1.冒泡排序（未优化前时间复杂度n\*n，优化后n）

function bubbleSort(arr){

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

let flag = true

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

if(arr[j] > arr[j+1]) {

flag = false

let temp = arr[j]

arr[j] = arr[j+1]

arr[j+1] = temp

}

}

// 这个flag的含义是：如果`某次循环`中没有交换过元素，那么意味着排序已经完成

if(flag)break;

}

return arr

}

2.快速排序

未优化版1：

function quickSort(arr) {

if(arr.length <= 1) return arr //递归终止条件

const pivot = arr.length / 2 | 0 //基准点，取整

const pivotValue = arr.splice(pivot, 1)[0]

const leftArr = []

const rightArr = []

arr.forEach(val => {

val > pivotValue ? rightArr.push(val) : leftArr.push(val)

})

return [ ...quickSort(leftArr), pivotValue, ...quickSort(rightArr)]

}

未优化版2：

function qSort(arr){

if(arr.length<=1) return arr;

const pivot=arr[0];

const arrLeft=[];

const arrRight=[];

for(let i=1;i<arr.length;i++){

arr[i]<=pivot?arrLeft.push(arr[i]):arrRight.push(arr[i]);

}

return qSort(arrLeft).concat(pivot,qSort(arrRight));

}

优化版（提升空间复杂度）

function quickSort(arr, left, right) { //这个left和right代表分区后“新数组”的区间下标，因为这里没有新开数组，所以需要left/right来确认新数组的位置

if (left < right) {

let pos = left - 1 //pos即“被置换的位置”，第一趟为-1

for(let i = left; i <= right; i++) { //循环遍历数组，置换元素

let pivot = arr[right] //选取数组最后一位作为基准数，

if(arr[i] <= pivot) { //若小于等于基准数，pos++，并置换元素, 这里使用小于等于而不是小于, 其实是为了避免因为重复数据而进入死循环

pos++

let temp = arr[pos]

arr[pos] = arr[i]

arr[i] = temp

}

}

//一趟排序完成后，pos位置即基准数的位置，以pos的位置分割数组

quickSort(arr, left, pos - 1)

quickSort(arr, pos + 1, right)

}

return arr //数组只包含1或0个元素时(即left>=right)，递归终止

}

//使用

var arr = [5,1,4,2,3]

var start = 0;

var end = arr.length - 1;

quickSort(arr, start, end)

3.防抖函数

function debounce(func, wait) {

let timer = null

return function() {

let context = this

let args = arguments

if (timer) clearTimeout(timer)

timer = setTimeout(() => {

func.apply(context, args)

}, wait)

}

}

function handle(){

console.log("handle22222");

}

window.addEventListener("scroll",debounce(handle,3000));//resize

4.节流1（定时器）

function throttle(func, wait) {

let timer = null

return function() {

let context = this

let args = arguments

if (!timer) {

timer = setTimeout(() => {

timer = null

func.apply(context, args)

}, wait)

}

}

}

节流2（时间戳）

function throttle1(func,wait){

let prev=0;

return function(){

let context = this

let args = arguments

let now=Date.now()

if(now-prev>wait){

func.apply(context,args);

pre=now;

}

}

}

5.判断是否是质数

function isPrime(num){

if(!isNaN(num)&&num>1)

{ if(num<=3) return true;

if(num%2==0) return false;

for(var i=3;i<=math.sqrt(num);i+=2)

{

if(num%i==0)

return false;

}

return true;

}

}else

console.log("请输入大于1的数字")

}

isPrime(17);

6.手写new

1.创建一个空对象，

2.将它的引用赋给this，继承构造函数的原型（继承构造函数原型的属性）。

3.通过this将属性和方法添加至这个对象。（继承构造器的构造属性）

4.最后返回this指向的新对象

//有参数情况

function myNew(func,…args){

let obj=Object.create(func.prototype);

let result=func.apply(obj,args);

return Object.prototype.toString.call(result)===’[object Object]’?result:obj;

