03-JS⾼级-常⻅⾯试题

# this绑定

## this的绑定规则有⼏种？

默认绑定：ᇿ⽴函数调⽤，函数没有被绑定到某个对象上进⾏调⽤

隐式绑定：通过某个对象发起的函数调⽤，在调⽤对象内部有⼀个对函数的引⽤。显式绑定：明确this指向的对象，第⼀个参数相同并要求传⼊⼀个对象。

apply/call bind

new绑定：

创建⼀个全新对象

新对象被执⾏prototype链接新对象绑定到函数调⽤的this

如果函数没有返回其他对象，表达式会返回这个对象

* 1. **this的⾯试题解析**

var name = "window"; function Person(name) {

this.name = name; this.foo1 = function () {

console.log(this.name);

};

this.foo2 = () => console.log(this.name); this.foo3 = function () {

return function () { console.log(this.name);

};

};

this.foo4 = function () { return () => {

console.log(this.name);

};

};

}

var person1 = new Person("person1"); var person2 = new Person("person2");

// person1.foo1() // person1 隐式调⽤

// person1.foo1.call(person2) // person2 显示调⽤ this指向person2所在的对象

// person1.foo2(); // person1 箭头函数 向上层作⽤查找 上层作⽤域中的this为person1指向

的对象

// person1.foo2.call(person2); // person1 箭头函数 显示绑定没⽤

// person1.foo3()() // window 相当于将返回的函数赋值给⼀个变量 指向该变量 是ᇿ⽴函数调⽤

// person1.foo3.call(person2)() // window 默认调⽤

// person1.foo3().call(person2) // person2 将函数的this显示绑定到person2所在的对象

// person1.foo4()() // person1 箭头函数 向上层作⽤域中查找this foo4中的this隐式绑定为 person1

// person1.foo4.call(person2)() // person2 箭头函数 向上层作⽤域中查找this foo4中的this显示绑定为person2

// person1.foo4().call(person2) // person1 箭头函数显示绑定没⽤ 和person1.foo4()

()xiang'tong

# 作⽤域

## 什么是变量提升、函数提升？

变量提升：

简单说就是在js代码执⾏前引擎会先进⾏预编译，预编译期间会将变量声明与函数声明提升⾄其对应作⽤域的最顶端，函数内声明的变量只会提升⾄该函数作⽤域最顶层。

当函数内部定义的⼀个变量与外部相同时，那么函数体内的这个变量就会被上升到最顶端。

举个例⼦，如：

console.log(a); // undefined

var a = 3; // 会将var a 的声明提升⾄最顶端

函数提升：

函数提升只会提升函数声明式写法，函数表达式的写法不存在函数提升。函数提升的优先级⼤于变量提升的优先级，即函数提升在变量提升之上。

## 说说你对GO/AO/VO的理解？

GO

Global Object JS代码在执⾏前会现在堆内存中创建⼀个全局对象(GO)

⽤于存放⼀些定义好的变量⽅法等包含Date Array String Number setTimeout等同时有⼀个window属性指向⾃⼰

同时在语法分析转成AST的过程中也会将⼀些变量 函数 存放在GO中 只是变量的初始值为

undeﬁned

AO

函数在执⾏前会先在堆内存中创建⼀个AO(Activation Object)对象 ⾥⾯存放这arguments 对应函数的形参 以及在函数中定义的变量 初始值为undeﬁned

VO

Variable Object 在执⾏函数时 会在执⾏上下⽂栈(ECS)中进⼊⼀个函数执⾏上下⽂(FEC)其中有三个核⼼ 核⼼之⼀是VO 指向的是该函数在内存中解析时创建的AO ⽽在全局执⾏上下⽂中指向的是 GO

## 说说你对作⽤域和作⽤域链的理解？

作⽤域

在ES5中，全局是⼀个作⽤域，函数也会产⽣作⽤域。

在ES6中，代码块、let、const等都会有属于⾃⼰的作⽤域。

作⽤域链

当进⼊到⼀个执⾏上下⽂时，执⾏上下⽂会关联⼀个作⽤域链。

通常作⽤域链在解析时就被确定，作⽤域链与函数的定义位置有关，与它的调⽤位置⽆关

## 你是如何理解闭包的,闭包到底是什么?

### 什么是闭包？

⼀个普通的函数function，如果它可以访问外层作⽤域的⾃由变量，那么这个函数和周围环境就是

⼀个闭包。

从ሀ义的⻆度来说：JavaScript中⼀个函数，如果访问了外层作⽤域的变量，那么它是⼀个闭包

### 应⽤场景

防抖 、节流 、⽴即执⾏函数 、组合函数等等

# 数组

## 常⽤的数组操作⽅法有哪些？

Array.shift()

删除并返回第⼀个元素 作⽤：从数组中删除第⼀个元素（即下标为0的元素），并返回该元素。 注意：1）删除元素之后，数组的⻓度-1。

Array.pop()

删除并返回最后⼀个元素 作⽤：从数组中删除最后⼀个元素（即下标为length-1的元素），并返回该元素。 注意：1）删除元素之后，数组的⻓度-1。

Array.push(param1[,param2,...paramN])

尾部添加元素 作⽤：在数组的尾部添加⼀个元素，并返回新数组的⻓度。 注意：1）它是直接修改该数组，⽽不是重新创建⼀个数组。

Array.unshift(newElement1[,newElement2,...newElementN])

头部添加元素 作⽤：在数组的头部添加⼀个或多个元素，并返回新数组的⻓度。 注意：1）它是直接修改该数组，⽽不是重新创建⼀个数组。

Array.join([separator])

转换成字符串 作⽤：把数组的所有元素放⼊到⼀个字符串中。 注意：1）参数separator表示字符串中元素的分隔符，可以为空，默认为半⻆逗号。

Array.reverse()

反转数组 作⽤：把数组的所有元素顺序反转。 注意：1）该⽅法会直接修改数组，⽽不会创建新的数组。

## 数组如何进⾏降维（扁平化）

利⽤Array.some⽅法判断数组中是否还存在数组，es6展开运算符连接数组

let arr = [1,2,[3,4]]

while (arr.some(item => Array.isArray(item))) { arr = [].concat(...arr);

}

使⽤数组的concat⽅法

let arr = [1,2,[3,4]] let result = []

result = Array.prototype.concat.apply([], arr)

使⽤数组的concat⽅法和扩展运算符

var arr = [1,2,[3,4]] var result = []

result = [].concat(...arr)

es6中的ﬂat函数也可以实现数组的扁平化

let arr = [1,2,['a','b',['中','⽂',[1,2,3,[11,21,31]]]],3];

let result = arr.flat( Infinity )

## 数组去重，能⽤⼏种⽅法实现？

利⽤ES6 Set去重（ES6中最常⽤）

function unique (arr) {

return Array.from(new Set(arr))

}

var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];

console.log(unique(arr))

//[1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {}, {}]

利⽤for嵌套for，然后splice去重（ES5中最常⽤）

function unique(arr){

for(var i=0; i<arr.length; i++){ for(var j=i+1; j<arr.length; j++){

if(arr[i]==arr[j]){

//第⼀个等同于第⼆个，splice⽅法删除第⼆

个

arr.splice(j,1); j--;

