面试题总结: <https://github.com/Advanced-Frontend/Daily-Interview-Question/blob/master/datum/summary.md>

变量一经声明，其值即为undefined

**作用域** -- 函数在创建时，便形成了作用域。

函数的作用域基于函数创建的位置。

一个函数可以访问在它的调用上下文中调用的变量，就叫词法作用域

**作用域链** -- js查找变量的时候，会先从当前上下文的变量对象中查找，如果没有找到的话，就会从父级执行上下文的对象中查找，一直找到全局上下文的变量对象，也就是全局对象，这样由多个执行上下文的变量对象构成的链表就叫做作用域链。

**箭头函数** -- 没有自己的this、super、arguments和new.target绑定，不能使用new来调用，没有原型对象，不可以改变this的绑定，形参名称不能重复

**闭包** -- 闭包是一个可以访问外部作用域的内部函数，即使这个外部作用域已经执行结束。

除了 timer 定时器，事件处理，Ajax 请求等比较常见的异步任务，还有其他的一些异步 API 比如 HTML5 Geolocation，WebSockets , requestAnimationFrame()也使用到闭包的这一特性。

闭包示例 –

function initEvents(){

for(var i=1; i<=3; i++){

$("#btn" + i).click(function showNumber(){

alert(i);//4

});

}

}

initEvents();

原因：闭包只存储外部变量的引用，而不会拷贝这些外部变量的值。

扩展：（加深记忆） -- Object 有create(返回带着指定的原型对象和属性的对象),is(a,b), keys() ,values(), entries(给定对象自身可枚举属性的键值对数组。)

apply、call作用完全一样，只是apply接受的第二个参数是数组,目的都是为了动态改变this

示例代码：

function Product(name, price) {

this.name = name;

this.price = price;

}

function Food(name, price) {

Product.call(this, name, price);

this.category = 'food';

}

function Toy(name, price) {

Product.call(this, name, price);

this.category = 'toy';

}

var cheese = new Food('feta', 5);

var fun = new Toy('robot', 40);

console.log(cheese)

console.log(fun)

打印结果：Food {name: "feta", price: 5, category: "food"}

Toy {name: "robot", price: 40, category: "toy"}

区别：Object.cerate() 必须接收一个对象参数，创建的新对象的原型指向接收的参数对象，new Object() 创建的新对象的原型指向的是 Object.prototype. （简洁点说就是前者继承指定对象， 后者继承内置对象Object）

 可以通过Object.create(null) 创建一个干净的对象，也就是没有原型，而 new Object() 创建的对象是 Object的实例，原型永远指向Object.prototype.

**Promise** -- all(),resolve(),reject()

[Promise.prototype.catch(onRejected)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Promise/catch)

[Promise.prototype.then(onFulfilled, onRejected)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Promise/then)

[Promise.prototype.finally(onFinally)](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Promise/finally)

pending: 初始状态，既不是成功，也不是失败状态。

fulfilled: 意味着操作成功完成。

rejected: 意味着操作失败

**async/await** -- 目的是简化使用多个promise时的同步行为，并对一组promises执行某些操作，在错误处理方面，async函数更容易捕获异常错误

代码：// 模拟其他语言中的 sleep，实际上可以是任何异步操作

const sleep = (timeountMS) => new Promise((resolve) => { setTimeout(resolve, timeountMS); });

(async () => { // 声明即执行的 async 函数表达式

for (var i = 0; i < 5; i++) {

if (i > 0) { await sleep(1000); }

console.log(new Date, i); }

await sleep(1000);

console.log(new Date, i);

})();

**This/apply/bind/call**

This: 在绝大多数情况下，函数的调用方式决定了this的值。

var person = {

name: "axuebin",

age: 25

};

function say(job){

console.log(this.name+":"+this.age+" "+job);

}

say.call(person,"FE"); //axuebin:25 FE

say.apply(person,["FE"]); // axuebin:25 FE

[].forEach.call(‘123456’,v=>{})

