刷题

12/9 sat

1、数组的方法对比。

返回的是什么，有没有更改原数组。

比如push，unshift，splice等等。有空做一个summary。

Slice（from,to） 当参数为0时，返回的是原数组哦。

数组的map方法，forEach方法

字符串的方法对比。Substring（from,to） slice(from,to) 注意数组的splice(from, len),

字符串还有一个substr(from,len)，但是不常用。

2、定位：

以下代码，分别给节点 #box 增加如下样式，问节点 #box 距离 body 的上边距是多少？

<body style=”margin:0;padding:0”>

<div id=”box” style=”top:10px;margin:20px 10px;”>

</div>

</body>

如果设置 position: static ; 则上边距为 1 px

如果设置 position: relative ; 则上边距为 2 px

如果设置 position: absolute ; 则上边距为 3 px

如果设置 position: sticky ; 则滚动起来上边距为 4 px

position: static ; 则上边距为(  20 ) px  静态定位 top值无效

position: relative ; 则上边距为(  30 ) px 移动的时候会包括margin  
position: absolute ; 则上边距为(  30 ) px 移动的时候会包括margin  
position: fixed ; 则上边距为(  30 ) px  固定定位的margin也会生效  移动的时候也会包括margin  
position: sticky ; 则上边距为(  20 ) px，页面滚动起来为（10）px，margin会无效；页面没滚动的 时候是静态定位

3、正则表达式！

4、

在Blink和WebKit的浏览器中，某个元素具有3D或透视变换（perspective transform）的CSS属性，会让浏览器创建单独的图层。

我们平常会使用left和top属性来修改元素的位置，但left和top会触发重布局，取而代之的更好方法是使用translate，这个不会触发重布局。

移动端要想动画性能流畅，应该使用3D硬件加速，因此最好给页面中的元素尽量添加translate3d或者translateZ(0)来触发3D硬件加速。（错。滥用硬件加速会导致严重性能问题，因为它增加了内存使用，并且它会导致移动端电池寿命减少。）

解决浏览器渲染的性能问题时，首要目标就是要避免层的重绘和重排。

5、flex大法，float大法，绝对定位大法

6、Hybrid APP 翻译过来就是混合APP。概念：半原生班 web 的混合类 APP，需要下载安装，看上去类似原生 APP，但只有少数 UI Web view ，访问的内容是 web，例如新闻类 APP，视频类 APP普遍采用原生框架，web 的内容。混合APP 追求原生 APP 的体验，但仍受限于技术，网速等。

与原生APP 比优势：成本低、更新快

劣势：操作速度慢，影响用户体验。

12/10 Sunday

IndexdDB 是 HTML5 的本地存储，把一些数据存储到浏览器（客户端）中，当与网络断开时，可以从浏览器中读取数据，用来做一些离线应用。

Cookie 通过在客户端 ( 浏览器 ) 记录信息确定用户身份，最大为 4 kb 。

url 参数用的是 get 方法，从服务器上获取数据，大小不能大于 2 kb 。

Session 是服务器端使用的一种记录客户端状态的机制 。

post 是向服务器传送数据，数据量较大。

local Storage 也是 HTML5 的本地存储，将数据保存在客户端中（一般是永久的）。

填写内容让下面代码支持a.name = “name1”; b.name = “name2”;

function obj(name){

    1

}

obj. 2 = "name2";

var a = obj("name1");

var b = new obj;

你的答案 (错误)

1 this.name=name;

2 name

参考答案   
(1) if(name){ this.name = name;}return this;  
(2) prototype.name

输出对象中值大于2的key的数组

var data = {a: 1, b: 2, c: 3, d: 4};

Object.keys(data).filter(function(x) { return 1 ;})

期待输出：[“c”,”d”]

你的答案 (错误)

1 x>'b'

参考答案   
(1) data[x]>2

请实现一个fibonacci函数，要求其参数和返回值如下所示：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | /\*\*   \*@desc: fibonacci   \*@param: count {Number}   \*@return: result {Number} 第count个fibonacci值，计数从0开始    fibonacci数列为：[1, 1, 2, 3, 5, 8, 13, 21, 34 …]    则getNthFibonacci(0)返回值为1    则getNthFibonacci(4)返回值为5   \*/  function getNthFibonacci(count) {  } |