}

//无参数情况

function myNew(){

let fn=[].shift.call(arguments);//arguments对象已经被删除了一项，后面应该用args=[].slice.call(arguments).slice(0)，或者顺序颠倒即可

let args=[].slice.call(arguments).slice()//等同于Array.prototype.slice.call(arguments)，把类数组对象转化为数组,除了第一个参数其余保留

let obj=Object.create(fn.prototype);

let result=fn.apply(obj,args);

return Object.prototype.toString.call(result)===’[object Object]’?result:obj;

}

7.手写call，apply，bind（return ()=>{}）

Function.prototype.call(thisArg,…args){ //Function.prototype.apply(thisArg,args){

let fn=Symbol(‘fn’);//声明一个独一无二的属性，防止覆盖已有属性

let thisArg=thisArg||window;

thisArg=this;//this指向调用call的函数

const result= thisArg[fn](…args);

delete thisArg[fn];//删除声明的fn属性

return result;

}

Function.prototype.myBind(thisArg,…args){

let fn=Symbol(‘fn’);

let thisArg=thisArg||window;

let \_this=this;

return function(){

let args=args.concat(Array.prototype.slice.call(arguments));

\_this[fn].call(thisArgs,…args);

}

}

8.数组扁平化

a.普通循环递归

function myFlatter(arr){

let res=[];

for(let i=0;i<arr.length;i++){ // **for** (**const** item **of** arr) {

Object.prototype.toString.call(arr[i])==='[object Array]' ? res=res.concat(myFlatter (arr[i])): res.push(arr[i]);

}

return res;

}

console.log(myFlatter([1,[2,3],4]));

b.序列化+正则

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

const str = `[${JSON.stringify(arr).replace(/(\[|\])/g, '')}]`

JSON.parse(str) // [1, 1, 2, 1, 2, 3]

c.reduce实现

function myFlatter(arr){

return arr.reduce((prev,item)=>{

return prev.concat(Array.isArray(item)?myFlatter(item):item);

},[])

}

d. ES6的flat

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

arr.flat(Infinity) // [1, 1, 2, 1, 2, 3]

e：迭代+展开运算符

// 每次while都会合并一层的元素，这里第一次合并结果为[1, 1, 2, 1, 2, 3, [4,4,4]]

// 然后arr.some判定数组中是否存在数组，因为存在[4,4,4]，继续进入第二次循环进行合并

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

while (arr.some(Array.isArray)) {

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

}

console.log(arr) // [1, 1, 2, 1, 2, 3, 4, 4, 4]

Arr=[[1,2,3],[5,6,7]]

newArr=Array.prototype.concat.apply([],arr)//[1,2,3,5,6,7]

9.数组去重

a:ES6 set

[…new Set(arr)]//Array.from(new set(arr))

b:indexOf

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

var res=[];

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

res.indexOf(item) > -1? ‘’:res.push(ietm);

return res;

}

//res=arr.slice(0,1);

c: 数组最后一项元素替换掉当前项元素，并删除最后一项元素

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

let remainArgs=arr.slice(i+1);

if(remainArgs.includes(arr[i])){

arr[i]=arr[arr.length-1];

arr.length--;

i--;

}

}

d:利用对象属性进行判断//效率最高

let obj={};

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

if(obj[arr[i]]){

arr[i]=arr[arr.length-1];//obj[arr[i]]=1;

arr.length--;

i--;

}

obj[arr[i]]=arr[i];

}

obj=null;//垃圾回收

console.log(arr);

e:利用Map

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

let map=new Map();

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

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

arr[i]=arr[arr.length-1];

arr.length--;

i--;

}else

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

}

console.log(arr);

10.ES5实现let

(function(){

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

console.log(i) // 0 1 2 3 4

}

})();

console.log(i) // Uncaught ReferenceError: i is not defined

11.ES5实现const原理

function \_const(key, value) {

const desc = {

value,

writable: false

}

Object.defineProperty(window, key, desc)