}

}

}

return arr;

}

var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];

console.log(unique(arr))

//[1, "true", 15, false, undefined, NaN, NaN, "NaN", "a", {…}, {…}] //NaN和{}没有去重，两个null直接消失了

利⽤indexOf去重

function unique(arr) {

if (!Array.isArray(arr)) { console.log('type error!') return

}

var array = [];

for (var i = 0; i < arr.length; i++) { if (array .indexOf(arr[i]) === -1) {

array .push(arr[i])

}

}

return array;

}

var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];

console.log(unique(arr))

// [1, "true", true, 15, false, undefined, null, NaN, NaN, "NaN", 0, "a", {…},

{…}] //NaN、{}没有去重

利⽤sort去重

function unique(arr) {

if (!Array.isArray(arr)) { console.log('type error!') return;

}

arr = arr.sort()

var arrry= [arr[0]];

for (var i = 1; i < arr.length; i++) { if (arr[i] !== arr[i-1]) {

arrry.push(arr[i]);

}

}

return arrry;

}

var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];

console.log(unique(arr))

// [0, 1, 15, "NaN", NaN, NaN, {…}, {…}, "a", false, null, true, "true", undefined] // NaN、{}没有去重

利⽤includes

function unique(arr) {

if (!Array.isArray(arr)) { console.log('type error!') return

}

var array =[];

for(var i = 0; i < arr.length; i++) {

if( !array.includes( arr[i]) ) {//includes 检测数组是否有某个值

array.push(arr[i]);

}

}

return array

}

var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];

console.log(unique(arr))

//[1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {…}, {…}]

// {} 没有去重

利⽤ﬁlter

function unique(arr) {

return arr.filter(function(item, index, arr) {

//当前元素，在原始数组中的第⼀个索引==当前索引值，否则返回当前元素 return arr.indexOf(item, 0) === index;

});

}

var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];

console.log(unique(arr))

//[1, "true", true, 15, false, undefined, null, "NaN", 0, "a", {…}, {…}]

利⽤递归去重

function unique(arr){

return arr.reduce((prev,cur) => prev.includes(cur) ? prev : [...prev,cur],

[]);

}

var arr = [1,1,'true','true',true,true,15,15,false,false, undefined,undefined, null,null, NaN, NaN,'NaN', 0, 0, 'a', 'a',{},{}];

console.log(unique(arr));

// [1, "true", true, 15, false, undefined, null, NaN, "NaN", 0, "a", {…}, {…}]

## 数组中的forEach和map的区别？

forEach() 和 map() 两个⽅法都是ECMA5中Array引进的新⽅法，都是⽤来遍历数组中的每⼀项。它们之间的区别：

map速度⽐forEach快

map会返回⼀个新数组，不对原数组产⽣影响,foreach不会产⽣新数组，forEach返回undeﬁned map因为返回数组所以可以链式操作，forEach不能

map⾥可以⽤return（return的是什么，相当于把数组中的这⼀项变为什么（并不影响原来的数组，只是相当于把原数组克隆⼀份，把克隆的这⼀份的数组中的对应项改变了） 。

forEach⾥⽤return不起作⽤，forEach不能⽤break，会直接报错。

## for in和for of的区别?

在JavaScript中遍历数组通常是使⽤for...i循环，在ES5具有遍历数组功能的还有forEach、map、ﬁlter、 some、every、reduce、reduceRight等。for...in和for...of是两种增强型循环，for...in是ES5标准，在 ES6中新增了for...of的循环⽅式。

**for...in** ：遍历以任意顺序迭代⼀个对象的除Symbol以外的可枚举属性，包括继承的可枚举属性。

**for...of**：遍历在可迭代对象，包括 Array ， Map ， Set ， String ， TypedArray ，arguments 对象等等

它们的区别：

for...in可以遍历对象和数组，for...of不能遍历对象

for...in 循环不仅遍历对象的键名，还会遍历⼿动添加的其它键，甚⾄包括原型链上的键

for...in遍历的索引为字符串类型

for..of适⽤遍历数/数组对象/字符串/map/set等拥有**迭代器对象**的集合，但是不能遍历对象 for...of与forEach()不同的是，它可以正确响应break、continue和return语句

具有迭代器对象才可以使⽤for...of

# 函数

## 说出apply、call、bind函数的⽤法和区别？

### ⽤法：

apply

第⼀个参数: 绑定this

第⼆个参数: 传⼊额外的实参, 以数组的形式

call

第⼀个参数: 绑定this

参数列表: 后续的参数以多参数的形式传递, 会作为实参

bind(不希望obj对象身上有函数)

var bar = foo.bind(obj) bar() // this -> obj

### 区别：

call、apply和bind都可以改变函数的this指向

call、apply和bind第⼀个参数的是this要指向的对象

call、apply和bind都可以后续为函数传参，apply是将参数并成⼀个数组，call和bind是将参数依次列出

call、apply都是直接调⽤，bind⽣成的this指向改变函数需要⼿动调⽤。

## 什么是纯函数？如何编写纯函数？

纯函数：纯函数⼀般具有以下的特点：

确定的输⼊⼀定会有确定的输出（外部环境的任何变化不会影响函数内部的操作产⽣的结果）纯函数的执⾏不会产⽣副作⽤。（函数内部的操作也不会对函数外部产⽣任何影响）

纯函数在react和redux中应⽤⽐较多。编写纯函数：

//⼀般的数学⽅法可以写成纯函数,例如相加 function sum(...args) {

var result = args.reduce((perValue, item) => { return preValue + item

}, 0)

return result

}

## 什么是函数柯⾥化？柯⾥化有什么作⽤？

函数的柯⾥化：

将传⼊多个参数的函数转变成传⼊单个参数并且返回⼀个函数⽤于接收剩余的参数的函数。每⼀层函数都接收⼀个参数并对参数进⾏处理。

柯⾥化的作⽤：

单⼀职责：每⼀个函数只⽤处理传⼊的单个参数，每个函数的职责单⼀⽽且确定

参数复⽤：可以拿到每⼀层函数执⾏的返回值作为⼀个新的函数，复⽤已经传⼊过的参数。

## 组合函数以及组合函数的作⽤？

组合函数：

组合函数是将多个函数组合到⼀起，进⾏依次调⽤的函数使⽤模式。组合函数的作⽤：

减少重复代码的编写，提⾼代码的复⽤性，便于开发。

可以对任意个函数进⾏组合，返回新的具有多个被组合函数功能的新函数

## 说说你对严格模式的理解？

严格模式是⼀种JavaScript的限制模式，因为种种历史原因，JavaScript语⾔在⾮严格模式下是⽐较松散的。在JavaScript不断优化和加⼊新特性的过程中，为了兼容早期的JavaScript，⼀些错误和不规范的写法也被保留了下来。这些错误也不会被抛出。在开启了严格模式后，js引擎会以⼀种更严格的规范执⾏ JavaScript代码，⼀些不规范的写法和错误也会直接抛出。

开启严格模式的⽅法：

对⽂件开启：在⽂件的开头写上"use strict"

对函数开启：在函数的开头写上"use strict"