这道题核心是递归，看大家都能做，不过有些细节可能没考虑到，比如说参数的判断：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | function fib(count) {      //参数判断      var count = parseInt(count);      if (isNaN(count) || count < 0) {          return 0;      }        function f(count) {          if (count <= 1)              return 1;          return arguments.callee(count - 1) + arguments.callee(count - 2);    //callee是装逼用的，直接用f也行      }      return f(count);  } |

### 在匿名递归函数中使用 arguments.callee

递归函数必须能够引用它本身。很典型的，函数通过自己的名字调用自己。然而，匿名函数 (通过 [函数表达式](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Operators/function) 或者 [函数构造器](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Function)创建) 没有名称。因此如果没有可访问的变量指向该函数，唯一能引用它的方式就是通过 arguments.callee。

下面的例子定义了一个函数，按流程，定义并返回了一个阶乘函数。该例并不是很实用，并且几乎都能够用 [命名函数表达式](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Operators/function) 实现同样结果的例子, and there are nearly no cases where the same result cannot be achieved with .

function create() {

return function(n) {

if (n <= 1)

return 1;

return n \* arguments.callee(n - 1);

};

}

var result = create()(5); // returns 120 (5 \* 4 \* 3 \* 2 \* 1)

圣杯布局！

Float + relative 布局大法

Flex 布局大法！

**如何获取下面表单 select**

**<form name="a">  
<select name="a" size="1" id=”obj”>  
<option value="a">1</option>  
<option value="b">2</option>  
<option value="c">3</option>  
</select>  
</form> 域的选择部分的文本？()**

obj.options[obj.selectedIndex].text

ajax 里并没有statusCode;

ajax响应：

    responseText:作为响应主体被返回的文本

    responseXML:如果响应的内容类型是"text/xml"或者"application/xml", 这个属性中将保存包含着响应数据XML DOM文档

    status:响应的HTTP状态

    statusText:HTTP状态的说明

函数 commafy，它接受一个数字作为参数，返回一个字符串，可以把整数部分从右到左每三位数添加一个逗号，如：12000000.11 转化为 12,000,000.11。

我的答案：

const commafy = (num) => {

let integer = Math.floor(num);

let decimal = num.toString().indexOf('.') === -1?'':num.toString()

.slice(num.toString().indexOf('.')); //是string

let numbers = [];

while( Math.floor(integer / 1000) !== 0){

numbers.push(integer % 1000 > 99?integer % 1000 + '':integer%1000>9?'0' + integer % 1000:'00' + integer % 1000);

integer = Math.floor(integer / 1000);

}

numbers.push(integer + '');

numbers[0] += decimal;

return numbers.reverse().reduce((acc,val) => acc + val + ',', '').replace(/,$/g,'');

}

改进法：

const commafy = (num) => {

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

let integer = arr[0].split('').reverse().reduce((acc,val,index) => index%3 !== 0 || index === 0?acc+val:acc+','+val,'');

return (arr[1])?integer.split('').reverse().join('') + '.' + arr[1]:integer.split('').reverse().join('');

}

因为reduce本来可以带有index此属性~！

正则法：

**function** **commafy** (num) {

**let** arr=num.toString().split('.')

**var** integer=arr[0].replace(/(?!\b)(?=(\d{3})+$)/g,',')

**if**(arr[1]){

**return** integer+'.'+arr[1]

}**else** **return** integer

}

// 1.函数 commafy，它接受一个数字作为参数，返回一个字符串，可以把整数部分从右到左每三位数添加一个逗号，如：12000000.11 转化为 12,000,000.11。

// 我的答案：

//

const commafy = (num) => {

let integer = Math.floor(num);

let decimal = num.toString().indexOf('.') === -1?'':num.toString()

.slice(num.toString().indexOf('.')); //是string

let numbers = [];

while( Math.floor(integer / 1000) !== 0){

numbers.push(integer % 1000 > 99?integer % 1000 + '':integer%1000>9?'0' + integer % 1000:'00' + integer % 1000);

integer = Math.floor(integer / 1000);

