## JS相关

### 1. 数组转成字符串的方式

1. 使用toString方法 转成字符串

例如 var arr = [1,2,3]

arr.toString() == '1,2,3';

1. 使用join()方法 转成字符串

例如 var arr = [1,2,3]

arr.join(',') == '1,2,3'; 还可以自由控制需要什么隔开例如 - , 如果不需要什么隔开就写空 arr.join('') == '123';

### 2. JS数组的排序方式

1. 使用数组的sort函数可以实现排序

但是默认sort是按照ASCII编码排序的 如果需要按照数值的从小到大 从大到小需要使用第二个参数 是一个回调函数 回调函数需要两个参数 用来说明你要按照什么方式排序

例如a,b  
如果 a - b < 0 表示 a 比 b小 那么真实的a代表的数字就会排在b前面  
如果 a - b == 0 那么表示2个值相等 就按顺序排序 如果 a - b > 0 表示 a比b大 那么真实a代表的数组就会排在b后面

function sortNumber(a, b) {  
   
 return a - b // 从小到大  
 // return b - a // 从大到小  
}  
  
var arr = [10, 5, 40, 25, 1000, 1]  
  
document.write(arr + "<br />")  
document.write(arr.sort(sortNumber));// 1 5 10 25 40 1000 从小到大

1. 自己使用算法实现排序 例如 冒泡排序

var arr = [10, 20, 1, 2];  
var t;  
for(var i=0;i<arr.length;i++){  
 for(j=i+1;j<arr.length;j++){  
 if(arr[i]>arr[j]){  
 t=arr[i];  
 arr[i]=arr[j];  
 arr[j]=t;  
 }  
 }  
}  
console.log(arr); //[1, 2, 10, 20]

1. 快速排序

// 快速排序  
var quickSort = function(arr) {  
    if(arr.length < 1) {//如果数组就是一项，那么可以直接返回  
        return arr;  
    }  
    var centerIndex = Math.floor(arr.length / 2);//获取数组中间的索引  
    var centerValue = arr[centerIndex];//获取数组中间项  
    var left = [], right = [];  
    for(var i = 0; i < arr.lenght; i++){  
        if(arr[i] < centerValue){  
            left.push(arr[i]);  
        }else{  
            right.push(arr[i]);  
        }  
    }  
    return quickSort(left).contanct([centerValue], quickSort(right));//递归调用  
}

### 3. JSON转换

1. 将JS对象转成JSON字符串 var str = JSON.stringify(obj);
2. 将JSON字符串转成JS对象

var obj = JSON.parse(str);

### 4. 对原型链的理解

1. 什么是原型链 简单来说就是对象和构造函数之间连接的一个链条 简称原型链
2. 原型链的作用 可以通过原型链继承构造函数对象定义的一些公共的属性方法
3. 原型链的流程图

* 原型链流程图
* 原型链流程图

### 5. JS变量提升和自执行函数的面试题

(function(){  
 console.log(a);// undefined 虽然此时在函数内打印有a 因为var变量声明会提前所以不会报错 但是赋值不会提前所以a没有值是 undefined  
 var a = b = 1;   
})();   
console.log(b);//1 因为 var a = b = 1 其实b并没在里面var所以 b是全局变量并且赋值为了1所以是1  
console.log(a);// 因为 var a = b = 1; 这个var是在自执行函数里面的自执行函数是一个函数作用域 所以var a是函数内的局部变量所以外界拿不到 即是报错 a is not defined

### 6. JS this指向的面试题

var name = '王五';  
var obj = {  
 name: '张三',  
 getName:function(){  
 return this.name;  
 },  
 children:{  
 name:'李四',  
 getName:function(){  
 return this.name;  
 }  
 }  
}   
console.log(obj.getName());//张三  
console.log(obj.children.getName());//李四  
var getName = obj.children.getName;  
console.log(getName());//王五  
// 那么如果我想调用getName() 但是想打印张三怎么办 可以使用call函数改变调用当前函数的this指针  
console.log(getName.call(obj));//张三

### 7. 使用JS的类实现一个栈的类 并实现栈里面的常用API

// 定义一个栈的类 实现栈的push pop getMax getMin等函数   
// 并实现先进后出 后进先出 而且能够不断求出 当前栈里面的数据的最大值最小值  
class Stack {  
 constructor(){  
 this.dataStack = [];  
 this.maxStack = [];  
 this.minStack = [];  
  
