1.

function sayHi(){

console.log(name)

console.log(age)

var name = 'Lydia';

let age = 21;

}

sayHi();

undefinded ReferenceError

var 声明 存在变量提升

let 和 const 也会存在变量提升，但初始化没有被提升。 在声明之前，不可访问。被称为‘暂时性死区’。

变量的赋值可以分为三个阶段：

创建变量，在内存中开辟空间；初始化变量，将变量初始化为undefined；真正赋值。

关于let 、var 、function：

let的创建过程被提升，但是初始化没有提升。

var的创建和初始化都被提升了。

function 的创建、初始化和赋值都被提升。

2.

for(var i=0;i<3;i++){

setTimeout(()=>console.log(i),1)

}

for(let i=0;i<3;i++){

setTimeout(()=>console.log(i),1)

}

3 3 3 和 0 1 2

由于JavaScript中的事件执行机制，setTimeout函数真正被执行时，循环已经走完。由于第一个循环中的变量i是使用var关键字声明的，因此该值是全局的。在循环期间，我们每次使用一元运算符++都会将i的值增加1。因此在第一个例子中，当调用setTimeout函数时，i已经被赋值为3。

使用let和const关键字声明的变量具有块级作用域的。在每次迭代期间，i将会被创建为一个人新值，并且每个值都会存在于循环内的块级作用域。

3．

const shape = {

radius:10,

diameter(){

return this.radius \* 2;

},

perimeter:()=>2 \* Math.PI \* this.radius

};

20 NAN

diameter 普通函数而perimeter 箭头函数。由于箭头函数，this关键字指向是它所在上下文（定义时位置）的环境。意味着调用perimeter时它不是指向shape,而是指定义时的环境（window）。没有radius属性，返回undefined。

4.

+true;

!"Lydia";

1 and false

一元加号会将Boolean 转换为数字类型。true转换为1，false转换为0。

5.

const bird = {

size: "small"

};

const mouse = {

name : "Mickey",

small : true

};

A:mouse.bird.size

B:mouse[bird.size]

C:mouse[bird['size']]

D:All of them are valid

A

在JavaScript 中，所有对象键都是字符串（除了Symbol）。尽管有时候我们可能不会给定字符串类型，但它总是被转换为字符串。

在使用方括号表示法时，它会看到第一个左括号[,然后继续，直到找到右括号]。只有在那个时候，它才会对这个语句求值。

但是点表示法不会。点表示法从左到右进行，点之后就取值。

6.

let c={greeting : 'Hey'};

let d;

d=c;

c.greeting='Hello';

console.log(d.greeting)

Hello

在JavaScript中，当设置他们彼此相等时，所有对象都通过引用进行交互。更改一个对象，可以更改所有对象。

7.

let a = 3;

let b = new Number(3);

let c = 3;

console.log(a == b)

console.log(a === b)

console.log(b === c)

true false false

new Number()是一个内置的构造函数。虽然它看起来像一个数字，但是它并不是一个真正的数字；它有一堆额外的功能，是一个对象。

== 隐士类型转换，只比较值

=== 类型和值都需要比较

8.

class Chameleon{

static colorChange(newColor){

this.newColor = newColor;

return this.newColor;

}

constructor({newColor='green'}={}){

this.newColor = newColor;

}

}

const freddie = new Chameleon({newColor:"purple"});

freddie.colorChange('orange');

TypeError: freddie.colorChange is not a function

colorChange方法是静态方法。静态方法仅在创建它们的构造函数中存在并且不能传递给任何子级。由于freddie是一个子级对象，函数不会传递，所以在freddie实例上不存在colorChange方法。类（class）通过 static 关键字定义静态方法。不能在类的实例上调用静态方法，而应该通过类本身调用

9.

let greeting;

greetign = {};

console.log(greetign)

{}

当我们错误地将greeting输入为greetign时，JS解释器实际上在浏览器中将其视为globa.greetign={}或window.greetign={};

为了避免这种情况，可以使用’use strict’。可以确保在将变量赋值之前必须声明变量。

10.

function bark(){

console.log('Woof!');

}

bark.animal = 'dog';

Nothing, this is totally fine! 没什么，这完全没有问题

函数时一种特殊类型的对象。该函数是具有属性的对象。此属性是可调用的。

11.

function Person(firstName, lastName){

this.firstName = firstName;

this.lastName = lastName;

}

const member = new Person('Lydia','Hallie');

Person.getFullName = function(){

return `${this.firstName} ${this.lastName}`;

};

console.log(member.getFullName())

不能像使用常规对象那样向构造函数添加属性。如果要一次向所有对象添加功能，则必须使用原型。

假设将此方法添加到构造函数本身，也许不是每个Person实例都需要这种方法。这会浪费大量内存空间，这占用了每个实例的内存空间。相反，如果我们只将它添加到原型中，我们只需要将它放在内存中的一个位置，但他们都可以访问它。

12.

function Person(firstName, lastName){

this.firstName = firstName;

this.lastName = lastName;

}

const lydia = new Person('Lydia','Hallie');

const sarah = Person('Sarah','Smith');

console.log(lydia);

console.log(sarah);

Person {firstName: "Lydia", lastName: "Hallie"} and undefined

<https://www.cnblogs.com/yzhihao/p/9306511.html>

13.

事件传播的三个阶段： Capturing捕获 -> Target目标 ->Bubbling冒泡

DOM事件流（event flow ）存在三个阶段：事件捕获阶段、处于目标阶段、事件冒泡阶段。

14.所有对象都有原型。 错的

除基础对象外，所有对象都有原型。基础对象可以访问某些方法和属性。例如 .toString。这就是可以使用内置JavaScript方法的原因。基础对象指原型链终点的对象。基础对象的原型是null。

15.

function sum(a,b){

return a+b;

}

sum(1,'2')

‘12’ 隐式类型转换先会先将1转换为string类型，进行字符串的拼接运算。

javascript中string与number相加得到字符串，相减却是数字

16.

let number = 0;

console.log(number++);

console.log(++number);

console.log(number);

1. 2 2

后置型递增（递减）**先赋值再计算**

前置型递增（递减）**先计算再赋值**

17.

function getPersonInfo(one,two,three){

console.log(one);

console.log(two);

console.log(three);

}

const person = 'Lydia';

const age = 21;

getPersonInfo`${person} is ${age} years old`;

["", " is ", " years old"] "Lydia" 21

如果使用标记的模板字符串，则第一个参数的值始终是字符串的数组。其余参数获取传递到模板字符串中的表达式的值。

模板字符串和标签模板是两个东西。

标签模板不是模板，而是函数调用的一种特殊形式。“标签”指的就是函数，紧跟在后面的模板字符串就是它的参数。

但是，如果模板字符串中有变量，就不再是简单的调用了，而是要将模板字符串先处理成多个参数，再调用函数。

模板字符串由变量和非变量组成，什么是变量，${}就是变量。模板标签函数调用的第一个参数是数组，是由模板字符串中那些非变量部分组成，包括空格。

模板字符串中变量必须是由非变量包含着的，如果变量在开始位置或者结束位置且没有非变量包含，那么该位置就是空字符串。

18.

function checkAge(data){

if(data === {age:18}){

console.log('You are an audlt!')

}else if(data=={age:18}){

console.log('You are still an audlt.')

}else{

console.log(`Hmm.You don't have an age I guess`)

}

}

checkAge({age:18})

Hmm.. You don't have an age I guess

在比较相等性，基本数据类型通过它们的值进行比较，而对象通过它们的引用进行比较。JavaScript检查是否具有对内存中相同位置的引用。

19.

function getAge(...args){

console.log(typeof args)

}

getAge(21)

"object"

扩展运算符（…args）返回一个带参数的数组。数组是一个对象。

20.

function getAge(){

'use strict';

age = 21;

console.log(21)

}

getAge()

ReferenceError:age is not defined

使用’use strict’,可以确保不会意外地声明全局变量。

21.

const sum = eval("10\*10+5");

105

eval 会为字符串传递的代码求值。如果它是一个表达式，就像在这种情况下一样，它会计算表达式。

22.

cool\_secret 可以访问多长时间？

sessionStorage.setItem(‘cool\_secret’,123);

用户关闭选项卡时

23.

var num = 8;

var num = 10;

console.log(num)

10

使用var关键字，可以用相同的名称声明多个变量，然后变量将保存最新的值。

24.

const obj = {1 : 'a', 2 : 'b', 3 : 'c'};

const set = new Set([1,2,3,4,5]);

obj.hasOwnProperty('1');

obj.hasOwnProperty(1);

set.has('1');

set.has(1);

true true false true

所有对象键（不包括Symbols）都会被存储为字符串，即使你没有给定字符串类型的键。上面的说法不适用与Set。在Set中没有’1’。有数字型1。

25.

const obj = {a:'one',b:'two',a:'three'};

console.log(obj)

 { a: "three", b: "two" }

如果对象有两个具有相同名称的键，则将替前面的键。它扔将处于第一个位置，但具有最后指定的值。

26.

JavaScript全局执行上下文为你创建了两个东西：全局对象和this关键字。

27.

for(let i=1;i<5;i++){

if(i===3)continue;

console.log(i);

}

1. 2 4

如果某个条件返回true，则continue语句跳过迭代。Continue 跳过当前循环，继续执行

28.

String.prototype.giveLydiaPizza=()=>{

return 'Just give Lydia pizza already!';

};

const name='Lydia';

name.giveLydiaPizza();

"Just give Lydia pizza already!"

String是一个内置的构造函数，可以为它添加属性。刚给它的原型添加了一个方法。基本数据类型的字符串自动转换为字符串对象，由字符串原型函数生成。因此，所有字符串（字符串对象）都可以访问该方法。

29.

const a = {};

const b = {key : 'b'};

const c = {key : 'c'}

a[b]=123;

a[c]=456;

console.log(a[b])

456

对象键自动转换为字符串。我们试图将一个对象设置为对象a的键，其值为123。

但是，当对象自动转换为字符串化时，它变成了[Object object]。 所以我们在这里说的是a["Object object"] = 123。 然后，我们可以尝试再次做同样的事情。 c对象同样会发生隐式类型转换。那么，a["Object object"] = 456。

然后，我们打印a[b]，它实际上是a["Object object"]。 我们将其设置为456，因此返回456。

30.

const foo = () => console.log('First');

const bar = () => setTimeout(()=>console.log('Second'));

const baz = () => console.log('Third');

bar();

foo();

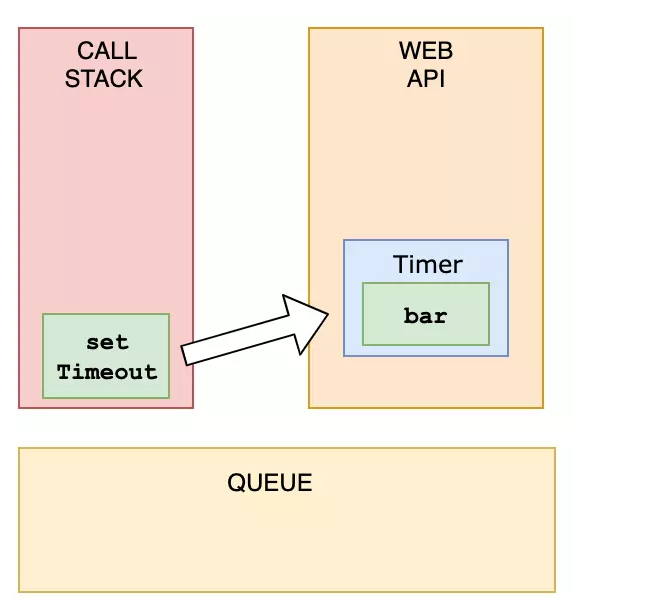
baz();