Call接受的参数取决于借调的方法需要的参数

如果要判断一个运行中函数的 this 绑定， 就需要找到这个函数的直接调用位置。 找到之后 就可以顺序应用下面这四条规则来判断 this 的绑定对象。

1. new 调用：绑定到新创建的对象，注意：显示return函数或对象，返回值不是新创建的对象，而是显式返回的函数或对象。
2. call 或者 apply（ 或者 bind） 调用：严格模式下，绑定到指定的第一个参数。非严格模式下，null和undefined，指向全局对象（浏览器中是window），其余值指向被new Object()包装的对象。
3. 对象上的函数调用：绑定到那个对象。
4. 普通函数调用： 在严格模式下绑定到 undefined，否则绑定到全局对象。

ES6 中的箭头函数：不会使用上文的四条标准的绑定规则， 而是根据当前的词法作用域来决定this， 具体来说， 箭头函数会继承外层函数，调用的 this 绑定（ 无论 this 绑定到什么），没有外层函数，则是绑定到全局对象（浏览器中是window）。 这其实和 ES6 之前代码中的 self = this 机制一样。

DOM事件函数：一般指向绑定事件的DOM元素，但有些情况绑定到全局对象（比如IE6~IE8的attachEvent）。

一定要注意，有些调用可能在无意中使用普通函数绑定规则。 如果想“ 更安全” 地忽略 this 绑 定， 你可以使用一个对象， 比如ø = Object.create(null)， 以保护全局对象。

**模拟实现call**

Function.prototype.call2 = function (context) {

var context = context || window;

context.fn = this;

var args = [];

for(var i = 1, len = arguments.length; i < len; i++) {

args.push('arguments[' + i + ']'); }

var result = eval('context.fn(' + args +')');

delete context.fn

return result;

}

var value = 2;

var obj = { value: 1 }

function bar(name, age) {

console.log(this.value);

return { value: this.value, name: name, age: age } }

bar.call(null); // 2

console.log(bar.call2(obj, 'kevin', 18));

**模拟实现new**

function Person(identity){

this.identity = identity || 'Person'

} // 封装自己的new

function \_new(Fuc) {

return function() {

var obj = { \_\_proto\_\_: Fuc.prototype }

Fuc.apply(obj, arguments) return obj } } // 封装完成，测试如下

var obj = \_new(Person)('son')

console.log(obj.constructor) // 输出结果：[Function: Person] console.log(obj.identity) // 输出结果：son

**原型prototype**

定义：给其他对象提供共享属性的对象

规范中明确描述了所有对象，都有一个隐式引用，它被称之为这个对象的 prototype 原型。

显式继承： Object.setPrototypeOf(b,a); Object.create()

**字符串转为数字的方法**： -- parseInt() Number() 直接运算 \*1 或者-0 +0

判断一个对象是不是数组用Array. isArray(obj)或者Object.prototype.toString.call(obj)

**JS事件流：** -- 首先是事件捕获阶段，从document 直到具体目标，接下来是处于目标阶段，最后是事件冒泡阶段，是从具体目标冒泡到document

**从输入URL到浏览器显示页面发生了什么：** --

首先是浏览器解析域名，查看本地硬盘的hosts文件中有没有对应域名的ip地址，没有的话，会发一个DNS请求到本地DNS(域名分布系统)服务器，本地DNS服务器,也就是移动电信联通啥的会查询有没有这个域名的缓存，没有的话，会向DNS根服务器进行查询，DNS根服务器没有记录的话，就告诉本地DNS服务器，让本地DNS服务器向域服务器发出请求，域服务器会告诉域名解析服务器的地址，然后本地DNS向域名解析服务器发出请求，然后把收到的ip地址返回到用户电脑并保存在缓存。

其次是建立TCP连接，三次握手，首先客户端先发一个带有SYN标志的数据包给服务器，然后服务器返回一个带有SYN/ACK标志的数据包给客户端，然后客户端再发送一个带有ACK标志的数据包给服务器，三次握手结束。