}

\_const('obj', {a: 1}) //定义obj

obj.b = 2 //可以正常给obj的属性赋值

obj = {} //无法赋值新对象

12.寄生式组合式继承// [https://juejin.cn/post/6844904116552990727#heading-13](https://juejin.cn/post/6844904116552990727" \l "heading-13)

function Parent(name) {

this.name = [name]

}

Parent.prototype.getName = function() {

return this.name

}

function Child() {

// 构造函数继承

Parent.call(this, 'zhangsan') //构造函数继承缺点：继承不到父类原型上的属性和方法

}

//原型链继承

// Child.prototype = new Parent() // 原型链继承缺点：某个Child实例的父类引用类型变量修改会影响所有的Child实例，无法传参

Child.prototype = Object.create(Parent.prototype) //组合方式继承缺点：为了解决构造函数被执行两次（Parent.call，new Parent()）的问题，将`指向父类实例`改为`指向父类原型`

Child.prototype.constructor = Child

//测试

const child = new Child()

const parent = new Parent()

child.getName() // ['zhangsan']

parent.getName() // 报错, 找不到getName()

13.对象深浅克隆

function deepClone(obj){

let map=new weakMap();

if(map.get(obj)){//判断对象是否被拷贝过，

return obj;//解决循环引用问题

}

let constructor=obj.constructor;//获取当前对象的构造函数，获得类型

if(/^(Date|RegExp)$/i.test(constructor.name)){//检测obj是否是正则或日期对象

return new constructor(obj);//创建一个obj类型的实例

}

if(obj!=='null'&& typeof obj==='object'||typeof obj==='function'){

map.set(obj,true);//为循环引用做标记

let cloneObj=Array.isArray(obj)?[]:{};

for(let prop in obj){

if(obj.hasOwnPrototype(prop)){

cloneObj[prop]=deepClone(obj[prop]);

}

}

return cloneObj;

}

else return obj;//null或基础数据类型

}

14.基于Promise封装ajax

ajax(method,url,data=””,dataType=”json”){

return new Promise((resolve,reject)=>{

//1.获取xhr

const xhr=new XMLHttpRequest();//需要做兼容性处理

const xhr=(function(){

if(window.XMLHttpReqquest)

return new XMLHttpRequest();

else return new ActiveXObject(“Microsoft.XMLHttp”);

})()

//2.创建请求

xhr.open(url,method,true);

//4.设置请求消息头

if(method==”post”){

xhr.setRequestHeader(“Content-Type”,”application/x-www-form-urlencoded”);

}

//3.设置回调

xhr.onreadystatechange=function(){

if(xhr. readystate===4){

if(xhr.status===200){

if(dataType.toLowerCase===”json”)

resolve(JASON.parse(xhr.responseText));

else

resolve(xhr.responseText);

}

}else

reject(“请求数据失败”+xhr.status);

}

//5.发送请求

xhr.send(data);

})

}

15.this的使用（对象不是作用域）

var bar = {

myName: "sjtu",

printName: function () {

console.log(myName)

}

}

function foo() {

let myName = "nic";

return bar.printName;

}

let myName = "sjtu\_nic";

let \_printName = foo();

\_printName(); // sjtu\_nic

bar.printName();//sjtu\_nic this指的是bar

var bar = {

myName: "sjtu",

printName: function () {

console.log(this.myName)

}

}

function foo() {

let myName = "nic";

return bar.printName;

}

let myName = "sjtu\_nic";

let \_printName = foo();

\_printName(); //undefined

bar.printName();//sjtu

function foo() {

// 返回一个箭头函数

return (a) => {

// this继承自foo();

console.log(this.a);

};

}

var obj1 = {

a: 2

};

var obj2 = {

a: 3

};

var bar = foo.call(obj1);

bar.call(obj2);// 2, 不是3 foo()内部创建的箭头函数会捕获调用时foo()的this

16. 实现发布订阅

发布订阅的核心:： 每次event. emit（发布），就会触发一次event. on（注册）

class EventEmitter(){

constructor(){

this.events={};//事件对象，存放订阅名字和事件

}

//订阅事件的方法

on(eventName,callback){

if(!this.events[eventName]){

this.events[eventName]=[callback]//一个事件名可以有多个事件

}else

this.events[eventName].push(callback);