严格模式下的语法限制：

不允许意外创建全局变量（不写var、let、const这种声明变量的关键字）会对静默失败的赋值操作抛出异常

试图删除不可删除的属性

不允许函数参数有相同的名称不允许只有0开头的⼋进制语法不允许使⽤with

⽆法获取eval中定义的变量

this绑定不会默认转成对象

# 浏览器和v8引擎

## 浏览器内核是什么？有哪些常⻅的浏览器内核？

浏览器内核⼜称浏览器渲染引擎，是浏览器的最核⼼部分。负责解析⽹⻚语法并渲染⽹⻚。常⻅的浏览器内核有：

trident（三叉戟） IE浏览器、360安全浏览器、UC浏览器、搜狗⾼速浏览器、百度浏览器

gecko（壁⻁） Mozilla、Firefox

pestro -> Blink Opera

Webkit Safari、360极速浏览器、搜狗⾼速浏览器、移动端浏览器

Webkit -> Blink Chrome、Edge

## 说出浏览器输⼊⼀个URL到⻚⾯显示的过程?

URL 输⼊

检查输⼊的内容是否是⼀个合法的 URL 链接

判断输⼊的 URL 是否完整, 如果不完整，浏览器可能会对域进⾏猜测，补全前缀或者后缀使⽤⽤户设置的默认搜索引擎来进⾏搜索

DNS 解析

浏览器不能直接通过域名找到对应的服务器 IP 地址

所以需要进⾏ DNS 解析，查找到对应的 IP 地址进⾏访问。建⽴ TCP 连接

发送 HTTP / HTTPS 请求（建⽴ TLS 连接）向服务器 发起 TCP 连接请求

当这个请求到达服务端后，通过 TCP 三次握⼿，建⽴ TCP 的连接。

* + 1. 客户端发送 SYN 包到服务器，并进⼊ SYN\_SEND 状态，等待服务器确认
    2. 服务器收到 SYN 包，必须确认客户的 SYN，同时⾃⼰也发送⼀个 SYN 包，此时服务器进⼊ SYN\_RECV 状态。

客户端收到服务器的 SYN包，向服务器发送确认包，此包发送完毕，客户端和服务器进

⼊ ESTABLISHED 状态，完成三次握⼿。服务器响应请求

当浏览器到 web 服务器的连接建⽴后，浏览器会发送⼀个初始的 HTTP GET 请求，请求⽬标

通常是⼀个 HTML ⽂件。服务器收到请求后，将发回⼀个 HTTP 响应报⽂，内容包括相关响应头和 HTML 正⽂。

浏览器解析渲染⻚⾯

处理 HTML 标记并构建 DOM 树。 处理 CSS 标记并构建 CSSOM 树。 将 DOM 与 CSSOM 合并成⼀个渲染树

根据渲染树来布局，以计算每个节点的⼏何信息。将各个节点绘制到屏幕上。

HTTP 请求结束，断开 TCP 连接.

## 说说你对 JS 引擎的理解

JavaSript 是⼀⻔解释型语⾔

JS 引擎是 JavaScript 语⾔的运⾏解释器

浏览器内核中有两种引擎，其中⼀种就是 JS 引擎

### 排版引擎

负责 HTML 和 CSS 解析和排版

### JS 引擎

负责解析和运⾏ JavaScript 语句

常⻅ JS 引擎有

SpiderMonkey -> 第⼀款 JavaScript 引擎，Brendan Eich 开发

Chakra -> 微软开发

WebKit -> JavaScriptCore -> APPLE 开发

Chrome -> V8 -> GOOGLE 开发

⼩程序 -> JSCore -> 腾讯开发

## 说说V8引擎的内存管理

JavaScript的内存管理是⾃动的

关于原始数据类型 直接在栈内存中分配关于复杂数据类型 在堆内存中分配

## 说说V8引擎的垃圾回收器

因为内存⼤⼩是有限的 所以在内存不需要的时候 需要进⾏释放 ⽤于腾出空间

GC对于内存管理有着对应的算法常⻅的算法

引⽤计数(Reference Count)

当⼀个对象有引⽤指向它时 对应的引⽤计数+1当没有对象指向它时 则为0 此时进⾏回收

但是有⼀个严重的问题 - 会产⽣循环引⽤标记清除(Mark-Sweep)

核⼼思路: 可达性

有⼀个根对象 从该对象出发 开始引⽤到所⽤到的对象 对于根对象没有引⽤到的对象 认

为是不可⽤的对象

对于不可⽤的对象 则进⾏回收

该算法有效的解决了循环引⽤的问题

⽬前V8引擎采⽤的就是该算法

V8引擎为了优化 在采⽤标记清除的过程中也引⽤了其他的算法

标记整理

和标记清除相似 不同的是回收时 会将保留下来的存储对象整合到连续的内存空间 避免内存碎⽚化

分代收集(Generational Collection)

将内存中的对象分为两组 新的空间 旧的空间

对于⻓期存活的对象 会将该对象从新空间移到旧空间中 同时GC检查次数减少

将新空间分为from和to 对象的GC查找之后从from移动到to空间中 然后to变为from from变为to 循环⼏次 对于依然存在的对象 移动到旧空间中

增量收集(Increment Collection)

如果存在许多对象 则GC试图⼀次性遍历所有的对象 可能会对性能造成⼀定的影响

所以引擎试图将垃圾收集⼯作分成⼏部分 然后这⼏部分逐⼀处理 这样会造成微⼩的延迟

⽽不是很⼤的延迟

闲时收集(IdIe-time Collection)

GC只会在CPU空闲的时候运⾏ 减少可能对代码执⾏造成的影响

## v8引擎执⾏代码的⼤致流程

Parse模块：将JavaScript代码转成AST Tree

Ignition :解释器 将ASTTree 转换为字节码(byte Code)

同时收集TurboFan 优化需要的信息

TurboFan :编译器 将字节码编译为CPU可以直接执⾏的机器码(machine code)

如果某⼀个函数呗被多次调⽤ 则会被标记为热点函数 会经过TurBoFan转换的优化的机器码让CPU执⾏ 提⾼代码性能

如果后续执⾏代码过程中 改函数调⽤时的参数类型发⽣了改变 则会逆向的转成字节码 让CPU执⾏

v8引擎执⾏流程:

⾸先会编译JavaScript 编译过程分为三步

1. 词法分析(scanner)

会将对应的每⼀⾏的代码的字节流分解成有意义的代码块 代码块被称为词法单元(token 进⾏记号化)

1. 语法分析(parser)

将对应的tokens分析成⼀个元素逐级嵌套的树 这个树称之为 抽象语法树(Abstract Syntax Tree AST)

这⾥也有对应的 pre-parser

1. 将AST 通过Ignition解释器转换成对应的字节码(ByteCode) 交给CPU执⾏ 同时收集信息

将可优化的信息 通过TurBoFan编译器 编译成更好使⽤的机器码交给CPU执⾏

如果后续代码的参数类型发⽣改变 则会逆优化(Deoptimization)为字节码

## 说说线程和进程的区别以及关系

### 进程

是 cpu 分配资源的最⼩单位；（是能拥有资源和ᇿ⽴运⾏的最⼩单位）计算机已经运⾏的程序，是操作系统管理程序的⼀种⽅式 (**官⽅说法**)

可以认为启动⼀个应⽤程序，就会默认启动⼀个进程（也可能是多个进程）**(个⼈解释)**也可以说进程是线程的容器

### 线程

是 cpu 调度的最⼩单位；（线程是建⽴在进程的基础上的⼀次程序运⾏单位，⼀个进程中可以有多个线程）

操作系统能够运⾏运算调度的最⼩单位，通常情况下它被包含在进程中 **(官⽅说法)**

每⼀个进程中，都会启动⾄少⼀个线程⽤来执⾏程序中的代码，这个线程被称之为主线程

### 操作系统的⼯作⽅式

如何做到同时让多个进程同时⼯作?