}

numbers.push(integer + '');

numbers[0] += decimal;

return numbers.reverse().reduce((acc,val) => acc + val + ',', '').replace(/,$/g,'');

}

// 改进法：

const commafy = (num) => {

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

let integer = arr[0].split('').reverse().reduce((acc,val,index) => index%3 !== 0 || index === 0?acc+val:acc+','+val,'');

return (arr[1])?integer.split('').reverse().join('') + '.' + arr[1]:integer.split('').reverse().join('');

}

// 因为reduce本来可以带有index此属性~！

//

// 正则法：

function commafy (num) {

let arr=num.toString().split('.')

var integer=arr[0].replace(/(?!\b)(?=(\d{3})+$)/g,',')

if(arr[1]){

return integer+'.'+arr[1]

}else return integer

}

// 2.完成 extname 函数，它会接受一个文件名作为参数，你需要返回它的扩展名。

// 例如，输入 emoji.png，返回 .png

//

const extname = (filename) => {

let arr = filename.split('.');

return arr.length === 1 || arr[0] === ''?'':'.' + arr[arr.length-1];

}

// 3.完成正则表达式 TRIM\_REGX，可以用它来删除一个字符串前后多余的空白字符。

// 注意：你只需要完成正则表达式的编写

const TRIM\_REGX = /^\s+|\s+$/g

// 4.完成函数 hexToRGB，它的作用将 16 进制颜色值转换成 RGB 值：

// hexToRGB('#F0F0F0') // => rgb(240, 240, 240)

// hexToRGB('#9fc') // => rgb(153, 255, 204)

// hexToRGB('无效颜色') // => null

//注意很多情况，比如hex如果为空？

//比如#4444？

//比如#script？

const hexToRGB = (hex) => {

if (hex){

let arr = hex.split('#');

let threetwobitarr = [];

if (arr.length === 1 || arr[1].length > 6 || arr[1].search(/[^0-9a-fA-F]/) !== -1) {

return null;

} else {

let colorhex = arr[1].split('');

if (colorhex.length === 6) {

let twobitarr = colorhex.reduce((acc,val,index,colorhex) => acc.concat(index%2 ===1?

(colorhex[index-1] + val):''),[]);

threetwobitarr = twobitarr.filter( v => v!== '');

} else if (colorhex.length === 3) {

threetwobitarr = colorhex.map( v => v + v );

} else {

return null;

}

let tenarr = threetwobitarr.map( v=> parseInt(v,16));

return tenarr.reduce((a,v,i) => a.concat(i!==2?v + ', ':v + ')'),'rgb(');

}

} else return null;

}

//5.完成 getChildAttributes 函数，它接受一个 DOM 元素作为参数和一个属性名作为参数，你需要返回这个 DOM 的 直接 子元素的特定属性列表。例如：

// <ul id='list'>

// <li data-name="Jerry" class="item"><span>1</span></li>

// <li data-name="Lucy" class="item"><span>2</span></li>

// <li data-name="Tomy"><span>3</span></li>

// </ul>

// getChildAttributes(el, 'data-name') // => ['Jerry', 'Lucy', 'Tomy']

// getChildAttributes(el, 'class') // => ['item', 'item', null]

// 只需要完成 getChildAttributes 的编写。

const getChildAttributes = (el, attrStr) =>{

let children = el.children;

let res = [];

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

res.push(children[i].getAttribute(attrStr));

}

return res;

}

//6.现在有很多只猫，都很肥：

// const cats = [

// { name: 'Tom', weight: 300 },

// { name: 'Lucy', weight: 400 },

// { name: 'Lily', weight: 700 },

// { name: 'Jerry', weight: 600 },

// ...

// ]

// 现在你需要把它们按照由胖到瘦的顺序把它们渲染到 id 为 cats-list 的 div 元素当中：

// <div id="cat-list">

// <div class='cat'>

// <span class='cat-name'>Lily</span>

// <span class='cat-weight'>700</span>

// </div>

// <div class='cat'>

// <span class='cat-name'>Jerry</span>

// <span class='cat-weight'>600</span>

// </div>

// <div class='cat'>

// <span class='cat-name'>Lucy</span>

// <span class='cat-weight'>400</span>

// </div>

// <div class='cat'>

// <span class='cat-name'>Tom</span>

// <span class='cat-weight'>300</span>

// </div>

// ...