 }  
 // 入栈  
 push(item){  
 if(this.minStack.length <= 0){  
 this.minStack.push(item)  
 }  
 else if(item <= this.getMin()){  
 this.minStack.push(item)  
 }  
  
 if(this.maxStack.length <= 0){  
 this.maxStack.push(item)  
 }  
 else if(item >= this.getMax()){  
 this.maxStack.push(item)  
 }  
 this.dataStack.push(item);   
 }  
 // 出栈  
 pop(){  
 if(this.dataStack.length <= 0){  
 return   
 }  
 let value = this.dataStack.pop();  
   
 if(value == this.getMin()){  
 this.minStack.pop()  
 }  
 if(value == this.getMax()){  
 this.maxStack.pop();  
 }  
 return value;  
 }  
 // 求最小值  
 getMin(){   
 if(this.minStack.length <=0){  
 return 0;  
 }  
 return this.minStack[this.minStack.length-1];  
 }  
 // 求最大值  
 getMax(){   
 if(this.maxStack.length <=0){  
 return 0;  
 }   
 return this.maxStack[this.maxStack.length-1];  
 }  
}   
// 创建栈的对象  
var stack = new Stack();  
// 调用入栈  
stack.push(3);  
stack.push(5);  
stack.push(10);  
stack.push(7);  
stack.push(6);  
// 调用求最大值 最小值的函数  
console.log(stack.dataStack);   
console.log(stack.getMax());  
console.log(stack.getMin());  
// 出栈  
stack.pop();  
stack.pop();  
// 出栈后的最大值最小值  
console.log(stack.getMax());  
console.log(stack.getMin());  
console.log(stack.minStack);  
console.log(stack.maxStack);  
stack.pop();  
console.log(stack.getMax());  
console.log(stack.getMin());  
console.log(stack.minStack);  
console.log(stack.maxStack);  
stack.pop();  
console.log(stack.getMax());  
console.log(stack.getMin());  
console.log(stack.dataStack);  
console.log(stack.maxStack);  
stack.pop();  
console.log(stack.getMax());  
console.log(stack.getMin());  
console.log(stack.dataStack);  
console.log(stack.maxStack);

### 8. JS笔试题对象赋值属性考察

var fun = function(){  
  
 }  
 fun.prototype = {  
 info:{  
 name:'张三',  
 age:20  
 }  
 }  
 var a = new fun();  
 var b = new fun();  
   
 a.info.name = '李四';  
 b.info.name = '王五';  
 console.log(a.info.name) // 王五  
 console.log(b.info.name) // 王五   
 // 为何当前两个对象的info.name都输出王五 因为构造函数fun 并没有给new出来的每个实例对象赋予单独的info 对象这个属性   
 // 因为原型上的对象属性继承给实例后其实是把对象的内存地址和a.info b.info关联起来 其实a.info b.info都指的是同一个info对象  
 // 那么a.info.name 和 b.info.name都找的是fun原型属性上的info   
 // 所以修改的是同一个值 取出来也是同一个值 故都是王五  
  
 var fun = function () {  
  
 }  
 fun.prototype = {  
 name: '张三',  
 age: 20  
 }  
 var a = new fun();  
 var b = new fun();  
 a.name = '李四';  
 b.name = '王五';  
 console.log(a.name) // 李四  
 console.log(b.name) // 王五  
 // 为何此时打印的a 和 b的name又分别是 李四 和 王五   
 // 因为name是普通属性 构造函数创建实例后 原型上的普通属性会继承到实例对象上会形成2个不同的变量 所以a.name 和 b.name 一开始虽然值一样但是他们代表的变量不一样  
 // 所以赋值改变不同后 a.name 和 b.name的属性是改变后的属性  
  
 var fun = function () {  
 this.info = {  
 name: '张三',  
 age: 20  
 }  
 }  
  
 var a = new fun();  
 var b = new fun();  
 a.info.name = '李四';  
 b.info.name = '王五';  
 console.log(a.info.name)// 李四  
 console.log(b.info.name)// 王五  
 // 为何此时打印的又是李四 王五 因为 fun构造函数里面的 this当前实例对象 就是分别是 a 和 b 是两个不一样的实例对象   
 // 是分别给实例对象添加info对象属性 那么此时 a.info 和 b.info 是分别属于对象自己的 a.info.name 和 b.info.name也是不同的2个属性  
 // 所以赋值取值后也是分别的