First Third Second

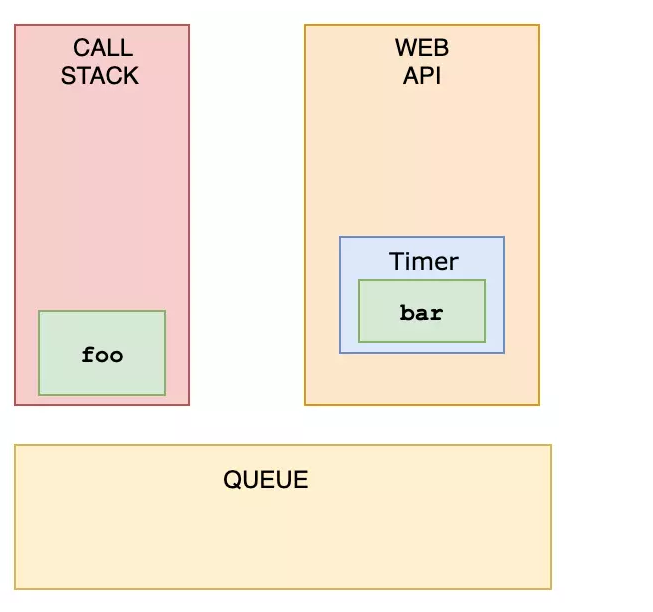
我们有一个setTimeout函数并首先调用它。 然而却最后打印了它。

这是因为在浏览器中，我们不只有运行时引擎，我们还有一个叫做WebAPI的东西。WebAPI为我们提供了setTimeout函数，例如DOM。

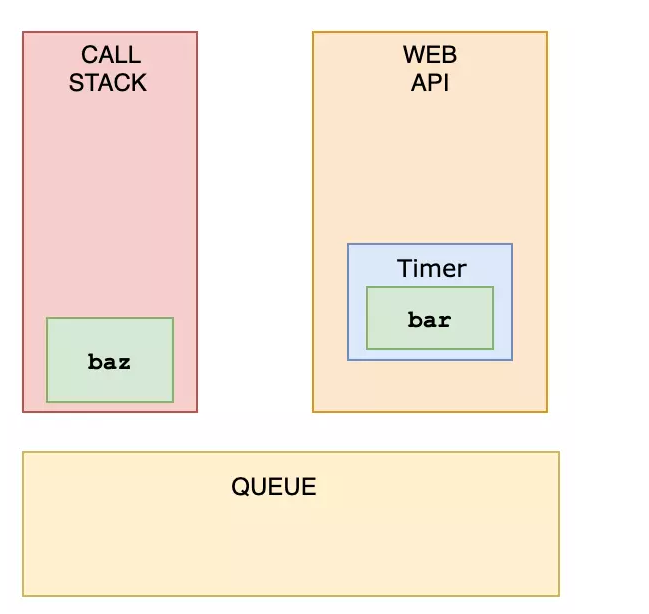
将callback推送到WebAPI后，setTimeout函数本身（但不是回调！）从堆栈中弹出。



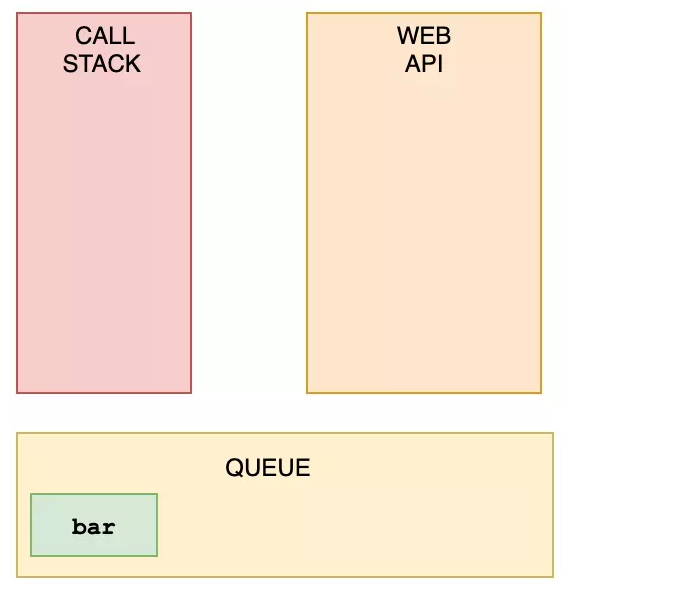
现在，调用foo，并打印First。



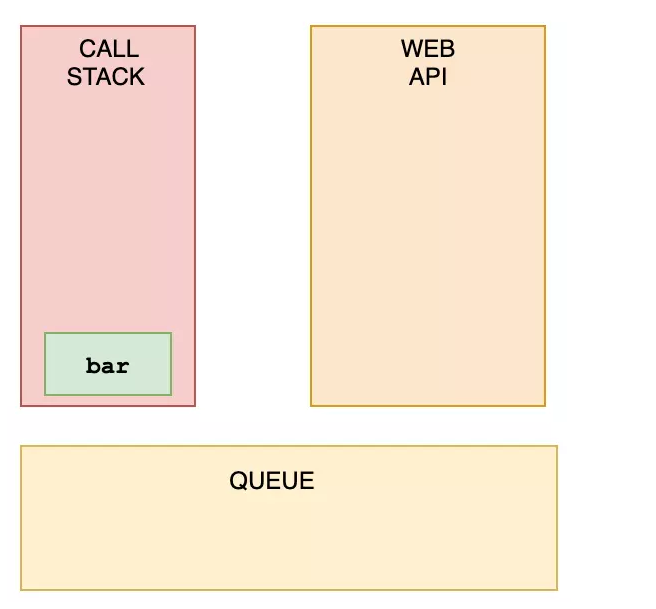
foo从堆栈弹出，baz被调用，并打印Third。



WebAPI不能只是在准备就绪时将内容添加到堆栈中。 相反，它将回调函数推送到一个称为任务队列的东西。



这是事件循环开始工作的地方。 事件循环查看堆栈和任务队列。 如果堆栈为空，则会占用队列中的第一个内容并将其推送到堆栈中。



bar被调用，Second被打印，它从栈中弹出。

31.单击按钮时event.target是什么？

<div onClick="console.log('first div')">

<div onClicl="console.log('second div')">

<button onClick="console.log('button')">

Click!

</button>

</div>

</div>

button

导致事件的最深嵌套元素是事件的目标。可以通过event.stopPropagation停止冒泡。

32.

单击下面的html片段，打印的内容是什么？

<div onClick="console.log('div')">

<p onClick="console.log('p')">

Click here!

</p>

</div>

P div

在事件传播期间，有三个阶段：捕获、目标、冒泡。默认情况下，事件处理程序在冒泡阶段执行。它从最深的嵌套元素向外延伸。

33.下面代码的输出是什么？

const person = {name : 'Lydia'};

function sayHi(age){

console.log(`${this.name} is ${age}`)

}

sayHi.call(person,21);

sayHi.bind(person,21);

Lydia is 21 function

使用两者，可以传递我们想要的this关键字引用的对象。但是.call方法会立即执行。.bind方法会返回函数的拷贝值，但带有绑定的上下文！它不会立即执行。

1.this对象的涵义就是指向当前对象中的属性和方法。

2.this指向的可变性。当在全局作用域时，this指向全局；当在某个对象中使用this时，this指向该对象；当把某个对象的方法赋值给另外一个对象时，this会指向后一个对象。

3.this的使用场合有：在全局环境中使用；在构造函数中使用，在对象的方法中使用。

4.this的使用注意点，最重要的一点就是要避免多层嵌套使用this对象。

总结一下call，apply，bind方法：

a：第一个参数都是指定函数内部中this的指向（函数执行时所在的作用域），然后根据指定的作用域，调用该函数。

b：都可以在函数调用时传递参数。call，bind方法需要直接传入，而apply方法需要以数组的形式传入。

c：call，apply方法是在调用之后立即执行函数，而bind方法没有立即执行，需要将函数再执行一遍。有点闭包的味道。

d：改变this对象的指向问题不仅有call，apply，bind方法，也可以使用that变量来固定this的指向。

34.下面代码的输出是什么？

function sayHi(){

return (()=>0)();

}

typeof sayHi();

‘number’

只有7种内置类型。null undefined Boolean number string object symbol.function不是一个类型，因为函数是对象，它的类型是object。

35.下面这些值哪些是假值？

0;

new Number(0);

("");

(" ");

new Boolean(false);

undefined;

0, '', undefined

JavaScript 只有6个假值：

0 false undefined null NaN ‘’(empty string)

函数构造函数，如new Number new Boolean 都是真值。

36.下面代码的输出是什么？

console.log(typeof typeof 1);

‘string’

Typeof 1 返回’number’ typeof ‘number’ 返回string

37.下面代码的输出是什么？

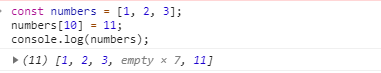
const numbers = [1, 2, 3];

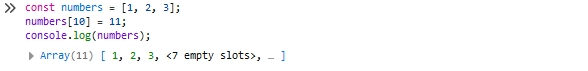
numbers[10] = 11;

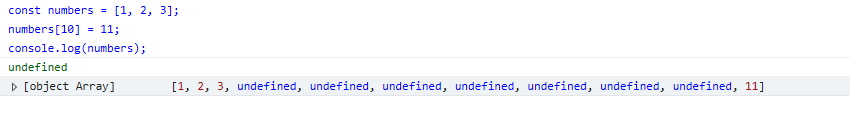
console.log(numbers);

[1, 2, 3, empty × 7, 11]

当你为数组中的元素设置一个超过数组长度的值时，JavaScript会创建一个名为‘空插槽’的东西。这些位置的值实际上是undefined，但你会看到类似的东西[1, 2, 3, empty × 7, 11]，这取决于你运行它的位置（每个浏览器有可能不同）

Chrome 

firefox

IE

38.下面代码的输出是什么?