}

//触发事件的方法

emit(eventName){

//根据触发的事件名称，遍历执行所有事件

this.events[eventName]&&this.events[eventName].forEach((cb)=>cb())

}

}

17，实现观察者模式

被观察者发生变化，主动通知所有观察者（基于发布订阅模式，收集观察者，状态变动后，通知所有观察者）

//创建被观察者对象

class Observable(){

constructor(state){

this.state= state;//被观察者的初始状态

this.observers=[];//存储所有观察者

}

//收集所有观察者

attach(0){

this.observers.push(0);

}

//被观察者状态发生改变

setState(newState){

this.state=newState;//更新状态

this.observers.forEach((o)=>o.update(this));//通知所有观察者修改状态，this指被观察者对象

}

}

//创建观察者对象

class Observer(){

constructor(name){

this.name=name;

}

update(obj){

console.log(“当前观察者”+this.name+”被通知了”+“被观察的obj对象的state发生改变”+obj.state)

}

}

18.实现函数柯里化

函数柯里化：给函数分步传入参数，每次传入部分参数，并返回一个更具体的函数接受剩下的参数，这中间可嵌套多层接收部分参数的函数，直至返回最后结果

function currying(fn,...args){

//fn.length函数参数的总和

//args.length currying 函数后面的参数总和

if(fn.length==args.length){

return fn(...args);

}else

{

return function (...newArgs){//继续分步传递新的参数

let allargs=[...args,...newArgs];//将旧传递的参数和新传递的参数合并在一起

return currying(fn,...allargs);

}

}

}

function add(a,b,c,d){

return a+b+c+d;

}

let fn1=currying(add,1,2);//function

let fn2=fn1(3);//function

let fn3=fn2(4);//10

19:正则1

/\*\*

\*查找字符串中关键词高亮，及出现的次数统计

//若统计字符串中所有字符的出现次数，应注意，先把字符串转化为数组，再去重，获取循环长度，再循环遍历原始字符串

\*/

(function searchStr(){

var context = "北京故宫是中国明清两代的皇家宫殿，旧称为紫禁城，位于北京中轴线的中心，是中国古代宫廷建筑之精华。" +

"北京故宫以三大殿为中心，占地面积72万平方米，建筑面积约15万平方米，有大小宫殿七十多座，房屋九千余间。" +

"是世界上现存规模最大、保存最为完整的木质结构古建筑之一。";

var words = ["北京","中国","世界"];

var arr=[];

for(const word of words){

var reg=new RegExp(word,"g");

var obj={};

if(reg.test(context)){

context=context.replace(reg,"<span style='color:red'>"+word+"</span>");

//console.log(context.match(reg).length);//出现的次数

var count=context.match(reg).length;

}

obj[word]=count;

arr.push(obj);

obj=null;

}

console.log(arr);

})()

20.图片懒加载

1.加载loading图片

2.判断哪些图片要加载【重点】

3.隐形加载图片

4.替换真图片

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width,initial-scale=1.0">

<title>图片可视化区域加载--懒加载</title>

</head>

<body>

<div class="imgList">

<img src="png\_2.png" data-src="1.jpg" /></br>

<img src="png\_2.png" data-src="2.jpg" /></br>

<img src="png\_2.png" data-src="3.jpg" /></br>

<img src="png\_2.png" data-src="4.jpg" /> </br>

<img src="png\_2.png" data-src="5.jpg" /></br>

<img src="png\_2.png" data-src="6.jpg" /></br>

<img src="png\_2.png" data-src="7.jpg" /></br>

</div>

<script>

//获取元素到浏览器顶部的高度

function getTop(e){

return e.offsetTop;

}

//懒加载实现

function lazyLoad(){

var imgs=document.querySelectorAll("img");