// </div>

// 完成 renderFatCats 函数，接受一个 cats 数组作为参数，然后它会往 div#cats-list 元素内渲染类似以上的结果。注意类名需要保持一致；另外renderFatCats 可能会被多次调用，注意清空上一次渲染的数据。

// 你可以使用 jQuery、React.js 等方式来完成。

// （你不需要调用 renderFatCats）。

//

function renderFatCats (cats) {

cats.sort( (m,n) => n.weight-m.weight);

let parent = document.getElementById("cats-list");

//init()，清空原来的列表

let kids = parent.childNodes;

for(var i = kids.length - 1; i>=0; i--){

parent.removeChild(kids[i]);

}

//添加新列表

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

let nameNode = document.createElement('span');

nameNode.setAttribute('class','cat-name');

nameNode.innerHTML = cats[i].name;

let weightNode = document.createElement('span');

weightNode.setAttribute('class','cat-weight');

weightNode.innerHTML = cats[i].weight;

let wrap = document.createElement('div');

wrap.setAttribute('class','cat');

wrap.appendChild(nameNode);

wrap.appendChild(weightNode);

parent.appendChild(wrap);

}

}

//6.完成一个生成计数器的函数 plusFor，调用它会返回一个计数器。计数器本身也是一个函数，每次调用会返回一个字符串。

// 达到以下的效果：

// const counter1 = plusFor('小明')

// counter1() // => 为小明+1s

// counter1() // => 为小明+2s

// counter1() // => 为小明+3s

// ...

// const counter2 = plusFor('李梅')

// counter2() // => 为李梅+1s

// counter2() // => 为李梅+2s

// counter2() // => 为李梅+3s

// ...

// 注意你只需要完成 plusFor 函数，不要使用额外的全局变量。

const plusFor = (str) => {

let i = 0;

return function(){

i++;

return '为' + str + '+' + i + 's';

}

}

//7.李雷向韩梅梅求婚，韩梅梅说过一段时间（20~50ms）再回复他。

// 完成 proposeToMissHan 函数，会传入一个布尔值参数 isOK，用来预先设定是否答应李雷的求婚。这个函数会返回一个 Promise，一段时间（20～50ms）以后，根据 isOK 参数，韩梅梅可能会说字符串 ok 答应李雷，也可能说字符串 no 来拒绝（reject）李雷。

// 你只需要完成 proposeToMissHan 函数的编写。

const proposeToMissHan = (isOK) => {

let p = new Promise((resolve,reject) => {

setTimeout(()=>{

if(isOK){

resolve("ok");

} else {

reject("no");

}

},22);

});

return p;

}

//8.数据的分页在我们的业务当中非常常见，例如 ScriptOJ 的问题列表就有分页。

// 完成分页函数 getPages，接收两个参数：

// getPages(total, itemsPerPage)

// total： 表示总共有多少条数据

// itemsPerPage：表示每页有多少条数据

// getPages(total, itemsPerPage) 会返回一个数字告诉我们需要有多少页数据。例如，总共 101 条数据，每页有 10 条，就需要 11 页，那么就返回 11。

// itemsPerPage 为 0 的时候返回 0。

// 你只需要完成 getPages 函数。

//

const getPages = (total, itemsPerPage) => !itemsPerPage?0:Math.ceil(total/itemsPerPage);

//9.在函数式编程当中有一个很重要的概念就是函数组合，实际上就是把处理数据的函数像管道一样连接起来，然后让数据穿过管道得到最终的结果。例如：

// const add1 = (x) => x + 1

// const mul3 = (x) => x \* 3

// const div2 = (x) => x / 2

// div2(mul3(add1(add1(0)))) // => 3

// 而这样的写法可读性明显太差了。我们可以构建一个 compose 函数，它接受任意多个函数作为参数（这些函数都只接受一个参数），然后 compose 返回的也是一个函数，达到以下的效果：

// const operate = compose(div2, mul3, add1, add1)

// operate(0) // => 相当于 div2(mul3(add1(add1(0))))