### 9. JS面试题 严格模式和非严格模式的考察

(function(){  
   
 var a = b = 10;  
 })();  
   
 console.log(b) // 10  
 // 为什么是b是10 呢 因为 上面的表达式虽然是写在一个只执行函数里面(闭包函数里面) 理论上 a和b在外面是无法访问到的  
 // 但是由于表达式是 var a = b = 10; 其实只声明了a 并没有var声明b 所以a是函数内的局部变量   
 // 但是b由于没有被声明 在非严格模式下 其实是一个全局变量 所以外部也能访问到打印就是10  
 (function(){  
 'use strict'  
 var a = b = 10;  
   
 })();  
  
 console.log(b) // 报错 b is not defined  
 // 为什么是此时都是未定义呢 因为 上面的表达式虽然是写在一个只执行函数里面(闭包函数里面) 理论上 a和b在外面是无法访问到的  
 // 但是由于表达式是 var a = b = 10; 其实只声明了a 并没有var声明b 所以a是函数内的局部变量   
 // 但是b由于没有被声明 但是当前模式是严格模式 严格模式下不允许使用未声明的变量 所以b是不存在的 无论在哪里打印都会报错不存在

### 10. 函数声明和函数表达式的区别

(function(){  
 console.log(typeof foo) // function 为何此时foo打印的是function 因为 函数声明会提前虽然在后面定义的函数foo其实声明会在最前面  
 console.log(typeof bar) // undefined 为何此时打印bar是undefined 因为bar是 函数表达式 相当于把函数通过表达式赋值给变量bar 表达式赋值不会提前所以此时还未定义就是undefined  
 var foo = 'hello',  
 bar = function(){  
 return 'world';  
 }  
 function foo(){  
 return 'hello'  
 }  
 })()

### 11. JS中什么是伪数组 有哪些常见的伪数组 如何转成普通数组

伪数组: 就是拥有数组的可以存储多个值的特性并拥有长度length属性 但是不拥有普通数组的内置方法 例如push pop slice等方法 哪些是常见的伪数组 DOM元素结构 jq获取的元素 这些都是伪数组

伪数组转化成普通数组 需要用到数组原型中的方法 slice

//需要使用到call方法，因为arr伪数组并没有该方法，需要使用call来借调 Array对象的slice方法 console.log(Array.prototype.slice.call(arr));

或者使用ES6的 console.log(Array.from(arr)) 函数也可以转化成普通数组

### 12. 函数返回值和 函数里面的this指向问题

```js var x = 20; var a = { x:15, fun:function(){ var x = 30; return function(){ return this.x; }

}  
}  
console.log(a.fun()) // ƒ (){return this.x;} a.fun的调用后的返回值是一个函数 所以输出就是当前返回的这个函数  
console.log((a.fun())()) // 20 a.fun() 调用后返回的函数再被表达式()包裹了起来 然后再调用 就是把当前返回的函数重新调用 但是当前返回的函数并没有赋值给谁那么调用者this指向window window的x是20  
console.log(a.fun()()) // 20 a.fun() 调用后返回的函数 然后再调用 就是把当前返回的函数重新调用 但是当前返回的函数并没有赋值给谁那么调用者this指向window window的x是20  
console.log(a.fun()() == (a.fun())()) // true 因为两种调用方法返回的结果都是20 所以为true  
console.log(a.fun().call(this)); // 20 将返回的函数使用call然后再传入this 此时的this还是window 所以函数里面的this也是window 所以还是外面的20  
console.log(a.fun().call(a)); // 15 将返回的函数call然后传入a 此时的函数里面的this指向的 当前对象a a有一个x为15 所以是a的15