建立TCP连接之后，会发起HTTP请求，请求一般分为三部分，请求方法，请求头，请求正文。

为了避免资源的占用和损耗，任意一方都可以发起关闭请求，需要四次握手，客户端传带FIN标志的数据包给服务器，服务器返回带ACK标志的数据包给客户端。服务器再发带FIN标志的数据包给客户端，客户端返回带ACK标志的数据包给服务器。

之后浏览器会解析获取到的HTML、CSS、JS、图片等资源。浏览器通过解析HTML，生成DOM树；解析CSS，生成CSS规则树，然后通过DOM树和CSS规则书生成渲染树。解析CSS的同时，可以继续加载解析HTML，但在执行JS脚本时，会停止解析后续的HTML，就会出现阻塞问题。

浏览器布局渲染。根据渲染树布局，计算CSS样式，即每个节点在页面的大小和位置等几盒信息。HTML默认是流式布局的，CSS和JS会打破这种布局，改变DOM的外观样式以及大小位置。 Repaint重绘，不影响整体布局。Reflow重排，元件的几何尺寸变了，需要重新验证并计算渲染树。

**数组去重的方法：** --

1.用ES6 Set去重。

**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))

2.用for嵌套for 然后splice去重

**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',{},{}];

3.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))

**Js方法：**

Pop() 删除原数组最后一项，并返回删除元素的值

push() 将参数添加到原数组末尾，并返回删除元素的值

unshift() 将参数添加到原数组开头，并返回数组长度

splice(start,deleteCount,val1,val2) 从start位置开始删除deleteCount项，并从该位置起插入val1,val2

shift() 删除原数组的第一项，并返回删除元素的值

判断整数-Number.isInteger()方法

**事件循环Event Loop：**

JS 主线程拥有一个 执行栈（同步任务） 和 一个 任务队列（microtasks queue），主线程会依次执行代码，

当遇到函数（同步）时，会先将函数入栈，函数运行结束后再将该函数出栈；当遇到task任务（异步）时，这些 task 会返回一个值，让主线程不在此阻塞，使主线程继续执行下去，而真正的task任务将交给 浏览器内核 执行，浏览器内核执行结束后，会将该任务事先定义好的回调函数加入相应的\*\*任务队列（microtasks queue/ macrotasks queue）\*\*中。

当JS主线程清空执行栈之后，会按先入先出的顺序读取microtasks queue中的回调函数，并将该函数入栈，继续运行执行栈，直到清空执行栈，再去读取任务队列。

当microtasks queue中的任务执行完成后，会提取 macrotask queue 的一个任务加入 microtask queue， 接着继续执行microtask queue，依次执行下去直至所有任务执行结束。

这就是 JS的异步执行机制

首先，执行栈开始顺序执行判断是否为同步，异步则进入异步线程，最终事件回调给事件触发线程的任务队列等待执行，同步继续执行执行栈空，询问任务队列中是否有事件回调任务队列中有事件回调则把回调加入执行栈末尾继续从第一步开始执行任务队列中没有事件回调则不停发起询问

常见的宏任务：

主代码块setTimeout、setInterval、setImmediate ()-Node、requestAnimationFrame ()-浏览器

常见的微任务：

process.nextTick ()-Node

Promise.then().Catch.finally、Object.observe、MutationObserver

**获取精度更高的时间**

浏览器使用 performance.now() 可以获取到 performance.timing.navigationStart 到当前时间之间的微秒数

Node.js 使用 process.hrtime 返回一个数组，其中第一个元素的时间以秒为单位，第二个元素为剩余的纳秒

**git rebase 和 merge 的区别**

主要的区别在于是否保留分支的 commit 提交节点，rebase 会给你一个简洁的线性历史树

**防抖、节流：**

防抖: 触发高频事件后n秒内函数只会执行一次，如果n秒内高频事件再次被触发，则重新计算时间。

function debounce(fn) {

let timeout = null; // 创建一个标记用来存放定时器的返回值

return function () {

clearTimeout(timeout); // 每当用户输入的时候把前一个 setTimeout clear 掉

timeout = setTimeout(() => { // 然后又创建一个新的 setTimeout, 这样就能保证输入字符后的 interval 间隔内如果还有字符输入的话，就不会执行 fn 函数

fn.apply(this, arguments);