//可视区域高度

var h=window.innerHeight;//document.documentElement.clientHeight一样

//滚动区域高度

var s=document.documentElement.scrollTop||document.body.scrollTop;

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

//当图片距离浏览器顶部的距离>滚动高度+可视区域高度，图片懒加载

if((h+s)>getTop(imgs[i])){

if(imgs[i].getAttribute("src")!=="png\_2.png")//imgs[i].src获得的是图片完整url

continue;

(function(i){

var temp=new Image();

temp.src=imgs[i].getAttribute("data-src");

//temp在内存中加载完毕，直接通过src加载到dom上

temp.onload=function(){

//temp.onload为异步事件，使用匿名函数自调传入imgs[i]，否则i=imgs.length

imgs[i].src=imgs[i].getAttribute("data-src");//用真图替换假图

}

})(i);

}

}

}

//滚屏加载第一种写法

window.onscroll=window.onload=lazyLoad;

//第二种写法

//lazyLoad();---window.onload加载一次

//window.onscroll=function(){lazyLoad()};

//第三种加节流处理

//window.onscroll=throttle(lazyLoad,200);

</script>

</body>

</html>

21.promise实现图片加载数量限制

var urls = [

"https://hexo-blog-1256114407.cos.ap-shenzhen-fsi.myqcloud.com/AboutMe-painting1.png",

"https://hexo-blog-1256114407.cos.ap-shenzhen-fsi.myqcloud.com/AboutMe-painting2.png",

"https://hexo-blog-1256114407.cos.ap-shenzhen-fsi.myqcloud.com/AboutMe-painting3.png",

"https://hexo-blog-1256114407.cos.ap-shenzhen-fsi.myqcloud.com/AboutMe-painting4.png",

"https://hexo-blog-1256114407.cos.ap-shenzhen-fsi.myqcloud.com/AboutMe-painting5.png",

"https://hexo-blog-1256114407.cos.ap-shenzhen-fsi.myqcloud.com/bpmn6.png"

];

function loadImg(url) {

return new Promise((resolve, reject) => {

const img = new Image();

img.onload = function() {

console.log("一张图片加载完成");

resolve(img);

};

img.onerror = function() {

reject(new Error('Could not load image at' + url));

};

img.src = url;

});

}

function limitLoad(urls, handler, limit) {

let sequence = [].concat(urls); // 复制urls

// 这一步是为了初始化 promises 这个"容器"

let promises = sequence.splice(0, limit).map((url, index) => {

return handler(url).then(() => {

// 返回下标是为了知道数组中是哪一项最先完成

console.log(index);

return index;

});

});

// 注意这里要将整个变量过程返回，这样得到的就是一个Promise，可以在外面链式调用

return sequence

.reduce((pCollect, url) => {

return pCollect

.then(() => {

return Promise.race(promises); // 返回已经完成的下标

})

.then(fastestIndex => { // 获取到已经完成的下标

// 将"容器"内已经完成的那一项替换

promises[fastestIndex] = handler(url).then(

() => {

console.log("这是2"+promises);

console.log("这是zuikuai"+promises[fastestIndex]);

return fastestIndex; // 要继续将这个下标返回，以便下一次变量

}

);

})

.catch(err => {

console.error(err);

});

}, Promise.resolve()) // 初始化传入

.then(() => { // 最后三个用.all来调用

return Promise.all(promises);

});

}

limitLoad(urls, loadImg, 3)

.then(res => {

console.log("图片全部加载完毕");

console.log(res);

})

.catch(err => {

console.error(err);

});

22.红绿灯问题

function red() {

console.log('red');

}

function green() {

console.log('green');

}

function yellow() {

console.log('yellow');

}

function light(cb, timer) {

return new Promise(resolve => {

//cb();也可以

setTimeout(() => {

cb();

resolve()

}, timer);

})

}

//eg1

function step() {

Promise.resolve().then(() => {

return light(red, 3000)

}).then(() => {

return light(green, 2000)

}).then(() => {

return light(yellow, 1000)