```

### 13. 定义一个JS数组去重的方法

```js //1: 思路：定义一个新数组，并存放原数组的第一个元素，然后将元素组一一和新数组的元素对比，若不同则存放在新数组中。

function unique(arr) {  
 let newArr = [arr[0]];  
 for (let i = 1; i < arr.length; i++) {  
 let repeat = false;  
 for (let j = 0; j < newArr.length; j++) {  
 if (arr[i] === newArr[j]) {  
 repeat = true;  
 break;  
 }else{  
   
 }  
 }  
 if (!repeat) {  
 newArr.push(arr[i]);  
 }  
 }  
 return newArr;  
 }  
  
 console.log(unique([1, 1, 2, 3, 5, 3, 1, 5, 6, 7, 4]));  
 // 结果是[1, 2, 3, 5, 6, 7, 4]  
 //2: 思路：先将原数组排序，在与相邻的进行比较，如果不同则存入新数组。  
  
 function unique2(arr) {  
 var formArr = arr.sort()  
 var newArr=[formArr[0]]  
 for (let i = 1; i < formArr.length; i++) {  
 if (formArr[i]!==formArr[i-1]) {  
 newArr.push(formArr[i])  
 }  
 }  
 return newArr  
 }  
 console.log(unique2([1, 1, 2, 3, 5, 3, 1, 5, 6, 7, 4]));  
 // 结果是[1, 2, 3, 4,5, 6, 7]  
  /3: 利用对象属性存在的特性，如果没有该属性则存入新数组。  
  
 function unique3(arr) {  
 var obj={}  
 var newArr=[]  
 for (let i = 0; i < arr.length; i++) {  
 if (!obj[arr[i]]) {  
 obj[arr[i]] = 1  
 newArr.push(arr[i])  
 }   
 }  
 return newArr  
 }  
 console.log(unique2([1, 1, 2, 3, 5, 3, 1, 5, 6, 7, 4]));  
 // 结果是[1, 2, 3, 5, 6, 7, 4]  
 //4: 利用数组的indexOf下标属性来查询。  
  
 function unique4(arr) {  
 var newArr = []  
 for (var i = 0; i < arr.length; i++) {  
 if (newArr.indexOf(arr[i])===-1) {  
 newArr.push(arr[i])  
 }  
 }  
 return newArr  
 }  
 console.log(unique4([1, 1, 2, 3, 5, 3, 1, 5, 6, 7, 4]));  
 // 结果是[1, 2, 3, 5, 6, 7, 4]  
 //5: 利用数组原型对象上的includes方法。  
  
 function unique5(arr) {  
 var newArr = []  
 for (var i = 0; i < arr.length; i++) {  
 if (!newArr.includes(arr[i])) {  
 newArr.push(arr[i])  
 }  
 }  
 return newArr  
 }  
 console.log(unique5([1, 1, 2, 3, 5, 3, 1, 5, 6, 7, 4]));  
 // 结果是[1, 2, 3, 5, 6, 7, 4]  
 //6: 利用数组原型对象上的 filter 和 includes方法。  
  
 function unique6(arr) {  
 var newArr = []  
 newArr = arr.filter(function (item) {  
 return newArr.includes(item) ? '' : newArr.push(item)  
 })  
 return newArr  
 }  
 console.log(unique6([1, 1, 2, 3, 5, 3, 1, 5, 6, 7, 4]));  
 // 结果是[1, 2, 3, 5, 6, 7, 4]  
 //7: 利用数组原型对象上的 forEach 和 includes方法。  
  
 function unique7(arr) {  
 var newArr = []  
 array.forEach(item => {  
 return newArr.includes(item) ? '' : newArr.push(item)  
 });  
 return newArr  
 }  
 console.log(unique7([1, 1, 2, 3, 5, 3, 1, 5, 6, 7, 4]));  
 // 结果是[1, 2, 3, 5, 6, 7, 4]  
 //8: 利用数组原型对象上的 splice 方法。  
  
 function unique8(arr) {  
 var i,j,len = arr.length;  
 for (i = 0; i < len; i++) {  
 for (j = i + 1; j < len; j++) {  
 if (arr[i] == arr[j]) {  
 arr.splice(j, 1);  
 len--;  
 j--;  
 }  
 }  
 }  
 return arr;  
 }  
 console.log(unique8([1, 1, 2, 3, 5, 3, 1, 5, 6, 7, 4]));  
 //9: 利用数组原型对象上的 lastIndexOf 方法。  
  
 function unique9(arr) {  
 var res = [];  
 for (var i = 0; i < arr.length; i++) {  
 res.lastIndexOf(arr[i]) !== -1 ? '' : res.push(arr[i]);  
 }  
 return res;  
 }  
 console.log(unique9([1, 1, 2, 3, 5, 3, 1, 5, 6, 7, 4]));  
 // 结果是[1, 2, 3, 5, 6, 7, 4]  
 //10: 利用 ES6的set 方法。  
  
 function unique10(arr) {  
 //Set数据结构，它类似于数组，其成员的值都是唯一的  
 return Array.from(new Set(arr)); // 利用Array.from将Set结构转换成数组  
 }  
 console.log(unique10([1, 1, 2, 3, 5, 3, 1, 5, 6, 7, 4]));