因为 CPU 的运算速度⾮常快, 可以快速的在多个进程之间迅速的切换当进程中的线程获取到世间⽚时, 就可以快速执⾏我们编写的代码

由于 CPU 执⾏速度过于变态, 对于⽤户来说是感受不到这种快速切换的

## JavaScript 为什么是单线程？

这主要和js的⽤途有关，js是作为浏览器的脚本语⾔，主要是实现⽤户与浏览器的交互，以及操作

dom

这决定了它只能是单线程，否则会带来很复杂的同步问题。

⽐如js被设计了多线程，如果有⼀个线程要修改⼀个dom元素，另⼀个线程要删除这个dom元素，此时浏览器就会⼀脸茫然，不知所措。

所以，为了避免复杂性，从⼀诞⽣，JavaScript就是单线程，这已经成了这⻔语⾔的核⼼特征，将来也不会改变

## 浏览器是多进程的?

在浏览器中，每打开⼀个tab⻚⾯，其实就是新开了⼀个进程，在这个进程中，还有ui渲染线程，js引擎线程，http请求线程等。

因此浏览器是⼀个多进程的。为了利⽤多核CPU的计算能⼒，HTML5提出Web Worker标准，允许 JavaScript脚本创建多个线程，但是⼦线程完全受主线程控制，且不得操作DOM。所以，这个新标准并没有改变JavaScript单线程的本质。

## 什么是重排重绘，如何减少重排重绘

### 重排(Reﬂow)：

元素的位置发⽣变动时发⽣重排，也叫回流。此时在关键渲染路径中的 Layout 阶段，计算每⼀个元素在设备视⼝内的确切位置和⼤⼩。当⼀个元素位置发⽣变化时，其⽗元素及其后边的元素位置都可能发⽣变化，代价极⾼。

### 重绘(Repaint):

元素的样式发⽣变动，但是位置没有改变。此时在关键渲染路径中的 Paint 阶段，将渲染树中的每个节点转换成屏幕上的实际像素，这⼀步通常称为绘制或栅格化

### 另外，重排必定会造成重绘。以下是避免过多重排重绘的⽅法：

1. 使⽤

DocumentFragment

1. CSS 样式尽量批量修改
2. 避免使⽤ table 布局

进⾏ DOM 操作，不过现在原⽣操作很少也基本上⽤不到

1. 为元素提前设置好⾼宽，不因多次渲染改变位置

# ⾯向对象

## 什么是原型、原型链？

### 原型：

在JavaScript中，每⼀个对象都会有⼀个属性[[prototype]]，这个属性就是对象的原型，这个属性的值也是⼀个对象，是原对象的原型对象。访问对象中属性时，会先在对象⾃身进⾏查找，如果没有找到，那么会去对象的原型对象上查找。

### 原型链：

每个对象都有⾃⼰的原型对象，原型对象也有⾃⼰的原型对象。在访问对象的属性时，会沿着对象⾃身

=>⾃身的原型对象=>原型对象的原型对象 这样的链条⼀路查找上去，这条链式结构就叫做原型链。

原型链的尽头是Object的原型对象的[[prototype]]属性，值为null。

## 如何通过原型链实现继承？

原型链继承：重写⼦类的显式原型对象，让⼦类的显式原型对象的隐式原型指向⽗类的显式原型对象。

function createObject(o) { function F() {}

F.prototype = o return new F()

}

function inherit(Subtype, Supertype) { Subtype.prototype = createObject(Supertype.prototype)

Object.defineProperty(Subtype.prototype, "constructor", { enumerable: false,

configurable: true, writable: true, value: Subtype

})

}

function Person() {} function Student() {

Person.call(this)

}

inherit(Student, Person)

## 继承的各个⽅案以及优缺点？

⽅案⼀：直接将⽗类的prototype赋值给⼦类的prototype，⽗类和⼦类共享原型对象缺点：在⼦类原型对象上添加⽅法和属性会影响到⽗类

function Person() {} function Student() {}

Student.prototype = Pesrson.prototype

⽅案⼆：通过new操作符创建⼀个新的对象，将这个对象作为⼦类的原型对象(显式原型)缺点：

⼦类的实例对象继承过来的属性是在原型上的，⽆法打印

没有完美的实现属性的继承（⼦类的实对象可以从⽗类继承属性，也可以拥有⾃⼰的属性）

function Person() {} function Student() {} var p = new Person() Student.prototype = p

⽅案三：通过new操作符创建⼀个新的对象，将这个对象作为⼦类的原型对象(显式原型)，并且在⼦类的内部通过借⽤构造函数的⽅法实现属性的继承

缺点：⽗类构造函数会被调⽤两次，并且⼦类的实例对象总是有两份相同的属性，⼀份在⾃身，⼀份在其原型对象上

function Person(arg1, arg2) {} function Student() {

Person.call(this, arg1, arg2)

}

var p = new Person() Student.prototype = p

⽅案四：让⼦类的原型对象(显式原型)的原型对象(隐式原型)指向⽗类的原型对象(显式原型)缺点：存在兼容性问题, proto 属性只有部分游览器⽀持

function Person() {} function Student() {} Student.prototype. proto

= Person.prototype

⽅案五：寄⽣组合式继承(ES5中实现继承的最终⽅案)

function createObject(o) { function F() {}

F.prototype = o return new F()

}

function inherit(Subtype, Supertype) { Subtype.prototype = createObject(Supertype.prototype)

Object.defineProperty(Subtype.prototype, "constructor", { enumerable: false,

configurable: true, writable: true, value: Subtype

})

}

function Person() {} function Student() {

Person.call(this)

}

inherit(Student, Person)

## 说说你对⾯向对象多态的理解

当对不同的数据类型执⾏同⼀个操作时, 如果表现出来的⾏为(形态)不⼀样, 那么就是多态的体现继承也是多态的前提

# ES6-13

## 说说let、const和var的区别？

作⽤域提升

var声明的变量是会进⾏作⽤域提升

let、const没有进⾏作⽤域提升，但是会在解析阶段被创建出来 let,const具有暂时性死区

块级作⽤域

var不存在块级作⽤域

let和const存在块级作⽤域重复声明

var允许重复声明变量

let和const在同⼀作⽤域不允许重复声明变量修改声明的变量

let,var 可以修改声明的变量

const它表示保存的数据⼀旦被赋值，就不能被修改,但是如果赋值的是引⽤类型，那么可以通过引⽤找到对应的对象，修改对象的内容

## 说说ES6~ES13新增了哪些知识点？

ES6 :