(()=>{

let x,y;

try{

throw new Error();

}catch(x){

(x=1),(y=2);

console.log(x);

}

console.log(x);

console.log(y);

})();

1 undefined 2

Catch块接受参数x。当我们传递参数时，这与变量的x不同。这个变量x是属于catch作用域的。之后，将这个块级作用域的变量设置为1，并设置变量y的值。现在，我们打印块级作用域的变量x,它等于1。在catch块外，x是undefined,而y是2。

39.JavaScript中的所有内容都是基本数据类型和对象。

基本数据类型(简单数据类型)：包括布尔值，数字，字符串，null和undefined，symbol

引用类型：object

40.下面代码的输出是什么？

[[0,1],[2,3]].reduce(

(acc,cur)=>{

return acc.concat(cur);

},

[1,2]

);

[1, 2, 0, 1, 2, 3]

arr.reduce(function(prev,cur,index,arr){…},init)

arr表示元数组

prev 表示上一次调用回调时的返回值，或者初始值init;

cur 表示当前正在处理的数组元素；

index 表示当前正在处理的数组元素的索引，若提供init值，则索引为0，否则索引为1

init 表示初始值

41.下面代码的输出是什么？

!!null;

!!"";

!!1;

false false true

42.setInterval方法的返回值是什么？

setInterval(() => console.log("Hi"), 1000);

一个唯一的id.此id可用于使用clearInterval()函数清除定时器。

43.返回什么？

[..."Lydia"];

["L", "y", "d", "i", "a"]

字符串是可迭代的。扩展运算符将迭代的每个字符映射到一个元素上。

一道经典的面试题，如何让：a == 1 && a == 2 && a == 3。

根据上面的拆箱转换，以及==的隐式转换，我们可以轻松写出答案：

const a = {

value:[3,2,1],

valueOf: function () {return this.value.pop(); },

}

在JavaScript中，每一个变量在内存中都需要一个空间来存储。

内存空间又被分为两种，栈内存与堆内存。

栈内存：

存储的值大小固定

空间较小

可以直接操作其保存的变量，运行效率高

由系统自动分配存储空间

JavaScript中的原始类型的值被直接存储在栈中，在变量定义时，栈就为其分配好了内存空间。

由于栈中的内存空间的大小是固定的，那么注定了存储在栈中的变量就是不可变的。

堆内存：

存储的值大小不定，可动态调整

空间较大，运行效率低

无法直接操作其内部存储，使用引用地址读取

通过代码进行分配空间

**内存泄漏的识别方法**

经验法则是，如果连续五次垃圾回收之后，内存占用一次比一次大，就有内存泄漏。

这就要求实时查看内存的占用情况。

**在 Chrome 浏览器中，我们可以这样查看内存占用情况**

打开开发者工具，选择 Performance 面板

在顶部勾选 Memory

点击左上角的 record 按钮

在页面上进行各种操作，模拟用户的使用情况

一段时间后，点击对话框的 stop 按钮，面板上就会显示这段时间的内存占用情况

在服务器环境中使用 Node 提供的 process.memoryUsage 方法查看内存情况

console.log(process.memoryUsage());

// {

// rss: 27709440,

// heapTotal: 5685248,

// heapUsed: 3449392,

// external: 8772

// }

复制代码process.memoryUsage返回一个对象，包含了 Node 进程的内存占用信息。

该对象包含四个字段，单位是字节，含义如下:

rss（resident set size）：所有内存占用，包括指令区和堆栈。

heapTotal："堆"占用的内存，包括用到的和没用到的。

heapUsed：用到的堆的部分。

external： V8 引擎内部的 C++ 对象占用的内存。

判断内存泄漏，以heapUsed字段为准。

**如何避免内存泄漏**

记住一个原则：不用的东西，及时归还。

减少不必要的全局变量，使用严格模式避免意外创建全局变量。

在你使用完数据后，及时解除引用（闭包中的变量，dom引用，定时器清除）。

组织好你的逻辑，避免死循环等造成浏览器卡顿，崩溃的问题。

**内存生命周期**

**JS 环境中分配的内存有如下声明周期：**

内存分配：当我们申明变量、函数、对象的时候，系统会自动为他们分配内存

内存使用：即读写内存，也就是使用变量、函数等

内存回收：使用完毕，由垃圾回收机制自动回收不再使用的内存

不再需要使用的变量也就是生命周期结束的变量，是局部变量，局部变量只在函数的执行过程中存在， 当函数运行结束，没有其他引用(闭包)，那么该变量会被标记回收。

全局变量的生命周期直至浏览器卸载页面才会结束，也就是说全局变量不会被当成垃圾回收。

**1.如何正确判断this的指向？**

谁调用它，this 就指向谁。

**1)全局环境中的 this**

浏览器环境：无论是否在严格模式下，在全局执行环境中（在任何函数体外部）this 都指向全局对象 window;

node 环境：无论是否在严格模式下，在全局执行环境中（在任何函数体外部），this 都是空对象 {};

**2)是否是 new 绑定**

如果是 new 绑定，并且构造函数中没有返回 function 或者是 object，那么 this 指向这个新对象。如下:构造函数返回值不是 function 或 object。

function Super(age){

this.age = age;

}

let instance = new Super('26');

console.log(instance.age); //26

构造函数返回值是 function 或 object，这种情况下 this 指向的是返回的对象。

function Super(age){

this.age = age;

let obj = {a: '2'};

return obj;

}

let instance = new Super('hello');

console.log(instance.age); //undefined

你可以想知道为什么会这样？我们来看一下 new 的实现原理:

1)创建一个新对象。

2)这个新对象会被执行 [[原型]] 连接。

3)属性和方法被加入到 this 引用的对象中。并执行了构造函数中的方法.

4)如果函数没有返回其他对象，那么 this 指向这个新对象，否则 this 指向构造函数中返回的对象。

function new(func) {

let target = {};

target.\_\_proto\_\_ = func.prototype;

let res = func.call(target);

//排除 null 的情况

if (res && typeof(res) == "object" || typeof(res) == "function") {

return res;

}

return target;

}

**3) 函数是否通过 call,apply 调用，或者使用了 bind 绑定，如果是，那么this绑定的就是指定的对象【归结为显式绑定】。**

这里同样需要注意一种特殊情况，如果 call,apply 或者 bind 传入的第一个参数值是 undefined 或者 null，严格模式下 this 的值为传入的值 null /undefined。非严格模式下，实际应用的默认绑定规则，this 指向全局对象(node环境为global，浏览器环境为window)

**隐式绑定，函数的调用是在某个对象上触发的，即调用位置上存在上下文对象。典型的隐式调用为: xxx.fn()**

function info(){

console.log(this.age);

}

var person = {

age: 20,

info

};

person.info();//20;执行的是隐式绑定

**默认绑定，在不能应用其它绑定规则时使用的默认规则，通常是独立函数调用。**

非严格模式： node环境，执行全局对象 global，浏览器环境，执行全局对象 window。

严格模式：执行 undefined

**4)箭头函数的this**

箭头函数没有自己的this，继承外层上下文绑定的this。

由于箭头函数不绑定this， 它会捕获其所在（即定义的位置）上下文的this值， 作为自己的this值

**2.JS中原始类型有哪几种？null 是对象吗？原始数据类型和复杂数据类型有什么区别？**

目前，JS原始类型有六种，分别为:

Boolean

String

Number

Undefined

Null

Symbol(ES6新增)

ES10新增了一种基本数据类型：BigInt

复杂数据类型只有一种: Object

null 不是一个对象，尽管 typeof null 输出的是 object，这是一个历史遗留问题，JS 的最初版本中使用的是 32 位系统，为了性能考虑使用低位存储变量的类型信息，000 开头代表是对象，null 表示为全零，所以将它错误的判断为 object 。

基本数据类型和复杂数据类型的区别为:

1)内存的分配不同

基本数据类型存储在栈中。

复杂数据类型存储在堆中，栈中存储的变量，是指向堆中的引用地址。

2)访问机制不同

基本数据类型是按值访问

复杂数据类型按引用访问，JS不允许直接访问保存在堆内存中的对象，在访问一个对象时，首先得到的是这个对象在栈内存中的地址，然后再按照这个地址去获得这个对象中的值。

3)复制变量时不同(a=b)

基本数据类型：a=b;是将b中保存的原始值的副本赋值给新变量a，a和b完全独立，互不影响

复杂数据类型：a=b;将b保存的对象内存的引用地址赋值给了新变量a;a和b指向了同一个堆内存地址，其中一个值发生了改变，另一个也会改变。

4)参数传递的不同(实参/形参)

函数传参都是按值传递(栈中的存储的内容)：基本数据类型，拷贝的是值；复杂数据类型，拷贝的是引用地址

**3.说一说你对HTML5语义化的理解**

语义化意味着顾名思义，HTML5的语义化指的是合理正确的使用语义化的标签来创建页面结构，如header,footer,nav，从标签上即可以直观的知道这个标签的作用，而不是滥用div。

语义化的优点有:

代码结构清晰，易于阅读，利于开发和维护

方便其他设备解析（如屏幕阅读器）根据语义渲染网页。

有利于搜索引擎优化（SEO），搜索引擎爬虫会根据不同的标签来赋予不同的权重

搜索引擎爬虫 （又被称为网页蜘蛛，网络机器人），是一种按照一定的规则，自动的抓取万维网信息的程序或者脚本。

SEO（Search Engine Optimization）：汉译为[搜索引擎](https://baike.baidu.com/item/%E6%90%9C%E7%B4%A2%E5%BC%95%E6%93%8E)优化。是一种方式：利用[搜索引擎](https://baike.baidu.com/item/%E6%90%9C%E7%B4%A2%E5%BC%95%E6%93%8E/104812)的规则提高[网站](https://baike.baidu.com/item/%E7%BD%91%E7%AB%99/155722)在有关搜索引擎内的[自然排名](https://baike.baidu.com/item/%E8%87%AA%E7%84%B6%E6%8E%92%E5%90%8D/2092669)。目的是：为网站提供生态式的[自我营销](https://baike.baidu.com/item/%E8%87%AA%E6%88%91%E8%90%A5%E9%94%80/1337542)解决方案，让其在行业内占据领先地位，获得[品牌](https://baike.baidu.com/item/%E5%93%81%E7%89%8C/235720)收益；SEO包含站外SEO和站内SEO两方面；为了从[搜索引擎](https://baike.baidu.com/item/%E6%90%9C%E7%B4%A2%E5%BC%95%E6%93%8E/104812)中获得更多的[免费流量](https://baike.baidu.com/item/%E5%85%8D%E8%B4%B9%E6%B5%81%E9%87%8F/13479480)，从[网站结构](https://baike.baidu.com/item/%E7%BD%91%E7%AB%99%E7%BB%93%E6%9E%84/11069102)、内容建设方案、用户[互动传播](https://baike.baidu.com/item/%E4%BA%92%E5%8A%A8%E4%BC%A0%E6%92%AD/8534525)、页面等角度进行合理[规划](https://baike.baidu.com/item/%E8%A7%84%E5%88%92/2273615)，还会使搜索引擎中显示的网站相关信息对用户来说更具有[吸引力](https://baike.baidu.com/item/%E5%90%B8%E5%BC%95%E5%8A%9B/9846334)。

语义化标签主要有：

title: 主要用于页面的头部的信息介绍

header: 定义文档的页眉

nav: 主要用于页面导航

main: 规定文档的主要内容。对于文档来说应当是唯一的。它不应包含在文档中重复出现的内容，比如侧栏、导航栏、版权信息、站点标志或搜索菜单。

article: 独立的自包含内容

h1~h6: 定义标题

ul: 用来定义无序列表

ol: 用来定义有序列表

address: 定义文档或者文章的作者/拥有者的联系信息

canvas: 用于绘制图像

dialog: 定义一个对象框、确认框或窗口

aside: 定义其所处内容之外的内容。<aside>的内容可用作文章的侧栏。

section: 定义文档中的节（section、区段）。比如章节、页眉、页脚或文档中的其他部分。

figure: 规定独立的流内容（图像、图表、照片、代码等等）。figure元素的内容应该与主内容相关，但如果被删除，则不应对文档流产生影响。

details: 描述文档或文档某一部分细节

mark: 带有记号的文本

**4.如何让 (a == 1 && a == 2 && a == 3) 的值为true？**

1）

let a = {

valueOf: (function() {

let i = 1;

//闭包的特性之一：i 不会被回收

return function() {

return i++;

}

})()

}

console.log(a == 1 && a == 2 && a == 3);