```

### 14. 定义一个长度为10 的数组并赋值为0-9

```js // 最简单最笨的方法 但是这样写的话 就显得的初级水平 如果定义长度为100 赋值为0-99就会很麻烦了 考点其实是数组的方法 var arr = [0,1,2,3,4,5,6,7,8,9]

// 一、使用Array.apply  
let arr= Array.apply(null, { length: 10 }).map((item,index)=>{  
 return index;  
 });  
console.log(arr);  
//(10) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
  
//原理：Array.apply的第二个参数是类数组调用Array.apply(null, { length: 10 })等于生成了长度为10内容都为undefinded的数组  
  
// 二、使用Array.from  
let arr= Array.from({length:10}).map((item,index)=>{  
 return index;  
 });  
console.log(arr);  
//(10) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
  
// 三、使用Array.prototype.fill  
let arr= new Array(10).fill(1).map((item,index)=>{  
 return index;  
 });  
console.log(arr);  
//(10) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]  
  
//说明一下：new Array(10).fill(1)生成一个长度为10的空数组并用1填充

```

### 15. 封装一个JS函数满足以下要求

```js var num1 = 1; var num2 = 2; var num3 = 3; var num4 = 4; // computed 函数 计算函数 但是本身不计算 返回一个计算函数 再调用的时候在计算 把当前计算函数的参数和调用计算返回值的参数累加起来计算 function computed(num1, num2) { return function (num) {  
return num+num1+num2; } } // add 函数+函数 返回+当前数字 function add(num) { return +num; } // mutiply函数 -函数 返回-当前数字 function mutiply(num) { return -num; } console.log(computed(mutiply(num2), add(num3))(num1)); // 结果等于 num1-num2+num3 == 2

var num1 = 1;  
 var num2 = 2;  
 var num3 = 3;  
 var num4 = 4;  
  
 function computed(num1, num2,num3) {  
 return function (num) {   
   
 return num+num1+num2+num3;  
 }  
 }  
  
 function add(num) {  
 return +num;  
 }  
  
 function mutiply(num) {  
 return -num;  
 }  
 // console.log(computed(mutiply(num2), add(num3))(num1)); // num1-num2+num3 == 2  
 console.log(computed(add(num2),mutiply(num3), add(num4))(num1)); // num1+num2-num3+num4 == 4  
  
  
 // 前面的方法如果有很多个数需要计算方法就要写很多次无法实现复用 下面是改造版本  
 var num1 = 1;  
 var num2 = 2;  
 var num3 = 3;  
 var num4 = 4;  
 // ...args表示把所有单个项参数装在一个数组args里面  
 function computed(...args) {  
 return function (num) {   
 console.log(args)  
 // 获取所有参数数组args 遍历 然后累加到num上   
 args.map((item) => {  
 num+=item  
 })  
 // 最后return num  
 return num  
 }  
 }  
  
 function add(num) {  
 return +num;  
 }  
  
 function mutiply(num) {  
 return -num;  
 }  
 console.log(computed(add(num2),mutiply(num3), add(num4))(num1)); // num1+num2-num3+num4 == 4

```

### 16. 使用观察者模式实现一个事件监听器 包含 监听事件on 删除监听off 触发监听emit 满足以下执行结果

class EventEmitter {  
 constructor() {  
 this.\_events = {}  
 }  
  
 on(event, callback) { //监听event事件，触发时调用callback函数  
 let callbacks = this.\_events[event] || []  
 callbacks.push(callback)  
 this.\_events[event] = callbacks  
 return this  
 }  
 off(event, callback) { //停止监听event事件  
 let callbacks = this.\_events[event]  
 this.\_events[event] = callbacks && callbacks.filter(fn => fn !== callback)  
 return this  
 }  
 emit(...args) { //触发事件，并把参数传给事件的处理函数  
 const event = args[0]  
 const params = [].slice.call(args, 1)  
 const callbacks = this.\_events[event]  
 callbacks.forEach(fn => fn.apply(this.params))  
 return this  
 }  
 once(event, callback) { //为事件注册单次监听器  
 let wrapFanc = (...args) => {  
 callback.apply(this.args)  
 this.off(event, wrapFanc)  
 }  
 this.on(event, wrapFanc)  
 return this  
 }  
  
}  
  
function query(val) {  
 console.log("query");  
}  
  
function query2(val) {  
 console.log("quer2");  
}  
  
  
var emitter = new EventEmitter();  
console.log(emitter)  
emitter.on("query", query);  
emitter.on("query", query2);  
emitter.emit("query"); // query query2  
emitter.off("query", query);  
emitter.emit("query"); // query2