}, 500);

};

}

function sayHi() {

console.log('防抖成功');

}

var inp = document.getElementById('inp');

inp.addEventListener('input', debounce(sayHi)); // 防抖

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

function throttle(fn) {

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;

}, 500);

};

}

function sayHi(e) {

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

}

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

**js创建对象的模式**

工厂模式：

function createPerson(name, age, job){

var o = new Object();

o.name = name;

o.age = age;

o.job = job;

o.sayName = function(){

alert(this.name);

};

return o;

}

var person1 = createPerson("Nicholas", 29, "Software Engineer");

var person2 = createPerson("Greg", 27, "Doctor");

构造函数模式：

function Person(name, age, job){

this.name = name;

this.age = age;

this.job = job;

this.sayName = function(){

alert(this.name);

};

}

var person1 = new Person("Nicholas", 29, "Software Engineer");

var person2 = new Person("Greg", 27, "Doctor");

原型模式：

function Person(){

}

Person.prototype.name = "Nicholas";

Person.prototype.age = 29;

Person.prototype.job = "Software Engineer";

Person.prototype.sayName = function(){

alert(this.name);

};

var person1 = new Person();

person1.sayName(); //"Nicholas"

var person2 = new Person();

person2.sayName(); //"Nicholas"

alert(person1.sayName == person2.sayName); //true

外观模式：

外观模式最大的体现其实就是入口，比如init()函数，把一些内部的函数都放在这个门面之下，只需要调用这个门面函数，其他乱七八糟的功能都可以实现。外观模式不仅简化类中的接口，而且对接口与调用者也进行了解耦。外观模式经常被认为开发者必备，它可以将一些复杂操作封装起来，并创建一个简单的接口用于调用。

说白了就是用一个接口封装其它的接口。

外观模式优点就是易使用。缺点则是，当连续使用外观模式创建的接口时，可能会产生性能问题。

function YaSuo() {

}

function Wskill (hero) {

hero.prototype.Wskill = function() {

console.log('风墙')

}

}

function Rskill (hero) {

hero.prototype.Rskill = function() {

console.log('痛里唉该痛')

}

}

function Skin (hero, skin) {

hero.prototype.skin = skin

}

function CreateYasuo () {

Wskill(YaSuo)

Rskill(YaSuo)

Skin(YaSuo, 'originSkin')

return new YaSuo()

}

CreateYasuo()

**JS设计模式：**

1. 单例模式: 确保只有一个实例、可以全局访问
2. const singleton = function(name) {
3. this.name = name
4. this.instance = null
5. }
6. singleton.prototype.getName = function() {
7. console.log(this.name)
8. }
9. singleton.getInstance = function(name) {
10. if (!this.instance) { // 关键语句
11. this.instance = new singleton(name)
12. }
13. return this.instance
14. }
15. // test
16. const a = singleton.getInstance('a') // 通过 getInstance 来获取实例
17. const b = singleton.getInstance('b')
18. console.log(a === b)

事实上都是第一个执行的实例。

**VUE深入响应式原理：**

用了 ES5 的 Object.defineProperty劫持getter,setter方法

**为什么需要HTTPS：**

http存在的问题：

通信使用明文（不加密），内容可能被窃听。

无法证明报文的完整性，所以可能遭篡改

不验证通信方的身份，因此有可能遭遇伪装

**移动端1Px解决方案：**

使用伪元素：

.setOnePx{

position: relative;

&::after{

position: absolute; content: '';

background-color: #e5e5e5;

display: block; width: 100%;

height: 1px; /\*no\*/

transform: scale(1, 0.5);

top: 0;

left: 0; } }

**word-break和white-space: normal**

都可以换行

**项目重点、难点**

**路由控制：**

**全局指令：**

**仿微信SDK：**

**权限控制：**

查看端口占用 netstat –lntp 服务器部署项目时要用pm2保持启动，命令为 pm2 start index.js 如果不管用 就用pm2先杀进程，再重新start