2）

let a = {

reg: /\d/g,

valueOf () {

return this.reg.exec(123)[0]

}

}

console.log(a == 1 && a == 2 && a == 3);

3）

let a = [1, 2, 3];

a.join = a.shift;

console.log(a == 1 && a == 2 && a == 3); //true

const a = {

value:[1,2,3],

valueOf: function () {return this.value.shift(); },

}

console.log(a == 1 && a == 2 && a == 3)

4）

const a = {

value:[3,2,1],

valueOf: function () {return this.value.pop(); },

}

pop() 方法用于删除数组的最后一个元素并返回删除的元素。

shift() 方法用于把数组的第一个元素从其中删除，并返回第一个元素的值。

**5.防抖(debounce)函数的作用是什么？有哪些应用场景**

函数防抖（debounce）

函数防抖，就是指触发事件后在 n 秒内函数只能执行一次，如果在 n 秒内又触发了事件，则会重新计算函数执行时间。

简单的说，当一个动作连续触发，则只执行最后一次。

防抖函数的作用就是控制函数在一定时间内的执行次数。防抖意味着N秒内函数只会被执行一次，如果N秒内再次被触发，则重新计算延迟时间。

举例说明：小思最近在减肥，但是她非常贪吃。为此，与其男朋友约定好，如果10天不吃零食，就可以购买一个包(不要问为什么是包，因为包治百病)。但是如果中间吃了一次零食，那么就要重新计算时间，直到小思坚持10天没有吃零食，才能购买一个包。所以，管不住嘴的小思，没有机会买包(悲伤的故事)...这就是防抖。

函数节流（throttle）

节流函数的作用是规定一个单位时间，在这个单位时间内最多只能触发一次函数执行，如果这个单位时间内多次触发函数，只能有一次生效。

限制一个函数在一定时间内只能执行一次。

**常见应用场景**

函数防抖的应用场景

连续的事件，只需触发一次回调的场景有：

1.搜索框输入查询，如果用户一直在输入中，没有必要不停地调用去请求服务端接口，等用户停止输入的时候，再调用，设置一个合适的时间间隔，有效减轻服务端压力。

2.表单验证

3.按钮提交事件。

4.浏览器窗口缩放，resize事件等。需窗口调整完成后，计算窗口大小。防止重复渲染。

函数节流的应用场景

间隔一段时间执行一次回调的场景有：

1.按钮点击事件

2.拖拽事件

3.onScoll

4.计算鼠标移动的距离(mousemove)

函数防抖，将多次执行的事件合并成一次。函数节流，保持一段时间执行一次。

**6. 深拷贝和浅拷贝的区别是什么？**

深拷贝

深拷贝复制变量值，对于非基本类型的变量，则递归至基本类型变量后，再复制。 深拷贝后的对象与原来的对象是完全隔离的，互不影响，对一个对象的修改并不会影响另一个对象。

浅拷贝

浅拷贝是会将对象的每个属性进行依次复制，但是当对象的属性值是引用类型时，实质复制的是其引用，当引用指向的值改变时也会跟着变化。

**可以使用 for in、 Object.assign、 扩展运算符 ... 、Array.prototype.slice()、Array.prototype.concat() 等**

可以看出浅拷贝只最第一层属性进行了拷贝，当第一层的属性值是基本数据类型时，新的对象和原对象互不影响，但是如果第一层的属性值是复杂数据类型，那么新对象和原对象的属性值其指向的是同一块内存地址。

**7. let、const、var 的区别有哪些？**

**声明方式 变量提升 暂时性死区 重复声明 块作用域有效 初始值 重新赋值**

**var 会 不存在 允许 不是 非必须 允许**

**let 不会 存在 不允许 是 非必须 允许**

**const 不会 存在 不允许 是 必须 不允许**

**8.说一说你对JS执行上下文栈和作用域链的理解？**

**JS执行上下文：**

执行上下文就是当前 JavaScript 代码被解析和执行时所在环境的抽象概念， JavaScript 中运行任何的代码都是在执行上下文中运行。

执行上下文类型分为：

全局执行上下文

函数执行上下文

eval函数执行上下文(不被推荐)

执行上下文创建过程中，需要做以下几件事:

创建变量对象：首先初始化函数的参数arguments，提升函数声明和变量声明。

创建作用域链（Scope Chain）：在执行期上下文的创建阶段，作用域链是在变量对象之后创建的。

确定this的值，即 ResolveThisBinding

**作用域**

作用域负责收集和维护由所有声明的标识符（变量）组成的一系列查询，并实施一套非常严格的规则，确定当前执行的代码对这些标识符的访问权限。—— 摘录自《你不知道的JavaScript》(上卷)

作用域有两种工作模型：词法作用域和动态作用域，JS采用的是词法作用域工作模型，**词法作用域意味着作用域是由书写代码时变量和函数声明的位置决定的**。(with 和 eval 能够修改词法作用域，但是不推荐使用，对此不做特别说明)

作用域分为：

全局作用域

函数作用域

块级作用域

**作用域链**

作用域链就是从当前作用域开始一层一层向上寻找某个变量，直到找到全局作用域还是没找到，就宣布放弃。这种一层一层的关系，就是作用域链。

**9.什么是XSS攻击，XSS攻击可以分为哪几类？我们如何防范XSS攻击？**

XSS(Cross-Site Scripting，跨站脚本攻击)是一种代码注入攻击。攻击者在目标网站上注入恶意代码，当被攻击者登陆网站时就会执行这些恶意代码，这些脚本可以读取 cookie，session tokens，或者其它敏感的网站信息，对用户进行钓鱼欺诈，甚至发起蠕虫攻击等。

XSS 的本质是：恶意代码未经过滤，与网站正常的代码混在一起；浏览器无法分辨哪些脚本是可信的，导致恶意脚本被执行。由于直接在用户的终端执行，恶意代码能够直接获取用户的信息，利用这些信息冒充用户向网站发起攻击者定义的请求。

**防范XSS攻击:**

1.Content Security Policy

在服务端使用 HTTP的 Content-Security-Policy 头部来指定策略，或者在前端设置 meta 标签。

2.输入内容长度控制

对于不受信任的输入，都应该限定一个合理的长度。虽然无法完全防止 XSS 发生，但可以增加 XSS 攻击的难度。

3.输入内容限制

对于部分输入，可以限定不能包含特殊字符或者仅能输入数字等。

4.其他安全措施

HTTP-only Cookie: 禁止 JavaScript 读取某些敏感 Cookie，攻击者完成 XSS 注入后也无法窃取此 Cookie。

验证码：防止脚本冒充用户提交危险操作。

**10.** **如何隐藏页面中的某个元素？**

屏幕并不是唯一的输出机制，比如说屏幕上看不见的元素（隐藏的元素），其中一些依然能够被读屏软件阅读出来（因为读屏软件依赖于可访问性树来阐述）。为了消除它们之间的歧义，我们将其归为三大类：

完全隐藏：元素从渲染树中消失，不占据空间。

视觉上的隐藏：屏幕中不可见，占据空间。

语义上的隐藏：读屏软件不可读，但正常占据空。

**完全隐藏**

**1.display 属性(不占据空间)**

display: none;

**2.hidden 属性 (不占据空间)**

HTML5 新增属性，相当于 display: none

<div hidden>

</div>

**视觉上的隐藏**

1. **利用 position 和 盒模型 将元素移出可视区范围**
2. **利用 transform**
3. **设置其大小为0**
4. **设置透明度为0 (占据空间)**
5. **visibility属性 (占据空间)**
6. **层级覆盖，z-index 属性 (占据空间)**
7. **clip-path 裁剪 (占据空间)**

**语义上的隐藏**

**aria-hidden 属性 (占据空间)**

读屏软件不可读，占据空间，可见。

<div aria-hidden="true">

</div>

**11. 浏览器事件代理机制的原理是什么？**

**事件流**

在说浏览器事件代理机制原理之前，我们首先了解一下事件流的概念，早期浏览器，IE采用的是事件捕获事件流，而Netscape采用的则是事件冒泡。"DOM2级事件"把事件流分为三个阶段，捕获阶段、目标阶段、冒泡阶段。现代浏览器也都遵循此规范。

**事件代理机制的原理**

事件代理又称为事件委托，在祖先级 DOM 元素绑定一个事件，当触发子孙级DOM元素的事件时，利用事件冒泡的原理来触发绑定在祖先级 DOM 的事件。因为事件会从目标元素一层层冒泡至 document 对象。

**12. setTimeout 倒计时为什么会出现误差？**

**setTimeout 只能保证延时或间隔不小于设定的时间。因为它实际上只是将回调添加到了宏任务队列中，但是如果主线程上有任务还没有执行完成，它必须要等待。**

**JS的运行机制**

（1）所有同步任务都在主线程上执行，形成一个执行栈（execution context stack）。

（2）主线程之外，还存在"任务队列"(task queue)。

（3）一旦"执行栈"中的所有同步任务执行完毕，系统就会读取"任务队列"，看看里面有哪些事件。那些对应的异步任务，于是结束等待状态，进入执行栈，开始执行。

（4）主线程不断重复上面的第三步。

**12.** **什么是闭包？闭包的作用是什么？**

闭包是指有权访问另一个函数作用域中的变量的函数，创建闭包最常用的方式就是在一个函数内部创建另一个函数。在本质上，闭包是将函数内部和函数外部连接起来的桥梁。

javascript语言的特别之处就在于：函数内部可以直接读取全局变量，但是在函数外部无法读取函数内部的局部变量。

注意点：在函数内部声明变量的时候，一定要使用var命令。如果不用的话，你实际上声明的是一个全局变量！

如何从外部读取函数内部的局部变量？

那就是在函数内部，再定义一个函数。

子对象会一级一级地向上寻找所有父对象的变量。所以，父对象的所有变量，对子对象都是可见的，反之则不成立。

既然f2可以读取f1中的局部变量，那么只要把f2作为返回值，我们不就可以在f1外部读取它的内部变量了吗！

它的最大用处有两个，一个是前面提到的可以读取函数内部的变量，另一个就是让这些变量的值始终保持在内存中，不会在f1调用后被自动清除。

**JavaScript中匿名函数this指向问题**

this对象是在运行时基于函数执行环境绑定的，在全局函数中，this=window，在函数被作为某个对象的方法调用时，this等于这个对象。

但是**匿名函数的执行环境是全局性的**。

var name = 'window';

var person = {

name : 'Alan',

sayName : function(){

return function(){

console.log(this.name);

}

}

};

person.sayName()(); // window

sayName方法return了一个匿名函数，这个匿名函数中this指向window

解决方法还是有的，我们可以把外部作用域的this传递给匿名函数

var name = 'window';

var person = {

name : 'Alan',

sayName : function(){

var that = this;

return function(){

console.log(that.name);