使⽤class⽤来定义类

constructor构造器 extends实现继承

super关键字代表继承的⽗类对象字⾯量的增强

属性的简写

⽅法的简写计算属性名

ES7 :

解构

let/const的使⽤

不能重复声明变量不存在作⽤域提升存在暂时性死区 不添加window

存在块级作⽤域字符串模板

在模板字符串中，我们可以通过 ${expression} 来嵌⼊动态的内容标签模板字符串

函数的默认参数

函数的剩余参数箭头函数

没有显式原型prototype

不绑定this、arguments、super参数展开语法

在函数调⽤时使⽤；在数组构造时使⽤；

展开运算符其实是⼀种浅拷⻉

在构建对象字⾯量时，也可以使⽤展开运算符，这个是在ES2018（ES9）中添加的新特性；规范了⼆进制和⼋进制的写法

新增Symbol

Set、WeakSet、Map、WeakMap

ES8 :

ES9 :

Array Includes

通过includes来判断⼀个数组中是否包含⼀个指定的元素，根据情况，包含返回 true，否则返回false。

指数exponentiation运算符

\*\*对数字来计算乘⽅。

Object values

通过Object.values 来获取所有的value值

Object entries

通过 Object.entries 可以获取到⼀个数组，数组中会存放可枚举属性的键值对数组

String Padding

padStart 和 padEnd ⽅法，分别对字符串的⾸尾进⾏填充的。

Trailing Commas

允许在函数定义和调⽤时多加⼀个逗号：

Object.getOwnPropertyDescriptors

构建对象字⾯量时，可以使⽤展开运算符

ES10 :

ﬂat

ﬂat() ⽅法会按照⼀个可指定的深度递归遍历数组，并将所有元素与遍历到的⼦数组中的元素合并为⼀个新数组返回。

ﬂatMap

ﬂatMap是先进⾏map操作，再做ﬂat的操作 ﬂatMap中的ﬂat相当于深度为1

Object fromEntries

Object.formEntries将entries转换成⼀个对象 trimStart trimEnd

去除字符串前⾯或者后⾯的空格

ES11 :

BigInt

BigInt，⽤于表示⼤的整数(超过最⼤安全整数)

BitInt的表示⽅法是在数值的后⾯加上n

空值合并操作符

??当前⾯的值为null或者undeﬁned是,显式??后⾯的值 Optional Chaining

可选链?.

当?.前⾯的值为空时返回undeﬁned

Global This

JavaScript环境的全局对象 for..in标准化

for...in遍历对象时遍历的是key

## 说说Set、WeakSet、Map、WeakMap的特点？

### Set:

⽤来存储数据,类似于数组,

与数组的区别是元素不能重复,

可以使⽤forEach⽅法和使⽤for...of...遍历常⻅属性和⽅法

size：返回Set中元素的个数 add(value)：添加某个元素，返回Set对象本身

delete(value)：从set中删除和这个值相等的元素，返回boolean类型 has(value)：判断set中是否存在某个元素，返回boolean类型 clear()：清空set中所有的元素

forEach(callback, [, thisArg])：通过forEach遍历set

### WeakSet:

只能存储对象类型,不能存放基本数据类型,

对对象的引⽤是⼀个弱引⽤,如果没有其他对对象的引⽤,那么相应对象会被GC进⾏清除,不能遍历

常⻅的⽅法

add(value)：添加某个元素，返回WeakSet对象本身 delete(value)：从WeakSet中删除和这个值相等的元素，返回boolean类型 has(value)：判断WeakSet中是否存在某个元素，返回boolean类型

### Map:

⽤于存储映射关系,存储的为键值对,每个键值对为⼀个数组,

与对象的区别是存储的key可以为⼀个对象

可以使⽤forEach⽅法和使⽤for...of...遍历常⻅属性和⽅法

size：返回Set中元素的个数

set(key, value)：在Map中添加key、value，并且返回整个Map对象 get(key)：根据key获取Map中的value has(key)：判断是否包括某⼀个key，返回Boolean类型 delete(key)：根据key删除⼀个键值对，返回Boolean类型 clear()：清空所有的元素

forEach(callback, [, thisArg])：通过forEach遍历Map

### WeakMap:

存储的key只能为对象,不允许是其他类型

对对象的引⽤是⼀个弱引⽤,如果没有其他对对象的引⽤,那么相应对象会被GC进⾏清除,不能进⾏遍历

常⻅的⽅法

set(key, value)：在Map中添加key、value，并且返回整个Map对象 get(key)：根据key获取Map中的value has(key)：判断是否包括某⼀个key，返回Boolean类型 delete(key)：根据key删除⼀个键值对，返回Boolean类型

# 异步

## 说说Promise的作⽤和使⽤⽅法（各个回调的作⽤）

### Promise的作⽤

是异步编程的⼀种解决⽅案，⽐传统回调函数解决⽅案更加合理和更强⼤。

**Promise** 对象⽤于表示⼀个异步操作的最终完成（或失败）及其结果值。

能够把异步操作最终的成功返回值或者失败原因和相应的处理程序关联起来，这样使得异步

⽅法可以像同步⽅法那样返回值。

**Promise**对象的状态不受外界影响，⼀旦状态改变,就不会再变 **Promise的使⽤⽅法**

const promise = new Promise((resolve,reject) => { resolve(value) //该函数执⾏时会回调onFulfilled

// reject(reason) //该函数执⾏时会回调onRejected

console.log("这个回调函数会被⽴即执⾏~")

})

// 监听promise对象的状态 ⽅式⼀

promise.then(onFulfilled).catch(onRejected)

// 监听promise对象的状态 ⽅式⼆

promise.then(onFulfilled,onRejected)

## Promise的实例⽅法和类⽅法

Promise的实例⽅法: then(onFulﬁlled,onRejected)

onFulﬁlled >成功时的回调

onRejected > 失败时的回调

返回值是⼀个新的promise对象 所以promise⽀持链式调⽤的原因

catch(onRejected)

onRejected >失败时的回调

ﬁnally(callback)

callback >不管promise最后的状态,在执⾏完then或catch指定的回调函数后,都会执

⾏的回调

Promise的类⽅法

all()

接受⼀个数组作为参数,数组元素是promise对象,返回⼀个新的promise对象可以不是数组,但必须是可迭代对象,且返回的每⼀个成员都是Promise实例

只有数组⾥所以的promise对象都是fulﬁlled状态时,返回的promsie的状态是fulﬁlled当数组中的promise对象有⼀个的rejected状态时,返回的promise的状态时rejected

race()

接受⼀个数组作为参数,数组元素是promise对象,返回⼀个新的promise对象只要数组中的实例有⼀个率先改变,返回的promise对象就跟着改变

allSettled()

接受⼀个数组作为参数,数组元素是promise对象,返回⼀个新的promise对象

只有等数组中所有的promise对象都发⽣状态改变后,返回的promsie对象状态才会改变返回的promsie对象,⼀旦状态发⽣改变,状态总是fulﬁlled

any()