}).finally(() => {

return step()//递归循环调用

})

}

//eg2

async function step(){

await lamp(green,3000);

await lamp(red,2000);

await lamp(yellow,1000);

}

step();

23.间隔1s打印出1,2,3

方法一：function test1(){

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

(function(i){

setTimeout(function(){

console.log("test1:"+i);

},1000);

})(i);

}

}

方法二：

function sleep(wait){

return new Promise(resolve=>{

setTimeout(()=>{

resolve();

},wait)

})

}

async function test(){

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

console.log(i);

await sleep(1000);

}

}

test();

方法三：

let arr = [1, 2, 3];

arr.reduce((pre, cur) => {

return pre.then(() => { //p.then()参数如果不是函数，会发生透传，当成同步代码执行

return new Promise(r => {

setTimeout(() => {

r(console.log(cur))

}, 1000);

})

})

}, Promise.resolve())

24.封装 console.log 的打印的功能，封装clg函数; 在上面封装的clg函数基础上， 添加功能： 每次调用clg2函数，打印的内容前面固定加上指定的字符 “XX友情提示：”

function clg(str){

return console.log("%s"," XX友情提示："+str)

}

sm.replace(/\b[a-z]/g,function(kw){return kw.toUpperCase()});//每个单词首字母大写

25.函数式编程

事实上柯里化是一种“预加载”函数的方法，通过传递较少的参数，得到一个已经记住了这些参数的新函数，某种意义上讲，这是一种对参数的“缓存”，是一种非常高效的编写函数的方法。

传递给函数一部分参数来调用它，让它返回一个函数去处理剩下的参数。

function addCurrying(a){

return function(b){

return function(c){

return a+b+c;

}

}

}

手写柯里化

function add(a,b,c,d){ return a+b+c+d }

function currying(fn,...args){

//args的长度与fn参数个数一样

if(fn.length===args.length){

return fn(...args)

}

else{

//args的长度小于fn参数个数

return function(...newArgs){

//已经传过的参数args+未传入参数

let allArgs=[...args,...newArgs];

return currying(fn,...allArgs)

}

}

}

console.log(add.length);//4

let fn1=currying(add,1,2);//function

let fn2=fn1(3);function

let fn3=fn2(4);10

26. promise

Promise.resolve('hello');

// 相当于

const promise = new Promise(resolve => {

resolve('hello');

});

function maicai(resolve,reject){

setTimeout(function(){resolve("西红柿")},3000)

}

function cook(resolve,reject){

console.log("zuofankaishi...")

setTimeout(function(){resolve("面条"),2000})

}

new Promise(maicai).then(function(data){console.log(data),return new Promise(cook)})//3s后输出西红柿

new Promise(maicai).then(data=>{new Promise(cook)})

(async ()=>{

let 蔬菜 = await 买菜();

let 饭菜 = await 做饭(蔬菜);

let 送饭结果 = await 送饭(饭菜);

let 通知结果 = await 通知我(送饭结果);

})();

eg2:

const fn = new Promise(function (resolve, reject) {

setTimeout(()=>{

let num = Math.ceil(Math.random() \* 10) // 假设num为7

if (num > 5) {

resolve(num) //返回7

} else {

reject(num)

}

},2000)

})

--------方法一

fn.then((res)=>{

console.log(res) // 7

},(err)=>{

console.log(err)

})

--------方法2

fn.then((res)=>{

console.log(res) // 7

}).catch((err)=>{

console.log(err)

})

\*\*\*\*\*\*\*\*\*\*\*

fn = new Promise(function (resolve, reject) {

let num = Math.ceil(Math.random() \* 10)

if (num > 5) {

resolve(num)

} else {

reject(num)

}

})

// 第一次回调

fn.then((res)=>{

console.log(`res==>${res}`)

return new Promise((resolve,reject)=>{

if(2\*res>15){

resolve(2\*res)

}else{

reject(2\*res)

}

})

}，(err)=>{

console.log(`err==>${err}`)

}).then((res)=>{ // 第二次回调

console.log(res)