}

}

};

person.sayName()(); //Alan

**13. 异步加载 js 脚本的方法有哪些？**

**同步加载**

我们平时最常使用的就是这种同步加载形式：

<script src="http://yourdomain.com/script.js"></script>

同步模式，又称阻塞模式，会阻止浏览器的后续处理，停止了后续的解析，因此停止了后续的文件加载（如图像）、渲染、代码执行。

js 之所以要同步执行，是因为 js 中可能有输出 document 内容、修改dom、重定向等行为，所以默认同步执行才是安全的。

以前的一般建议是把<script>放在页面末尾</body>之前，这样尽可能减少这种阻塞行为，而先让页面展示出来。

简单说：加载的网络 timeline 是瀑布模型，而异步加载的 timeline 是并发模型。

**异步加载**又叫非阻塞，浏览器在下载执行 js 同时，还会继续进行后续页面的处理。

1. **<script> 标签中增加 async(html5) 或者 defer(html4) 属性,脚本就会异步加载。**

* **defer 和 async 的区别在于：**英 [əˈsɪŋk]

async属性是HTML5新增属性，需要Chrome、FireFox、IE9+浏览器支持

async属性规定一旦脚本可用，则会异步执行

async属性仅适用于外部脚本

此方法不能保证脚本按顺序执行

defer属性规定是否对脚本执行进行延迟，直到页面加载为止

如果脚本不会改变文档的内容，可将defer属性加入到<script>标签中，以便加快处理文档的速度

兼容所有浏览器

此方法可以确保所有设置了defer属性的脚本按顺序执行

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），在window.onload 之前执行；

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。

如果有多个 defer 脚本，会按照它们在页面出现的顺序加载

多个 async 脚本不能保证加载顺序

**2) 动态创建 script 标签**

**动态创建的 script ，设置 src 并不会开始下载，而是要添加到文档中，JS文件才会开始下载。**

let script = document.createElement('script');

script.src = 'XXX.js';

//添加到html文件中才会开始下载

document.body.append(script);

**3)$(document).ready()**

需要引入jquery

兼容所有浏览器

$(document).ready(function(){

alert("加载完成！");

})

**4) XHR 异步加载JS**

let xhr = new XMLHttpRequest();

xhr.open("get",'js/xxx.js',true);

xhr.send();

xhr.onreadystatechange = function(){

if(xhr.readyState == 4 && xhr.status == 200){

eval(xhr.responseText);

}

};

**14.** **可迭代对象有什么特点**

ES6 规定，默认的 Iterator 接口部署在数据结构的 Symbol.iterator 属性，换个角度，也可以认为，一个数据结构只要具有 Symbol.iterator 属性(Symbol.iterator 方法对应的是遍历器生成函数，返回的是一个遍历器对象)，那么就可以其认为是可迭代的。

可迭代对象的特点

具有 Symbol.iterator 属性，Symbol.iterator() 返回的是一个遍历器对象

可以使用 for ... of 进行循环

原生具有 Iterator 接口的数据结构：

Array

Map Map是一组键值对的结构，具有极快的查找速度。

Set Set和Map类似，也是一组key的集合，但不存储value。由于key不能重复，所以，在Set中，没有重复的key。

String

TypedArray

函数的 arguments 对象

NodeList 对象

**15.CSS学习笔记-内联样式、内部样式、外部样式表**

**内联样式**就是把css代码直接写在现有的HTML标签中，如下面代码：

<p style="color:red">这里文字是红色。</p>

在标签中添加内联样式。添加style属性 。在style中属性和值得表示方法为：style=“属性:值;属性:值”;

**内部样式**写在style标签中。

格式为：

<style type="text/css">

div(标签名){

属性：值;

属性：值;

}

</style>

**外联样式**：<link href="" rel="stylesheet" type="text/css" />（属于外部样式表）

三者的优先级为：内联样式>内部样式>外部 样式。

总结来说，就是--就近原则（离被设置元素越近优先级别越高）。

**16.箭头函数**

箭头函数的基本语法如下：

(参数1, 参数2, …, 参数N) => { 函数声明 }

(参数1, 参数2, …, 参数N) => 表达式（单一）

// 当只有一个参数时，圆括号是可选的：

(单一参数) => {函数声明}

单一参数 => {函数声明}

// 没有参数的函数应该写成一对圆括号。

() => {函数声明}

**如果返回一个对象，需要特别注意，如果是单表达式要返回自定义对象，不写括号会报错，因为和函数体的{ ... }有语法冲突。**

**注意，用小括号包含大括号则是对象的定义，而非函数主体**

**x => {key: x} // 报错**

**x => ({key: x}) // 正确**

**箭头函数内部的this是词法作用域，由上下文确定**。（词法作用域就是定义在词法阶段的作用域。换句话说，**词法作用域是由你在写代码时将变量和块作用域写在哪里来决定的**，因此当词法分析器处理代码时会保持作用域不变 。）

**17. JSONP原理及简单实现**

尽管浏览器有同源策略，但是 <script> 标签的 src 属性不会被同源策略所约束，可以获取任意服务器上的脚本并执行。jsonp 通过插入 script 标签的方式来实现跨域，参数只能通过 url 传入，仅能支持 get 请求。

实现原理:

Step1: 创建 callback 方法

Step2: 插入 script 标签

Step3: 后台接受到请求，解析前端传过去的 callback 方法，返回该方法的调用，并且数据作为参数传入该方法

Step4: 前端执行服务端返回的方法调用

**18. 实现一个数组去重的方法**

**法1: 利用ES6新增数据类型 Set**

Set类似于数组，但是成员的值都是唯一的，没有重复的值。

function uniq(arry) {

return [...new Set(arry)];

}

**复制代码法2: 利用 indexOf**

**f**unction uniq(arry) {

var result = [];

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

if (result.indexOf(arry[i]) === -1) {

//如 result 中没有 arry[i],则添加到数组中

result.push(arry[i])

}

}

return result;

}

**复制代码法3: 利用 includes**

function uniq(arry) {

var result = [];

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

if (!result.includes(arry[i])) {

//如 result 中没有 arry[i],则添加到数组中

result.push(arry[i])

}

}

return result;

}

**复制代码法4：利用 reduce**

function uniq(arry) {

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

}

**复制代码法5：利用 Map**

function uniq(arry) {

let map = new Map();

let result = new Array();

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

if (map.has(arry[i])) {

map.set(arry[i], true);

} else {

map.set(arry[i], false);

result.push(arry[i]);

}

}

return result;

}

**19.** **清除浮动的方法有哪些？**

当容器的高度为auto，且容器的内容中有浮动（float为left或right）的元素，在这种情况下，容器的高度不能自动伸长以适应内容的高度，使得内容溢出到容器外面而影响（甚至破坏）布局的现象。这个现象叫浮动溢出，为了防止这个现象的出现而进行的CSS处理，就叫CSS清除浮动。

**1. 利用 clear 属性**

在 <div class='outer'> 内创建一个空元素，对其设置 clear: both; 的样式。

优点：简单，代码少，浏览器兼容性好。

缺点：需要添加大量无语义的html元素，代码不够优雅，后期不容易维护。

**2. 利用 clear 属性 + 伪元素**

.outer:after{

content: '';

display: block;

clear: both;

visibility: hidden;

height: 0;

}

复制代码IE8以上和非IE浏览器才支持:after，如果想要支持IE6、7，需要给 outer 元素，设置样式 zoom: 1;

**3. 利用 BFC 布局规则**

根据 BFC 的规则，计算 BFC 的高度时，浮动元素也参与计算。因此清除浮动，只需要触发一个BFC即可。

可以使用以下方法来触发BFC

position 为 absolute 或 fixed

overflow 不为 visible 的块元素

display 为 inline-block, table-cell, table-caption

如：

.outer {

overflow: hidden;

}

复制代码注意使用 display: inline-block 会产生间隙。

**20.** **编写一个通用的柯里化函数 currying**

函数柯里化是把接受多个参数的函数变换成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下的参数而且返回结果的新函数的技术。

const currying = (fn, ...args) =>

args.length < fn.length

//参数长度不足时，重新柯里化该函数，等待接受新参数

? (...arguments) => currying(fn, ...args, ...arguments)

//参数长度满足时，执行函数

: fn(...args);

function sumFn(a, b, c) {

return a + b + c;

}

var sum = currying(sumFn);

console.log(sum(2)(3)(5));//10

console.log(sum(2, 3, 5));//10

console.log(sum(2)(3, 5));//10

console.log(sum(2, 3)(5));//10

函数柯里化的主要作用：

参数复用

提前返回 – 返回接受余下的参数且返回结果的新函数

延迟执行 – 返回新函数，等待执行

1. new的实现原理是什么？

new 的实现原理:

1）创建一个空对象，构造函数中的this指向这个空对象

2）这个新对象被执行 [[原型]] 连接

3）执行构造函数方法，属性和方法被添加到this引用的对象中

4）如果构造函数中没有返回其它对象，那么返回this，即创建的这个的新对象，否则，返回构造函数中返回的对象。



1. 如何正确判断this的指向？

如果用一句话说明 this 的指向，那么即是: 谁调用它，this 就指向谁。

但是仅通过这句话，我们很多时候并不能准确判断 this 的指向。因此我们需要借助一些规则去帮助自己：

this 的指向可以按照以下顺序判断:

**全局环境中的 this**

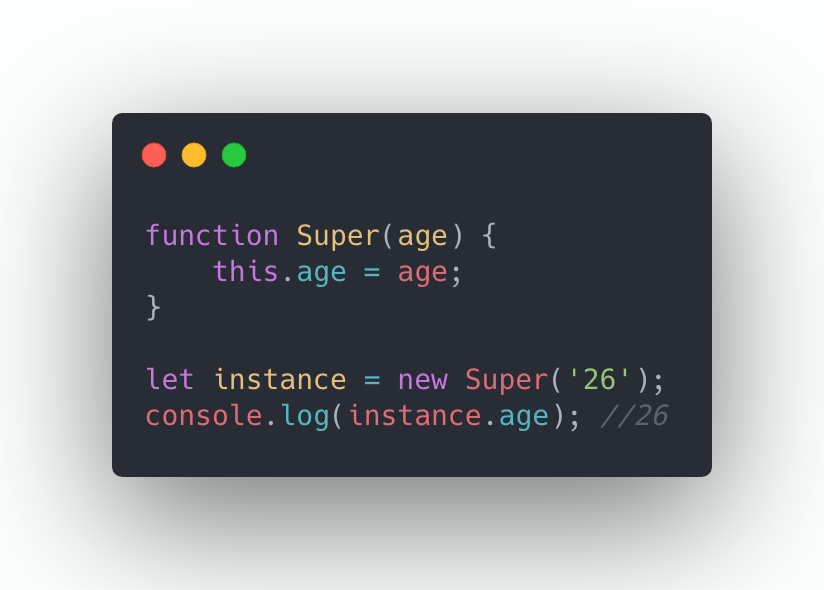
浏览器环境：无论是否在严格模式下，在全局执行环境中（在任何函数体外部）this 都指向全局对象 window;

node 环境：无论是否在严格模式下，在全局执行环境中（在任何函数体外部），this 都是空对象 {};

**是否是 new 绑定**

如果是 new 绑定，并且构造函数中没有返回 function 或者是 object，那么 this 指向这个新对象。如下:

构造函数返回值不是 function 或 object。new Super() 返回的是 this 对象。



构造函数返回值是 function 或 object，new Super()是返回的是Super中返回的对象。



**函数是否通过 call,apply 调用，或者使用了 bind 绑定，如果是，那么this绑定的就是指定的对象【归结为显式绑定】。**

这里同样需要注意一种特殊情况，如果 call,apply 或者 bind 传入的第一个参数值是 undefined 或者 null，严格模式下 this 的值为传入的值 null /undefined。非严格模式下，实际应用的默认绑定规则，this 指向全局对象(node环境为global，浏览器环境为window)