### 17. js中的与解析

1. js 中的声明  
声明就是 变量的声明和函数的声明, 其目的是让 js 解释引擎知道有什么东西.  
声明时不参与运算的, 是不参与执行的, 是在预解析阶段就完成的.  
变量的声明  
  
// 变量的声明就是 var 变量名.  
var num = 123;  
// 这是一个语法糖,可以理解成  
var num;  
num = 123;  
函数的声明  
  
function 函数名 () { ... }  
// 在一个独立于任何语句( 表达式, if 结构, while 结构 等 )的独立结构中, 或函数中出现的代码, 为函数声明.  
2. js 预解析代码如何执行  
js 的代码执行要经理两个步骤, 首先是预解析. 预解析会通读所有代码. 如果发现错误则停下, 如果遇到声明则记录.  
在声明的时候, 如果是变量名的声明, 解析器内部就会记录下这个变量. 如果使用遍历则检查是否有记录.  
  
在声明的时候, 如果是函数声明, 则 解析器会先记录函数的名字( 相当于变量声明 ), 然后将函数的名字与函数体联系在一起.  
  
在预解析中, 如果出现重复声明, 则第一次声明起作用, 其后所有的同名的声明无效.  
  
// 例如:  
 var num = 1;  
 var num = 2;  
// 等价  
 var num = 1;  
 num = 2;  
声明结束后代码就会再从第一句话开始一句一句的执行.

### 18. js事件循环机制

在事件循环中，每进行一次循环操作称为 tick，每一次 tick 的任务处理模型是比较复杂的，但关键步骤如下：

在此次 tick 中选择最先进入队列的任务(异步队列)，如果有则执行(一次)

检查是否存在 Microtasks，如果存在则不停地执行，直至清空 Microtasks Queue

更新 render

主线程重复执行上述步骤在上诉tick的基础上需要了解几点：

JS分为同步任务和异步任务

同步任务都在主线程上执行，形成一个执行栈

主线程之外，事件触发线程管理着一个任务队列，只要异步任务有了运行结果，就在任务队列之中放置一个事件。

一旦执行栈中的所有同步任务执行完毕（此时JS引擎空闲），系统就会读取任务队列，将可运行的异步任务添加到可执行栈中，开始执行

### 19. 宏任务与微任务

1. 宏任务

macrotask，可以理解是每次执行栈执行的代码就是一个宏任务（包括每次从事件队列中获取一个事件回调并放到执行栈中执行）。

浏览器为了能够使得JS内部macrotask与DOM任务能够有序的执行，会在一个macrotask执行结束后，在下一个macrotask 执行开始前，对页面进行重新渲染，流程如下：

macrotask->渲染->macrotask->...

宏任务包括：script(整体代码)、 setTimeout、 setInterval I/O、 UI交互事件 、postMessage、 MessageChannel、 setImmediate(Node.js 环境)

### 20. 微任务

microtask,可以理解是在当前 task 执行结束后立即执行的任务。也就是说，在当前task任务后，下一个task之前，在渲染之前。

所以它的响应速度相比setTimeout（setTimeout是task）会更快，因为无需等渲染。也就是说，在某一个macrotask执行完后，就会将在它执行期间产生的所有microtask都执行完毕（在渲染前）。

微任务包括： Promise.then  、Object.observe、MutaionObserver、process.nextTick(Node.js 环境)

console.log('begin')  
  
        setTimeout(() => {    
  
//宏任务  
  
            console.log('setTimeout')  
  
        }, 0);  
  
  
  
        Promise.resolve().then(() => {  
  
//微任务  
  
            console.log('promise');  
  
        });  
  
        console.log('end');  
  
执行结果是 begin  ——>  end  ——> promise ——> setTimeout

let con = "<h2>页面渲染</h2>";  
  
        document.write(con);  
  
        console.log(1);  
  
        Promise.resolve().then(()=>{  
  
            console.log('promise  2 ');  
  
            alert('promise then')  
  
        })  
  
        setTimeout(()=>{  
  
            console.log('setTimeout 3');  
  
            alert('setTimeout')  
  
        },0)  
  
        console.log(4)  
  
执行顺序：  1  ——>  4  ——> promise 2 ——> 页面渲染 ——>  setTimeout 3  
  
由此我们可知 微任务 > DOM渲染 > 宏任务