}，(err)=>{

console.log(`err==>${err}`)

})

// resolve(成功) onFulfilled会被调用

const promise = new Promise((resolve, reject) => {

resolve('fulfilled'); // 状态由 pending => fulfilled

});

promise.then(result => { // onFulfilled

console.log(result); // 'fulfilled'

}, reason => { // onRejected 不会被调用

})

// reject(失败) onRejected会被调用

const promise = new Promise((resolve, reject) => {

reject('rejected'); // 状态由 pending => rejected

});

promise.then(result => { // onFulfilled 不会被调用

}, reason => { // onRejected

console.log(rejected); // 'rejected'

})

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

console.log(1);

setTimeout(function(){

console.log(2);

new Promise(function(resolve){

console.log(3);

resolve();

}).then(function(){

console.log(4);

})

},0);

new Promise(function(resolve){

console.log(5);

resolve();

}).then(function(){console.log(6)});

setTimeout(function(){

console.log(7);

new Promise(function(resolve){

console.log(8);

resolve();

}).then(function(){

console.log(9);

})

},3);

//1,5,6,2,3,4,7,8,9

setTimeout内部回调函数执行顺序在浏览器环境与node环境是有差异的。11以下的node运行结果是1,5,6,2,3,7,8,4,9

26.千分符(截取实现，正则实现)

let num1 = 1234567894532;

let num2 = 123435.4542;

function numFormatter(num){

num=num.toString().split('.');

let arr=num[0].split('').reverse();

let res=[];

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

if(i%3==0&&i!==0){

res.push(',');

}

res.push(item);

})

res.reverse();

return num[1]? (res.concat(`.${num[1]}`)).join(''):res.join('');

}

console.log(numFormatter(num2));// 123,435.4542

console.log(numFormatter(num1));// 1,234,567,894,532

function numFormatter(num) {

return num.toString().replace(/\d{1,3}(?=(\d{3})+$)/g, val => val + ',')

}

27.遍历器，for of可遍历拥有Iterator接口的数据结构

function makeIterator(arr){

let nextIndex=0;

let len=arr.length;

return {

next:function(){

return ((nextIndex)<len)?

{

value:arr[nextIndex++],

done:false

}:

{ value:undefined,

done:true

}

}

}

}

const y=makeIterator(['a','b']);

y.next();//{value: "a", done: false}

y.next();//{value: "b", done: false}

y.next();//{value: undefined, done: true }

28.无限循环累加函数

sum 返回一个函数，收集所有的累加项，使用递归实现

返回函数带有 valueOf 属性，用于统一计算

function sum(...args){

const fn=function(...rest){

return sum(...[...args,...rest])

}

fn.valueOf=()=>{

return args.reduce((p,item)=>p+item,0);

}

return fn;

}

sum(1).valueOf()//1

sum(1)(2).valueOf()//3

sum(1,2)(3).valueOf()//6

20201116--------

将多维数组转化为一维(数组扁平化)

方法1：reduce

let arr = [[0, 1], [2, 3], [4,[5,6,7]]]

const newArr = function(arr){

return arr.reduce((pre,cur)=>pre.concat(Array.isArray(cur)?newArr(cur):cur),[])

}

console.log(newArr(arr)); //[0, 1, 2, 3, 4, 5, 6, 7]

方法2：

newArr=arr.toString().split(“,”).map((item)=>Number(item));

方法3：

arr.join(",").split(",").map(function(item){return parseInt(item)})

数组去重(利用对象、set 、array.sort())

1.function distinct(a, b) {

return Array.from(new Set([...a, ...b]))

}

2.function distinct(a, b) {

let arr = a.concat(b)

arr = arr.sort()

let result = [arr[0]]

for (let i=1, len=arr.length; i<len; i++) {

arr[i] !== arr[i-1] && result.push(arr[i])

}

return result

}

3.function distinct(a, b) {

let arr = a.concat(b)

let result = []

let obj = {}

for (let i of arr) {

if (!obj[i]) {

result.push(i)

obj[i] = 1;