**隐式绑定，函数的调用是在某个对象上触发的，即调用位置上存在上下文对象。典型的隐式调用为: xxx.fn()**

**默认绑定，在不能应用其它绑定规则时使用的默认规则，通常是独立函数调用。**

非严格模式： node环境，执行全局对象 global，浏览器环境，执行全局对象 window。

严格模式：执行 undefined

**箭头函数的情况：**

箭头函数没有自己的this，继承外层上下文绑定的this。

1. 深拷贝和浅拷贝的区别是什么？实现一个深拷贝

深拷贝和浅拷贝是针对复杂数据类型来说的，浅拷贝只拷贝一层，而深拷贝是层层拷贝。

**深拷贝**

深拷贝复制变量值，对于非基本类型的变量，则递归至基本类型变量后，再复制。 深拷贝后的对象与原来的对象是完全隔离的，互不影响，对一个对象的修改并不会影响另一个对象。

**浅拷贝**

浅拷贝是会将对象的每个属性进行依次复制，但是当对象的属性值是引用类型时，实质复制的是其引用，当引用指向的值改变时也会跟着变化。

**浅拷贝实现**：

可以使用 for in、 Object.assign、 扩展运算符 ... 、Array.prototype.slice()、Array.prototype.concat() 等，例如:



可以看出浅拷贝只最第一层属性进行了拷贝，当第一层的属性值是基本数据类型时，新的对象和原对象互不影响，但是如果第一层的属性值是复杂数据类型，那么新对象和原对象的属性值其指向的是同一块内存地址。

**深拷贝实现**

1. **深拷贝最简单的实现是: JSON.parse(JSON.stringify(obj))**

JSON.parse(JSON.stringify(obj)) 是最简单的实现方式，但是有一些缺陷：

1）对象的属性值是函数时，无法拷贝。

2）原型链上的属性无法拷贝

3）不能正确的处理 Date 类型的数据

4）不能处理 RegExp

5）会忽略 symbol

6）会忽略 undefined

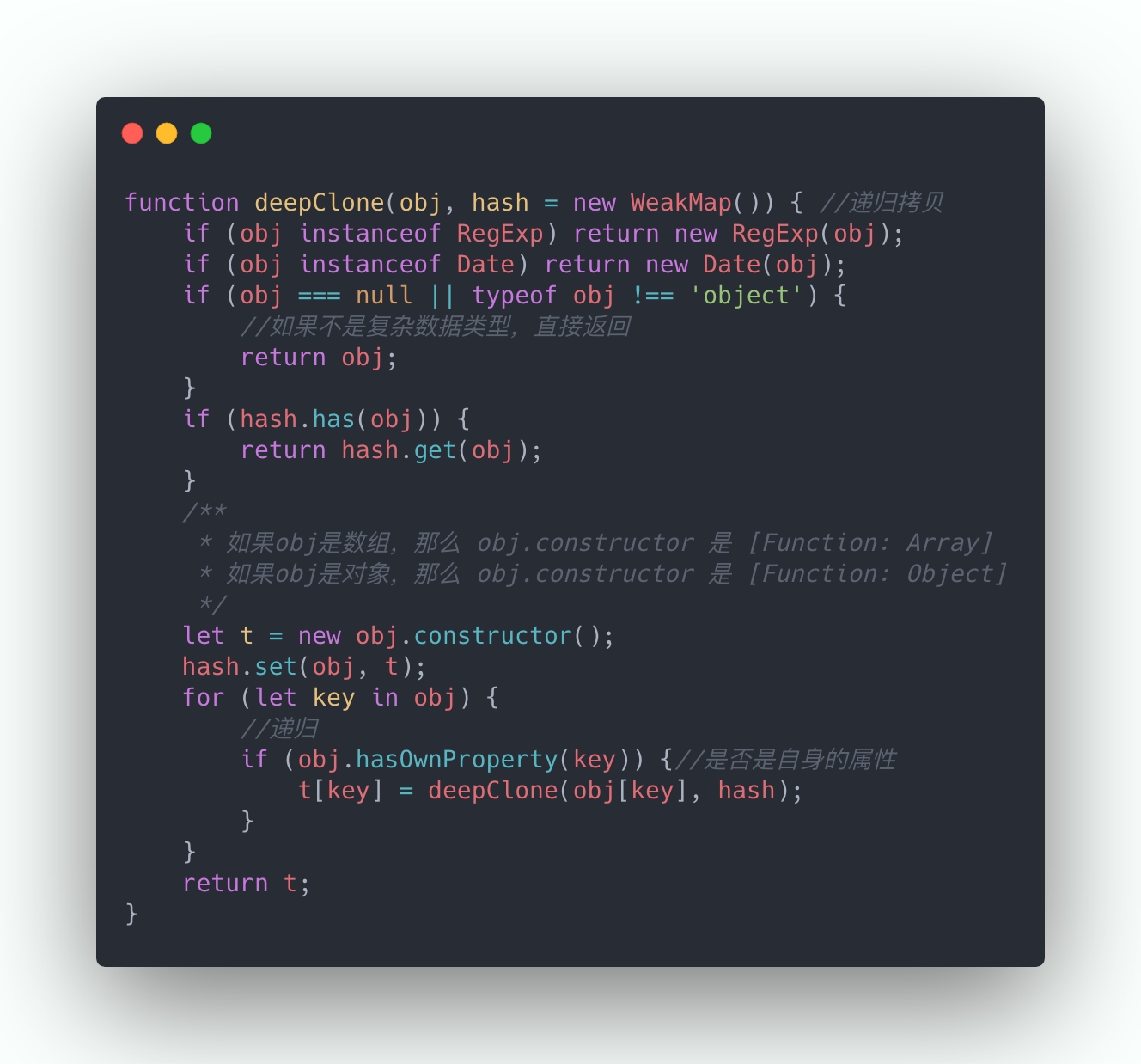
**2.实现一个 deepClone 函数**

1）如果是基本数据类型，直接返回

2）如果是 RegExp 或者 Date 类型，返回对应类型

3）如果是复杂数据类型，递归。

4）考虑循环引用的问题



1. **call/apply 的实现原理是什么？**

call 和 apply 的功能相同，都是改变 this 的执行，并立即执行函数。区别在于传参方式不同。

func.call(thisArg, arg1, arg2, ...)：第一个参数是 this 指向的对象，其它参数依次传入。

func.apply(thisArg, [argsArray])：第一个参数是 this 指向的对象，第二个参数是数组或类数组。

一起思考一下，如何模拟实现 call ？

首先，我们知道，函数都可以调用 call，说明 call 是函数原型上的方法，所有的实例都可以调用。即: Function.prototype.call。

1）在 call 方法中获取调用call()函数

2）如果第一个参数没有传入，那么默认指向 window / global(非严格模式)

3）传入 call 的第一个参数是 this 指向的对象，根据隐式绑定的规则，我们知道 obj.foo(), foo() 中的 this 指向 obj;因此我们可以这样调用函数 thisArgs.func(...args)

4）返回执行结果



**apply 的实现思路和 call 一致，仅参数处理略有差别。如下：**

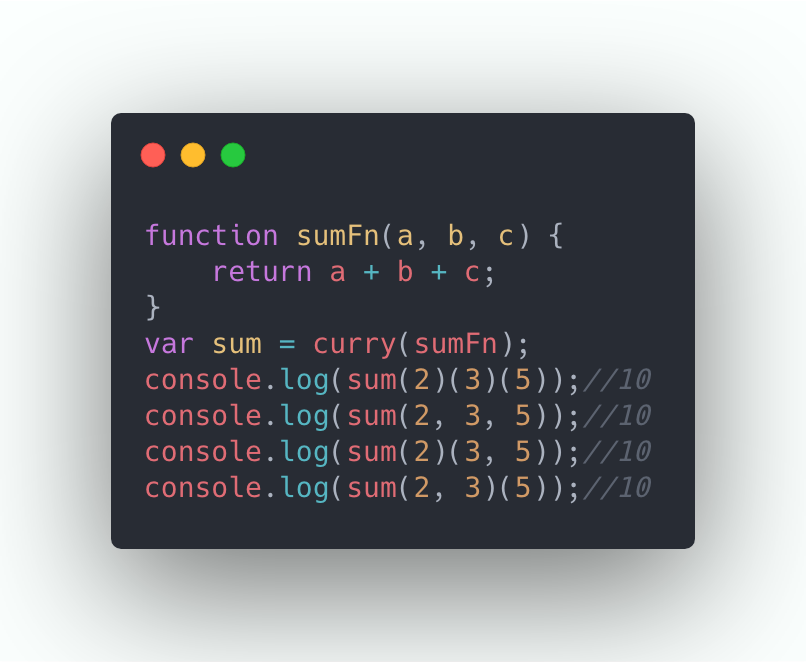


1. **柯里化函数实现**

在开始之前，我们首先需要搞清楚函数柯里化的概念。

函数柯里化是把接受多个参数的函数变换成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下的参数而且返回结果的新函数的技术。





函数柯里化的主要作用：

1）参数复用

2）提前返回 – 返回接受余下的参数且返回结果的新函数

3）延迟执行 – 返回新函数，等待执行

1. **什么是BFC？BFC的布局规则是什么？如何创建BFC？**

Box 是 CSS 布局的对象和基本单位，页面是由若干个Box组成的。

元素的类型 和 display 属性，决定了这个 Box 的类型。不同类型的 Box 会参与不同的 Formatting Context。

Formatting Context 是页面的一块渲染区域，并且有一套渲染规则，决定了其子元素将如何定位，以及和其它元素的关系和相互作用。

Formatting Context 有 BFC (Block formatting context)，IFC (Inline formatting context)，FFC (Flex formatting context) 和 GFC (Grid formatting context)。FFC 和 GFC 为 CC3 中新增。

BFC布局规则

BFC内，盒子依次垂直排列。

BFC内，两个盒子的垂直距离由 margin 属性决定。属于同一个BFC的两个相邻Box的margin会发生重叠【符合合并原则的margin合并后是使用大的margin】

BFC内，每个盒子的左外边缘接触内部盒子的左边缘（对于从右到左的格式，右边缘接触）。即使在存在浮动的情况下也是如此。除非创建新的BFC。

BFC的区域不会与float box重叠。

BFC就是页面上的一个隔离的独立容器，容器里面的子元素不会影响到外面的元素。反之也如此。

计算BFC的高度时，浮动元素也参与计算。

**如何创建BFC**

根元素

浮动元素（float 属性不为 none）

position 为 absolute 或 fixed

overflow 不为 visible 的块元素

display 为 inline-block, table-cell, table-caption

**BFC 的应用**

1)防止 margin 重叠 (同一个BFC内的两个两个相邻Box的 margin 会发生重叠，触发生成两个BFC，即不会重叠)

2)清除内部浮动 (创建一个新的 BFC，因为根据 BFC 的规则，计算 BFC 的高度时，浮动元素也参与计算)

3)自适应多栏布局 (BFC的区域不会与float box重叠。因此，可以触发生成一个新的BFC)

