏幕分辨率的宽：window.screen.width

屏幕可用工作区高度：window.screen.availHeight

屏幕可用工作区宽度：window.screen.availWidth