接受⼀个数组作为参数,数组元素是promise对象,返回⼀个新的promise对象

只要数组实例中有⼀个变成fulﬁlled状态,返回的promise对象就会变成fulﬁlled状态只有当数组中所有的promise实例都变成rejected状态,返回的promise 对象才变成 rejected状态

resolve()

将现有对象转为promsie实例

rejected()

返回⼀个新的promsie实例,该实例的状态未为rejected

## 说说什么是异步函数？与普通函数有什么区别？

使⽤ async 关键字声明的函数是异步函数

// async function foo() {}

// const bar = async function() {}

// const baz = async () => {}

// class Person {

// async running() {}

// }

异步函数的执⾏流程

异步函数的内部代码执⾏过程和普通的函数是⼀致的，默认情况下也是会被同步执⾏

返回值和普通函数的区别

情况⼀：异步函数也可以有返回值，但是异步函数的返回值相当于被包裹

到 Promise.resolve 中

情况⼆：如果我们的异步函数的返回值是 Promise ，状态由会由 Promise 决定；

情况三：如果我们的异步函数的返回值是⼀个对象并且实现了 thenable ，那么会由对象的⽅ then 法来决定

如果在 async 函数中抛出异常

并不会报错, ⽽是作为 Promise 的 reject 来传递

# Proxy

## 说出Proxy和Object.deﬁneProperty的区别？

Proxy的设计初衷就是监听对象的改变，并且提供了13中⽅法监听对象的操作，⼤⼤⽅便和丰富了对对象的监听操作

拦截和监视外部对对象的访问可以直接监听数组的变化

Object.deﬁneProperty

该属性设计初衷是定义对象的属性,所以有些监听操作是监听不到的对于复杂的对象,层级很深的话,需要深度监听

删除属性,添加属性是不能被监听的

不能监听数组的变化

本质上是数组的length属性的数据属性描述符:

conﬁgurable: false 意味着length属性不能被修改,不能将length属性修改为存取属性描述符

所以数组⻓度的变化的不能被监听的

const num = [1,2,3] console.log(Object.getOwnPropertyDescriptors(num))

{

'0': { value: 1, writable: true, enumerable: true, configurable: true

},

'1': { value: 2, writable: true, enumerable: true, configurable: true

},

'2': { value: 3, writable: true, enumerable: true, configurable: true

},

length: { value: 3, writable: true, enumerable: false, configurable: false }

}

## 说说什么是Reﬂect和为什么需要使⽤它

### 什么是Reﬂect:

Reﬂect是⼀个对象，提供了多种⽅法⽅便我们统⼀管理对象。

**Reﬂect** 是⼀个内置的对象，它提供拦截 JavaScript 操作的⽅法。这些⽅法与proxy handlers (en-

US)的⽅法相同。

与⼤多数全局对象不同 Reflect 并⾮⼀个构造函数，所以不能通过new 运算符对其进⾏调⽤，或者将 Reflect 对象作为⼀个函数来调⽤。

Reflect 的所有属性和⽅法都是静态的（就像 Math 对象）。

### 为什么需要使⽤Reﬂect：

在对对象进⾏操作时有些⽅法会有返回值，操作对象变的更加规范。

Object作为构造函数，操作对象的⽅法放在它身上不是很合适,早期的设计不规范导致的。在使⽤Proxy监听对象时，⽤Reﬂect来操作对象避免了对原对象的直接操作。

# 迭代器和⽣成器

## 什么是迭代器？

迭代器是帮助我们对某个数据结构进⾏遍历的对象。

迭代器也是⼀个具体的对象，这个对象需要符合迭代器**协议**（iterable protocol）。迭代器协议定义了产⽣⼀系列值（⽆论是有限还是⽆限个）的标准⽅式

在 JavaScript 中这个标准就是⼀个特定的 next ⽅法

next ⽅法的要求

⼀个⽆参数或者⼀个参数的函数，返回⼀个应当拥有以下两个属性的对象：

done （ boolean ）

如果迭代器可以产⽣序列中的下⼀个值，则为 false 。（这等价于没有指定 done 这个属性。）

如果迭代器已将序列迭代完毕，则为 true 。这种情况下， value 是可选的，如果它依

然存在，即为迭代结束之后的默认返回值。

value

迭代器返回的任何 JavaScript 值。 done 为 true 时可省略

// 封装⼀个为数组创建迭代器的函数

function createArrayIterator(arr) { let index = 0

return {

next: function() {

if (index < arr.length) {

return { done: false, value: arr[index++] }

} else {

return { done: true }

}

}

}

}

## 什么是可迭代对象？

和迭代器不是⼀个概念

当⼀个对象实现了iterable protocol 协议时，它就是⼀个可迭代对象；

这个对象的要求是必须实现 @@iterator ⽅法，在代码中我们使⽤ Symbol.iterator 访问该属性

可迭代对象的好处

当⼀个对象变成⼀个可迭代对象的时候，就可以进⾏某些迭代操作

⽐如 操作时，其实就会调⽤它的 @@iterator ⽅法

for...of

实现可迭代协议的原⽣对象

String 、 Array 、 Map 、 Set 、 arguments 对象、 NodeList 集合...

可迭代对象的应⽤

JavaScript中语法： for ...of 、展开语法（ spread syntax ）、 yield\* 、解构赋值

（ Destructuring\_assignment ）

创建⼀些对象时： new Map([Iterable])、new WeakMap([iterable])、new

Set([iterable])、new WeakSet([iterable])

⼀些⽅法的调⽤： Promise.all(iterable)、Promise.race(iterable)、

Array.from(iterable)

迭代器的中断

⽐如遍历的过程中通过 break、return、throw 中断了循环操作

⽐如在解构的时候，没有解构所有的值

⾃定义类的迭代实现

class Person {

constructor(name, age, height, friends) { this.name = name

...

}

// 实例⽅法

/

running() {} [Symbol.iterator]() {

let index = 0 const iterator = {

next: () => {

if (index < this.friends.length) {

return { done: false, value: this.friends[index++] }

} else {

return { done: true }

}

}

}

return iterator

}

}

## 什么是⽣成器？⽣成器和迭代器有什么关系？

⽣成器是ES6中新增的⼀种函数控制、使⽤的⽅案，它可以让我们更加灵活的控制函数什么时候继续执

⾏、暂停执⾏等。

⽣成器函数也是⼀个函数，但是和普通的函数有⼀些区别

⾸先，⽣成器函数需要在function的后⾯加⼀个符号：\*

其次，⽣成器函数可以通过yield关键字来控制函数的执⾏流程：最后，⽣成器函数的返回值是⼀个（⽣成器）

⽣成器函数

⽣成器函数:

1.function后⾯会跟上符号: \* 2.代码的执⾏可以被yield控制

1. ⽣成器函数默认在执⾏时, 返回⼀个⽣成器对象
   * 要想执⾏函数内部的代码, 需要⽣成器对象, 调⽤它的next操作
   * 当遇到yield时, 就会中断执⾏