1. **请实现一个 flattenDeep 函数，把嵌套的数组扁平化**

**什么是数组扁平化**

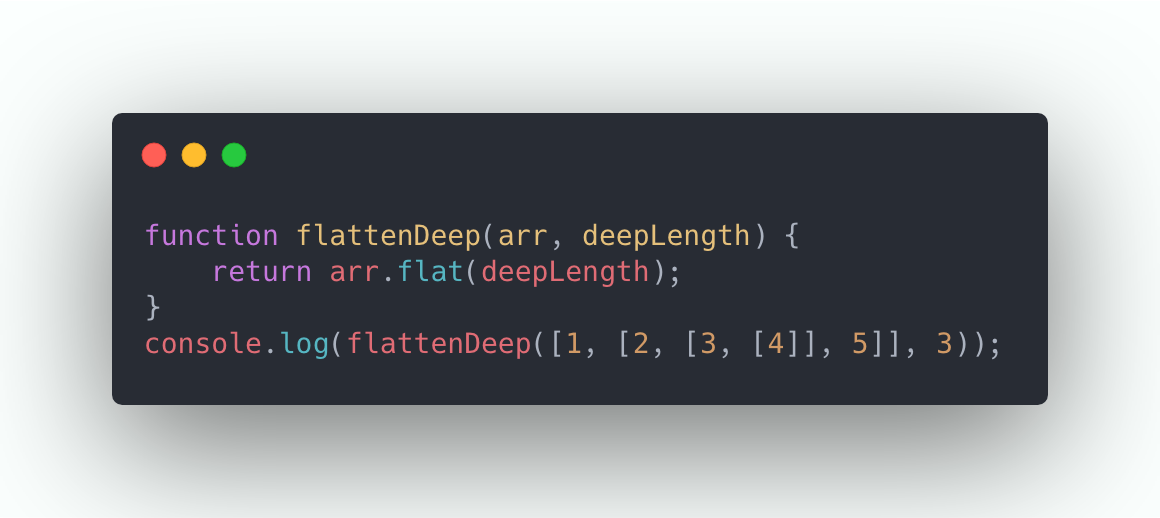
扁平化，顾名思义就是减少复杂性装饰，使其事物本身更简洁、简单，突出主题。

数组扁平化，对着上面意思套也知道了，就是将一个复杂的嵌套多层的数组，一层一层的转化为层级较少或者只有一层的数组。

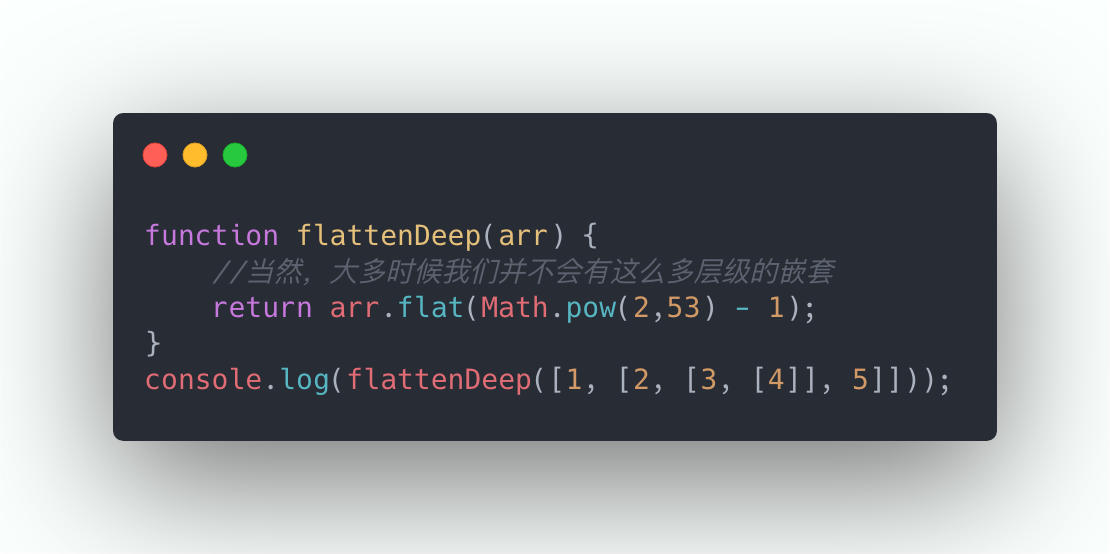
1. 利用 Array.prototype.flat

ES6 为数组实例新增了 flat 方法，用于将嵌套的数组“拉平”，变成一维的数组。该方法返回一个新数组，对原数组没有影响。

flat 默认只会 “拉平” 一层，如果想要 “拉平” 多层的嵌套数组，需要给 flat 传递一个整数，表示想要拉平的层数。



当传递的整数大于数组嵌套的层数时，会将数组拉平为一维数组，JS能表示的最大数字为 Math.pow(2, 53) - 1，因此我们可以这样定义 flattenDeep 函数



2. 利用 reduce 和 concat



3.使用 stack 无限反嵌套多层嵌套数组



4. 普通递归

/\* ES6 \*/

const flatten = (arr) => {

let result = [];

arr.forEach((item, i, arr) => {

if (Array.isArray(item)) {

result = result.concat(flatten(item));

} else {

result.push(arr[i])

}

})

return result;

};

const arr = [1, [2, [3, 4]]];

console.log(flatten(arr));

5. toString() 该方法是利用 toString 把数组变成以逗号分隔的字符串，然后遍历数组把每一项再变回原来的类型。

const flatten = (arr) => arr.toString().split(',').map((item)=>+item);

console.log(flatten(arr))

1. **ES5有几种方式可以实现继承？分别有哪些优缺点？**

**ES5 有 6 种方式可以实现继承，分别为：**

**1. 原型链继承**

原型链继承的基本思想是利用原型让一个引用类型继承另一个引用类型的属性和方法。

缺点：

1）通过原型来实现继承时，原型会变成另一个类型的实例，原先的实例属性变成了现在的原型属性，该原型的引用类型属性会被所有的实例共享。

2）在创建子类型的实例时，没有办法在不影响所有对象实例的情况下给超类型的构造函数中传递参数。

如 SubType.prototype = new SuperType();



**2. 借用构造函数**

借用构造函数的技术，其基本思想为:在子类型的构造函数中调用超类型构造函数。



优点:

可以向超类传递参数

解决了原型中包含引用类型值被所有实例共享的问题

缺点:

方法都在构造函数中定义，函数复用无从谈起，另外超类型原型中定义的方法对于子类型而言都是不可见的。

**3. 组合继承(原型链 + 借用构造函数)**

组合继承指的是将原型链和借用构造函数技术组合到一块，从而发挥二者之长的一种继承模式。

基本思路：

使用原型链实现对原型属性和方法的继承，通过借用构造函数来实现对实例属性的继承，既通过在原型上定义方法来实现了函数复用，又保证了每个实例都有自己的属性。



缺点:

无论什么情况下，都会调用两次超类型构造函数：一次是在创建子类型原型的时候，另一次是在子类型构造函数内部。



优点:

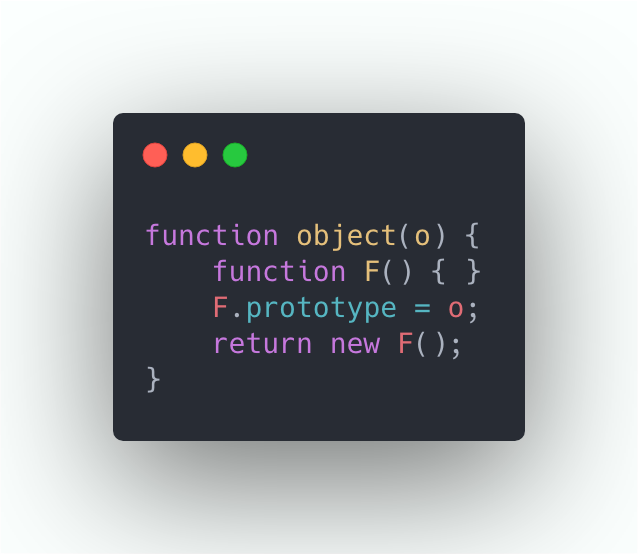
可以向超类传递参数

每个实例都有自己的属性

实现了函数复用

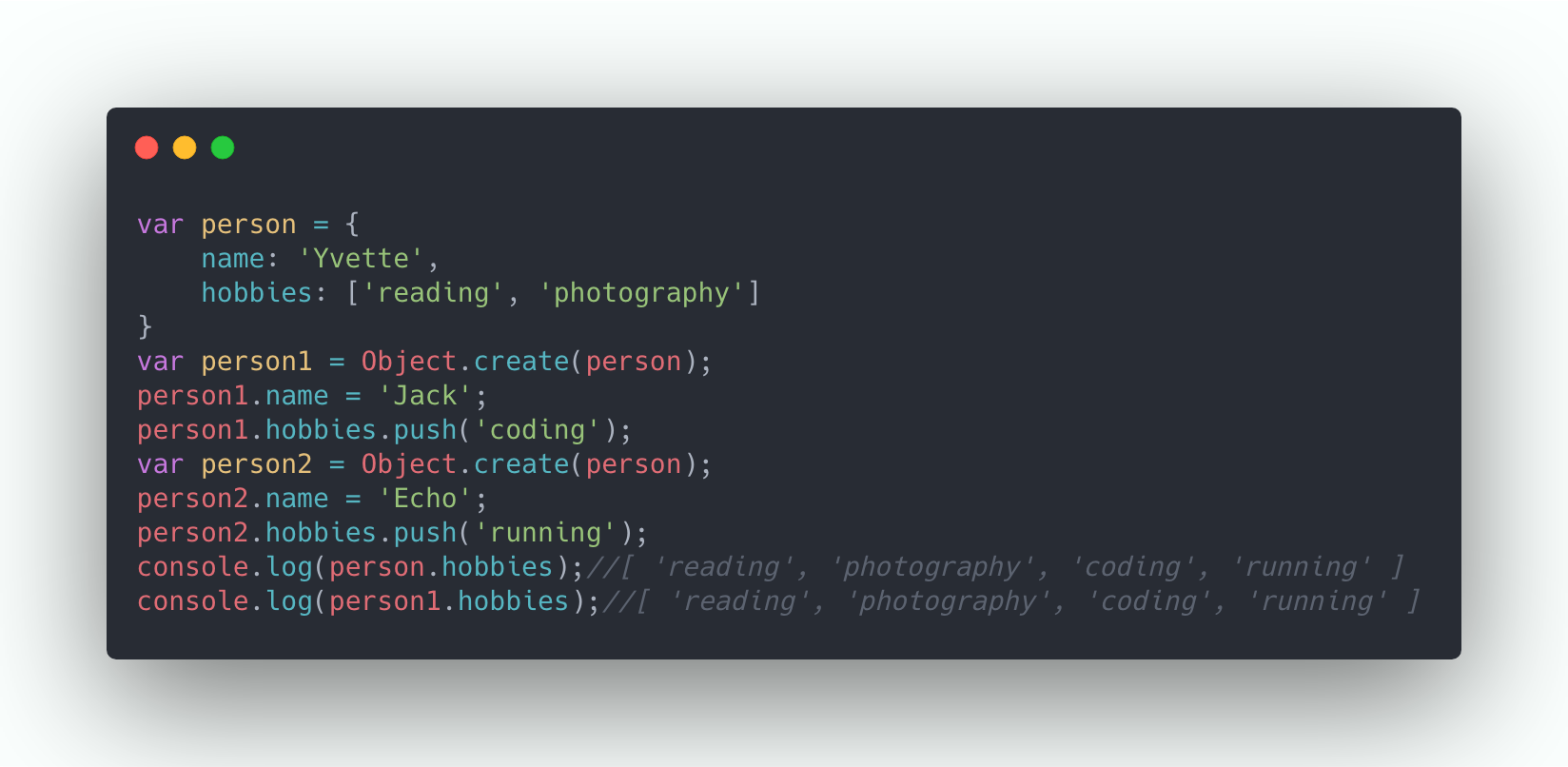
**4. 原型式继承**

原型继承的基本思想：借助原型可以基于已有的对象创建新对象，同时还不必因此创建自定义类型。



在 object() 函数内部，先穿甲一个临时性的构造函数，然后将传入的对象作为这个构造函数的原型，最后返回了这个临时类型的一个新实例，从本质上讲，object() 对传入的对象执行了一次浅拷贝。