}

}

return result

}

数组乱序

**先从数组末尾开始，选取最后一个元素，与数组中随机一个位置的元素交换位置**

**然后在已经排好的最后一个元素以外的位置中，随机产生一个位置，让该位置元素与倒数第二个元素进行交换**

function shuffle(arr) {

let len = arr.length;

i

while (len) {

let i = (Math.random() \* len--) >> 0; // 交换位置，>>0带符号右移0位转化为整数

let temp = arr[len];

arr[len] = arr[i];

arr[i] = temp;

}

return arr;

}

 random 函数因为循环周期的存在，无法生成真正的随机数

function randomSort(a,b) {

return .5 - Math.random();

}

let arr = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9'];

arr.sort(randomSort);

js实现双向数据绑定

<body>

<input type="text" id="user" value="">

<span id="show1" style="background:red"></span>

<script>

var obj = {};

var val = "123";

Object.defineProperty(obj, 'val', {

get: function() {

return val;

},

set: function(newValue) {

val = newValue;

document.getElementById("user").value = newValue;

document.getElementById("show1").innerHTML = newValue;

console.log(document.getElementById("show1")); //span先输出

}

})

window.addEventListener("keyup", function(e) {

obj.val = e.target.value;

console.log(obj.val); //后输出

})

设计模式：代理模式

//代理模式

var idol = {

name: "caixukun",

phone: 10086,

price: 1000

}

var agent = new Proxy(idol, {

get: function(target, key) { //target指向第一个参数idol

return key + "联系方式:" + target.phone;

},

set: function(target, key, value) {

if (key === 'price') {

if (target.price < 1000) throw new Error("报价过低");

target.price = value;

}

}

})

console.log(agent.name);

console.log(agent.phone);

agent.price = 100;

agent.price = 4000;

</script>

</body>

对象增加遍历器实现for of对象

var obj={

name:"zhangsan",

age:18,

[Symbol.iterator]: function() {

var self=this;

var ks=Object.keys(self);

var i=0;

return {

next: function() {

//方法一：

return {

value:ks[i++],

done:(i>ks.length)

}

//方法二：

/\* if(ks[i]){

return {

value:ks[i++],

done:false

}

}

else

{

return {

value:undefined,

done:true

}

}\*/

}

}

}

}

for(var b of obj){

console.log(b);

}

//name,age

promise，async改写回调地狱：

const p1=function (value,wait){

return new Promise((resolve)=>{

setTimeout(()=>{

console.log(value);

resolve();

},wait)

})

}

//promise方法：

/\*p1(111,1000).then(()=>{

p1(222,2000);

}).then(()=>{

p1(333,3000);

}).catch((err)=>{

console.log(err)

}\*/

//async 方法：

f1=async function(){

await p1(111,1000);

await p1(222,2000);

await p1(333,3000);

}

f1();

<script>

var template = "{{name}}是谁？，年龄{{age}}";

var content = { name: 'zhangsan', age: 18 };

function render(obj, str) {

return str.replace(/{{(.\*?)}}/g, (match, key) => obj[key] ? obj[key] : match)

}

console.log(render(content, template));

</script>

Object.is()方法和“==”运算符之间的区别//Object.is(NaN,NaN)//TRUE

1、“==”和“===”运算符将数字值“+0”和“-0”视为相等，而Object.is（）方法则将它们视为不等于。

1. 如果两个值都是数字并且都是“NaN”,“==”和“===”运算符也不会将其视为相等；而Object.is（）方法则视为相等。

字符串split，sort排序，join，match(/([a-z])\1\*)/ig) jion replace(//,(kw,$1)=>{})

Hash 对象处理效率高

window.a = 300;

function fn1() {

    this.a = 100;

    this.b = 200;

    return function () {

        alert(this.a);

    }.call(arguments[0]);//call函数会立即执行

}

var a = new fn1().b;//alert(300)---new fn1()会先执行，没有返回值，所以返回新创建的对象.b

console.log(a);//200