⽣成器事实上是⼀种特殊的迭代器

# 事件循环

## 说说你对事件队列、微任务、宏任务的理解

### 事件队列

事件队列是⼀种数据结构，可以存放要执⾏的任务。它符合**队列**“**先进先出**”的特点

### 宏/微任务

⾸先它们都是异步任务宏任务列队

⽤来保存待执⾏的宏任务（回调）

如: pajax、setTimeout、setInterval、DOM 监听、 UI Rendering 等 会被加⼊到宏列队

微任务列队

⽤来保存待执⾏的微任务（回调）

如: Promise 的 then 回调、 Mutation Observer API、queueMicrotask() 等 会被加⼊到微列队

JS 执⾏时会区别这两个队列

JS 引擎⾸先必须先执⾏所有的初始化同步任务代码

每次准备取出第⼀个宏任务执⾏前，都要将所有的微任务⼀个⼀个取出来执⾏，也就是优先级⽐宏任务⾼，且与微任务所处的代码位置⽆关

也就是宏任务执⾏之前，必须保证微任务队列是空的；

如果不为空，那么就优先执⾏微任务队列中的任务（回调）

# ⽹络HTTP

## 说说你对HTTP协议的理解

http

HyperText Transfer Protocol 超⽂本传输协议

超⽂本传输协议是⼀种⽤于分布式协作式的应⽤层协议

定义了客户端和服务器之间交换报⽂的格式和⽅式 默认为80端⼝使⽤tcp作为传输层协议 保证了数据的可靠性

组成: ⼀个HTTP请求主要包括: 请求和响应

请求

主要包含: 请求⾏ 请求头 请求体请求⾏:

请求⽅法字段

URL字段

HTTP协议版本字段

GET/index.html HTTP/1.1

请求头:

键值对组成

User-Agent: 对应展示的浏览器的类型

Content-type: 对应的请求内容的数据类型

application/x-www-form-urlencoded 数据以&分割 的键值对 键值对⽤=分割

application/json json类型 application/xml xml类型 text/plain ⽂本类型 multipart/form-data 表示上传⽂件

keep-alive

请求体: get/post所带的内容

响应

响应⾏

由协议版本 状态码 状态码的原因短语组成

HTTP/1.1 200 OK

请求⽅法

响应头响应体

get 向服务器获取数据

post 将响应实体交给指定的资源

head 请求⼀个与get请求响应相同的响应 没有实体

put 上传⽂件 ⽤于替换⽬标资源的所有

patch ⽤于对资源的部分修改

delete 删除指定的资源

connect: 建⽴⼀个到⽬标资源标识的服务器的隧道 通常⽤于代理服务器

track: 回显服务器收到的请求 主要⽤于测试和诊断

响应状态码

1. 表示请求被服务器端正常处理
2. post请求 创建新的资源

301 永久重定向 表示资源被分配了新的URI 并返回该URI

4xx 表示客户端发⽣错误

1. 请求报⽂存在语法错误
2. 未授权的错误 必须携带身份信息

403 没有权限访问

404 服务器找不到请求资源

5xx 服务器错误

500

503 服务器不可⽤ 处于维护或重载状态

## 说说XMLHttpRequest和Fetch请求的异同

Fecth提供了⼀种更加现代的处理⽅案

⽐如返回⼀个值是 Promise

在请求成功时调⽤resolve回调

与XMLHttpRequest不同 不⽤把所有操作放在同⼀个对象上

语法简单 更加语义化

基于标准的promise实现 ⽀持async/await更加底层

Fetch缺点

不⽀持abort(超时取消请求) 不⽀持超时控制没有办法检测请求进度 XHR可以

默认不会携带cookie

ajax缺点

使⽤起来⽐较繁琐

## 如何取消请求的发送

xhr

如果使⽤ XMLHttpRequest 发送请求可以使⽤ XMLHttpRequest.abort()

fetch

如果使⽤ fetch 发送请求可以使⽤ AbortController

const controller = new AbortController(); const signal = controller.signal; fetch('https://somewhere', { signal }) controller.abort()

axios

如果使⽤ axios ，取消原理同 fetch

var CancelToken = axios.CancelToken;

var source = CancelToken.source();

axios.get('/https://somewhere', { cancelToken: source.token

}

source.cancel()

## 什么是跨域？为什么有跨域问题？

### 什么是跨域：

跨域是指跨域名的访问，以下情况都属于跨域：

|  |  |
| --- | --- |
| **跨域原因说明** | **示例** |
| 域名不同 | [www.jd.com](http://www.jd.com/) 与 [www.taobao.com](http://www.taobao.com/) |
| 域名相同，端⼝不同 | www.jd.com:8080 与 www.jd.com:8081 |
| ⼆级域名不同 | item.jd.com 与 miaosha.jd.com |

如果**域名和端⼝都相同，但是请求路径不同**，不属于跨域。

### 为什么有跨域问题：

跨域问题是浏览器对于ajax请求的⼀种安全限制：**⼀个⻚⾯发起的ajax请求，只能是于当前⻚同域名的路径**，这能有效的阻⽌跨站攻击。

### 因此：跨域问题 是针对ajax的⼀种限制。

但是这却给我们的开发带来了不变，⽽且在实际⽣成环境中，肯定会有很多台服务器之间交互，地址和端⼝都可能不同，怎么办？这时我们就要解决跨域问题了

## 解决跨域问题

⽬前线上常⽤的跨域解决⽅案有2种：

Nginx反向代理

利⽤nginx反向代理把跨域为不跨域，⽀持各种请求⽅式缺点：需要在nginx进⾏额外配置

CORS规范化的跨域请求解决⽅案，安全可靠。

优势： 在服务端进⾏控制是否允许跨域，可⾃定义规则，⽀持各种请求⽅式缺点：CORS需要IE10+

## 什么是REST，⽤起来有什么好处

REST是⼀种软件架构模式（即API接⼝的设计模式），最常⽤的数据格式是JSON。

由于JSON能直接被JavaScript读取，所以，以JSON格式编写的REST⻛格的API具有简单、易读、易⽤的特点。

通过REST模式设计的API可以把webapp全部功能进⾏封装，可以很容易的实现前后端分离，使的前端代码易编写，后端代码易测试。

# 其它

## 说说async和defer的使⽤以及区别？

浏览器在解析构建DOM树的过程中 如果遇到script元素会停⽌构建DOM树 先下载JavaScript代码 执⾏对应的脚本

但是某些JavaScript代码中可能存在对某个节点的操作 如果等待DOM树构建完成 之后在进⾏对应的操作 则会造成⼤量的回流和重绘

同时在如果JavaScript 代码过多 则浏览器处理的时间会过⻓ 则会造成⻚⾯的阻塞为了解决这个问题 出现了两个属性 async defer

defer

脚本的下载会与DOM树的构建同时进⾏

如果脚本提前下载好了 则会等到DOM树构建完成之后 在DOMContentLoaded事件之前执⾏defer中的代码

同时多个defer属性的script标签 则会按照顺序执⾏推荐放到head标签中 可以早解析

对于script默认的内容 会忽略

async

脚本的下载会与DOM树的构建同时进⾏

让⼀个脚本完全ᇿ⽴ 脚本的解析 运⾏于DOM的构建⽆关多个async属性的脚本不保证运⾏顺序

通常⽤于ᇿ⽴的脚本 对于其他脚本 DOM没有依赖

## LocalStorage和SessionStorage的区别

LocalStorage提供⼀种永久性存储的⽅法 在⽹⻚关闭打开时 依然保留

SessionStorage: 会话存储 再关闭该⽹⻚时 存储的内容被清除区别:

localStorage永久性存储 SessionStorage在关闭当前⻚⾯时存储的内容就会失效 SessionStorage只能被同⼀个窗⼝的同源⻚⾯共享 localStorage除⾮⼿动删除 否则⼀直存在