ECMAScript5通过新增 Object.create()方法规范了原型式继承。这个方法接收两个参数：一个用作新对象原型的对象和（可选的）一个为新对象定义额外属性的对象(可以覆盖原型对象上的同名属性)，在传入一个参数的情况下，Object.create() 和 object() 方法的行为相同。



在没有必要创建构造函数，仅让一个对象与另一个对象保持相似的情况下，原型式继承是可以胜任的。

缺点:

同原型链实现继承一样，包含引用类型值的属性会被所有实例共享。

**5. 寄生式继承**

寄生式继承是与原型式继承紧密相关的一种思路。寄生式继承的思路与寄生构造函数和工厂模式类似，即创建一个仅用于封装继承过程的函数，该函数在内部已某种方式来增强对象，最后再像真地是它做了所有工作一样返回对象。



基于 person 返回了一个新对象 -—— person2，新对象不仅具有 person 的所有属性和方法，而且还有自己的 sayHi() 方法。在考虑对象而不是自定义类型和构造函数的情况下，寄生式继承也是一种有用的模式。

缺点：

使用寄生式继承来为对象添加函数，会由于不能做到函数复用而效率低下。

同原型链实现继承一样，包含引用类型值的属性会被所有实例共享。

**6. 寄生组合式继承**

所谓寄生组合式继承，即通过借用构造函数来继承属性，通过原型链的混成形式来继承方法，基本思路：

不必为了指定子类型的原型而调用超类型的构造函数，我们需要的仅是超类型原型的一个副本，本质上就是使用寄生式继承来继承超类型的原型，然后再将结果指定给子类型的原型。寄生组合式继承的基本模式如下所示：



第一步：创建超类型原型的一个副本

第二步：为创建的副本添加 constructor 属性

第三步：将新创建的对象赋值给子类型的原型

至此，我们就可以通过调用 inheritPrototype 来替换为子类型原型赋值的语句



优点:

只调用了一次超类构造函数，效率更高。避免在SuberType.prototype上面创建不必要的、多余的属性，与其同时，原型链还能保持不变。

因此寄生组合继承是引用类型最理性的继承范式。

**7.实现 ES6 的 class 语法**



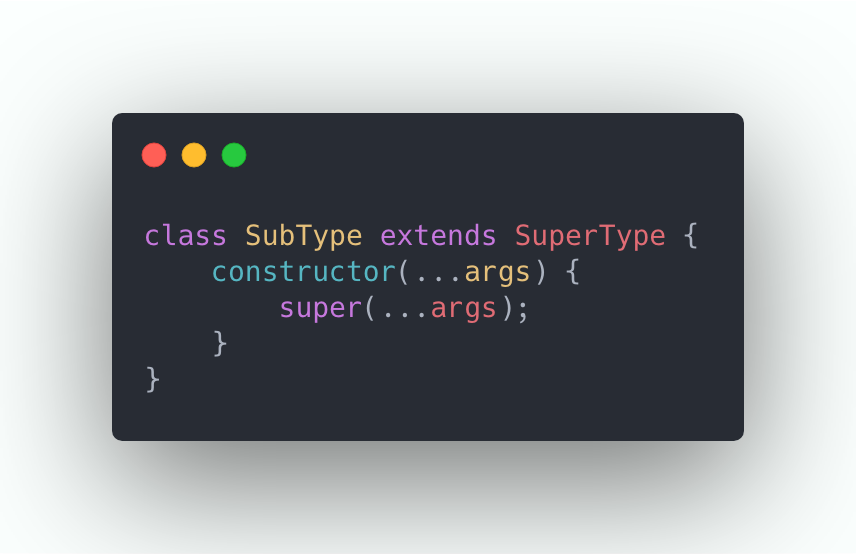
ES6 的 class 内部是基于寄生组合式继承，它是目前最理想的继承方式，通过 Object.create 方法创造一个空对象，并将这个空对象继承 Object.create 方法的参数，再让子类（subType）的原型对象等于这个空对象，就可以实现子类实例的原型等于这个空对象，而这个空对象的原型又等于父类原型对象（superType.prototype）的继承关系

而 Object.create 支持第二个参数，即给生成的空对象定义属性和属性描述符/访问器描述符，我们可以给这个空对象定义一个 constructor 属性更加符合默认的继承行为，同时它是不可枚举的内部属性（enumerable:false）

而 ES6 的 class 允许子类继承父类的静态方法和静态属性，而普通的寄生组合式继承只能做到实例与实例之间的继承，对于类与类之间的继承需要额外定义方法，这里使用 Object.setPrototypeOf 将 superType 设置为 subType 的原型，从而能够从父类中继承静态方法和静态属性

Class 可以通过extends关键字实现继承，如:





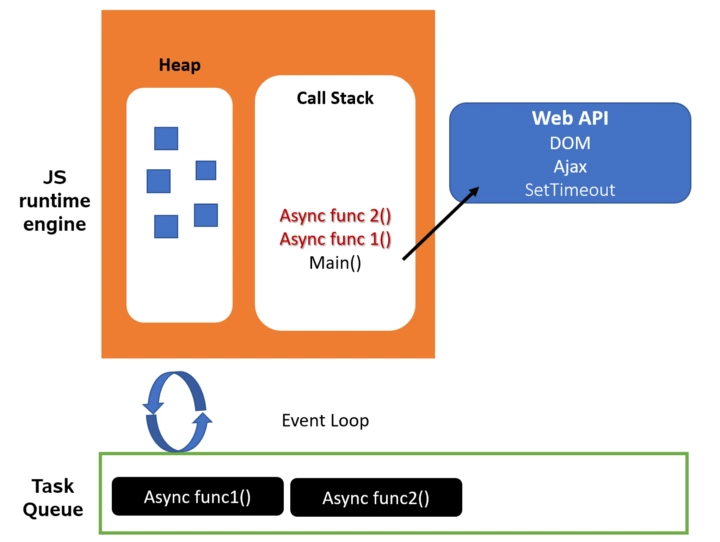
使用展开运算符合并对象和对象数组

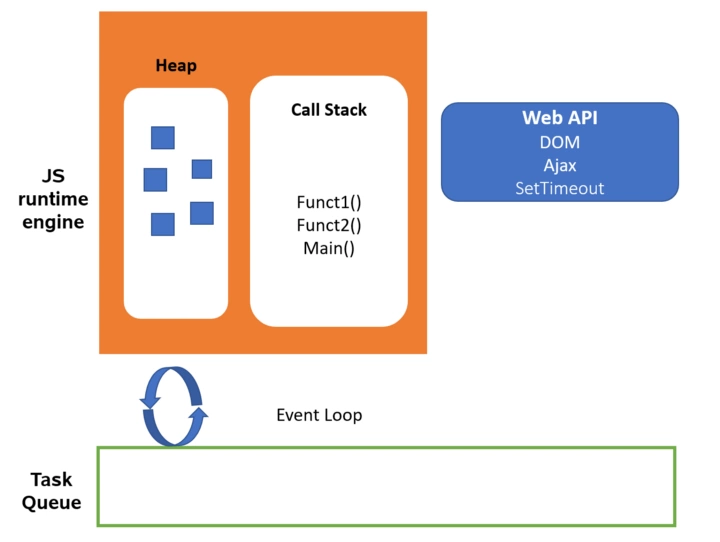
**8. 解释一下变量的提升**

变量的提升是JavaScript的默认行为，这意味着将所有变量声明移动到当前作用域的顶部，并且可以在声明之前使用变量。初始化不会被提升，提升仅作用于变量的声明。

1. **JavaScript如何处理同步和异步情况**

尽管JavaScript是一种只有一个调用堆栈的单线程编程语言，但它也可以使用一个称为\*\*事件循环(event loop)\*\*的机制来处理一些异步函数。从基本级别了解JavaScript如何工作是理解JS如何处理异步的关键部分。





如图所示，调用堆栈是定位函数的位置。一旦函数被调用，函数将被推入堆栈。然而，异步函数不会立即被推入调用堆栈，而是会被推入任务队列(Task Queue)，并在调用堆栈为空后执行。将事件从任务队列传输到调用堆栈称为事件循环。

1. **如何理解事件委托**

在DOM树上绑定事件监听器并使用JS事件处理程序是处理客户端事件响应的典型方法。 从理论上讲，我们可以将监听器附加到HTML中的任何DOM元素，但由于事件委派，这样做是浪费而且没必要的。

\*\* 什么是事件委托？\*\*

这是一种让父元素上的事件监听器也影响子元素的技巧。 通常，事件传播（捕获和冒泡）允许我们实现事件委托。 冒泡意味着当触发子元素（目标）时，也可以逐层触发该子元素的父元素，直到它碰到DOM绑定的原始监听器（当前目标）。 捕获属性将事件阶段转换为捕获阶段，让事件下移到元素; 因此，触发方向与冒泡阶段相反。 捕获的默认值为false。

**11.** **如何理解高阶函数**

JavaScript中的一切都是对象，包括函数。我们可以将变量作为参数传递给函数，函数也是如此。我们调用接受和或返回另一个函数称为高阶函数的函数。

**12. 如何区分声明函数和表达式函数**

// 声明函数

function hello() {

return "HELLO"

}

// 表达式函数

var h1 = function hello() {

return "HELLO"

}

**13. 解释一下严格模式(strict mode)**

严格模式用于标准化正常的JavaScript语义。严格模式可以嵌入到非严格模式中，关键字 ‘use strict’。使用严格模式后的代码应遵循JS严格的语法规则。例如，分号在每个语句声明之后使用。

**14. ES6模块和 CommonJS 模块有哪些差异？**

**1. CommonJS 模块是运行时加载，ES6模块是编译时输出接口。**

ES6模块在编译时，就能确定模块的依赖关系，以及输入和输出的变量。ES6 模块不是对象，它的对外接口只是一种静态定义，在代码静态解析阶段就会生成。

CommonJS 加载的是一个对象，该对象只有在脚本运行完才会生成。

**2.CommonJS 模块输出的是一个值的拷贝，ES6模块输出的是值的引用。**

- `CommonJS` 输出的是一个值的拷贝(注意基本数据类型/复杂数据类型)

- ES6 模块是动态引用，并且不会缓存值，模块里面的变量绑定其所在的模块。

**3. ES6 模块自动采用严格模式，无论模块头部是否写了 "use strict";**

**4. require 可以做动态加载，import 语句做不到，import 语句必须位于顶层作用域中。**

**5. ES6 模块的输入变量是只读的，不能对其进行重新赋值**

import name from './name';

name = 'Star'; //抛错

**6. 当使用require命令加载某个模块时，就会运行整个模块的代码。**

**7. 当使用require命令加载同一个模块时，不会再执行该模块，而是取到缓存之中的值。也就是说，CommonJS模块无论加载多少次，都只会在第一次加载时运行一次，以后再加载，就返回第一次运行的结果，除非手动清除系统缓存。**