## 说说new操作背后的原理？

在内存中创建⼀个空对象 。⽐如 var moni={}

将构造函数的显示原型赋值给这个对象的隐式原型 。moni.**proto**=Person.prototype this指向创建出来的新对象。this=moni

执⾏函数体代码

如果构造函数没有返回⾮空对象，那⾃动返回创建出来的新对象 return moni

## 说说你对防抖、节流的理解，以及它们的区别和应⽤场景？

**防抖**: 将多次执⾏函数变成最后⼀次执⾏ 等待固定时间还没有事件触发时执⾏的函数应⽤场景

按钮的点击

屏幕滚动时的复杂计算输⼊框输⼊时进⾏搜索

⽤户缩放浏览器的resize事件简单的防抖函数实现

function myDebounce(execFn, delay) {

let timer = 0

function \_debounce(...args) { if (timer) clearTimeout(timer) timer = setTimeout(() => {

execFn.apply(this, args) timer = null

}, delay)

}

return \_debounce

}

**节流**: 按照固定的时间频率(间隔)来执⾏对应的函数应⽤场景:

监听⻚⾯的滚动事件 通过节流来降低事件调⽤的频率

⿏标移动

⽤户频繁点击按钮的操作简单实现

function myThrottle(execFn, interval) {

let initTime = 0

function throttle(...args) { let nowTime = Date.now()

const waitTime = interval - (nowTime - initTime) if (waitTime <= 0) {

execFn.apply(this, args) initTime = nowTime

}

}

return throttle

}

## 什么是值传递，什么引⽤传递？

值类型(原始类型):

在变量中保存的是值本身,占据的空间是在栈内存中分配的引⽤类型(对象类型):

在变量中保存的是对象的“引⽤”,占据的空间是在堆内存中分配的

值传递: 是将值类型传递给函数参数，函数内部对参数的改变不会影响函数外部的变量， 例如:

function foo(a) { a = 200

}

var num = 100 foo(num)

console.log(num) // 100

引⽤传递: 是将引⽤类型传递给函数参数,函数参数保存的是对象的"引⽤"，在函数内部修改对象的属性，函数外部对象的属性也会跟着修改，例如:

function foo(a) { a.name = "HTML"

}

var obj = {

name: "JavaScript"

}

foo(obj) console.log(obj.name) // HTML

## 说说对象的引⽤赋值、浅拷⻉、深拷⻉的区别

对象的引⽤赋值

把源对象指向⾃身所在堆内存空间的指针给了新对象，两个对象所指向的内存空间是⼀样的，修改其中⼀个的值，另⼀个也会发⽣改变

对象的浅拷⻉

可以通过{...obj}的⽅式进⾏对象的浅拷⻉ (Object.assign({},obj))

对于obj中的值是原始数据类型的 将对应的值赋值给了newObj中对应的属性

对于obj中是复杂数据类型的值，把对应在内存中的指针赋值给了newObj中对应的key 对于复杂数据类型的value修改其中⼀个另⼀个也发⽣改变

对象的深拷⻉

newObj与obj中的属性值⼀样，但是是⼀个全新的对象，与原对象没有任何关系默认情况下 js没有提供对应的深拷⻉的⽅式 因为深拷⻉是⾮常消耗内存的。 有对应的库实现了深拷⻉（lodash...）。

实现深拷⻉

JSON.parse( JSON.stringfy(obj))

缺点: 对于某些属性如 undeﬁned,Symbol,function,Symbol 会⾃动忽略; 对于set map

会转成对象

⾃⼰实现或使⽤第三⽅库

## 如何实现深拷⻉和浅拷⻉？

浅拷⻉：

const info = { name: "why", age: 18, friend: {

name: "kobe"

},

running: function() {}, [Symbol()]: "abc"

}

// 浅拷⻉-⽅式⼀

const obj2 = { ...info }

// 浅拷⻉-⽅式⼆

const obj3 = Object.assign({}, info)

深拷⻉：

function isObject(obj) {

return obj !== null && (typeof obj === "object" || typeof obj === "function")

}

function deepClone(originValue) {

// Symbol类型

if (typeof originValue === "symbol") { return Symbol(originValue.description)

}

// 判断是否是对象

if (!isObject(originValue)) return originValue;

// set类型

if (originValue instanceof Set) { const newSet = new Set()

for (const setItem of originValue) { newSet.add(deepClone(setItem))

}

return newSet

}

// 判断是函数

if (typeof originValue === "function") { return originValue

}

// 判断返回值是数组还是对象

const newObj = Array.isArray(originValue) ? [] : {} if (Reflect) {

for (let key of Reflect.ownKeys(originValue)) {

{

let value = originValue[key]

// 让 SymbolKey的值不同

if (typeof key === "symbol") {

const newSymbolKey = Symbol(key.description)

// 将原来的值赋值给新⽣成的Symbol key value = originValue[key]

key = newSymbolKey

}

newObj[key] = deepClone(value)

}

}

} else {

for (const key in originValue) { const value = originValue[key] newObj[key] = deepClone(value)

}

// 对于Symbol类型的key forin ⽆法便利出来

const symbolKeys = Object.getOwnPropertySymbols(originValue) for (const symbolKey in symbolKeys) {

const originSymbolValue = symbolKeys[symbolKey]

const newSymbol = Symbol(originSymbolValue.description) newObj[newSymbol] = deepClone(originValue[originSymbolValue])

}

}

return newObj

}

const info = { name: "why", age: 18, friend: {

name: "kobe"

},

running: function() {},

[Symbol()]: "abc"

}

const newObj = deepCopy(info)

## 说说服务端渲染和前后端分离的区别

### SSR(server side rendering) 服务端渲染

优点:

更快的响应时间 不⽤等待所有的js加载完成 也能显示⽐较完整的⻚⾯

更好的SEO 可以将SEO的关键信息直接在后台渲染成html 保证了搜索引擎能爬取到关键数据

⽆需占⽤客户端资源 解析模板交给后端⼯作 对于客户端的资源占⽤更少

缺点

占⽤服务器资源 ⼀个⼩⼩的⻚⾯的改动 都需要请求⼀次完整的html⻚⾯ 有悖于程序员的

=="DRY(Don`t repeat yourself)"==原则 如果短时间访问过多 对服务器造成⼀定的访问压⼒

⼀些常⻅的api需要先对运⾏环境判断再使⽤

### 前后端分离

优点:

缺点:

前端专注于ui界⾯的开发 后端专注于api的开发 单⼀体验更好

第⼀次响应内容较慢 不如服务端渲染快

不利于SEO优化 只是记录⼀个⻚⾯ 对于SEO较差