// operate(2) // => 相当于 div2(mul3(add1(add1(2))))

// 简而言之：compose 可以把类似于 f(g(h(x))) 这种写法简化成 compose(f, g, h)(x)。请你完成 compose 函数的编写。

// 额外挑战：你能通过 1~2 行代码实现 compose 吗。

//

const f1 = (x) => x + 1

const f2 = (x) => x \* 3

const f3 = (x) => console.log(x);

// const compose = (...args) => args[0] = () => args[1] =() => args[2];

//

//

// 一行代码实现法

// const compose = (...fns) => {

// return x => fns.reduceRight((v, f) => f(v), x);

// }

const compose = (...args) => {

return x => {

let re = args.pop()(x);

return args.length?compose(...args)(re):re;

}

}

// var a = compose(f3,f1,f2);

//

//

//

// 10.小科去了一家新的公司做前端主管，发现里面的前端代码有一部分是 C/C++ 程序员写的，他们喜欢用下划线命名，例如： is\_good。小科决定写个脚本来全部替换掉这些变量名。

// 完成 toCamelCaseVar 函数，它可以接受一个字符串作为参数，可以把类似于 is\_good 这样的变量名替换成 isGood。

// 变量名首尾的下划线不需要做处理，中间的下划线全部删除并且处理成驼峰。

//

const toCamelCaseVar = (variable) => variable.replace(/[^\_]\_+[a-zA-Z0-9]/g,

(val) => val.substring(0,1) + val.substring(val.length-1,val.length).toUpperCase());

//11.函数式编程当中有一个非常重要的概念就是 函数柯里化。一个接受 任意多个参数 的函数，如果执行的时候传入的参数不足，那么它会返回新的函数，新的函数会接受剩余的参数，直到所有参数都传入才执行操作。这种技术就叫柯里化。请你完成 curry 函数，它可以把任意的函数进行柯里化，效果如下：

/\*

const f = (a, b, c d) => { ... }

const curried = curry(f)

curried(a, b, c, d)

curried(a, b, c)(d)

curried(a)(b, c, d)

curried(a, b)(c, d)

curried(a)(b, c)(d)

curried(a)(b)(c, d)

curried(a, b)(c)(d)

// ...

// 这些函数执行结果都一样

// 经典加法例子

const add = curry((a, b) => a + b)

const add1 = add(1)

add1(1) // => 2

add1(2) // => 3

add1(3) // => 4

注意，传给 curry 的函数可能会有任意多个参数。

\*/

var curry = function curry(fn) {

var arr = arguments[1]

? arguments[1]

: []

return function f1() {

var args = [].slice.call(arguments)

return function f2(arg) {

return arg.length === fn.length

? fn.apply(null, arg)

: curry(fn, arg)

}(arr.concat(args))

}

}

//es6方法

const curry = ( f, arr = []) => (...args) => ( a => a.length === f.length ? f(...a) : curry(f, a))([...arr, ...args]);

//12

/\*

完成 getPageTags 函数，判断你的代码所执行的页面用到了哪些标签。

例如，如果页面中：

<html>

<head></head>

<body></body>

那么 getPageTags() 则返回数组 ['html', 'head' 'body', 'script']（顺序不重要）。

\*/

//xiao tips

//为什么这个操做可以把HTMLCollection转为Array?

//

//var eles=document.getElementsByTagName('div');

// var elesArray=Array.prototype.slice.call(eles,0);

//

// 回答这个问题你首先需要知道，NodeList是一个array like的object，不是真正的array.其次你需要知道Array.prototype.slice方法可以把一个array like的object转成新的array，方法是把slice方法bind给待转对象。bind的过程可以通过call方法进行。call第一个参数就是显式指定的this. 在此处就是这个NodeList.最后这个方法可以简化成

//

// [].slice.call(elems)

const getPageTags = () => {

let arr = [].slice.call(document.getElementsByTagName('\*')).reduce((acc,val) => acc.concat(val.tagName.toLowerCase()),[]);

return unique = [...new Set(arr)];

}

//13.从某数据库接口得到如下值：

/\*

{

rows: [

["Lisa", 16, "Female", "2000-12-01"],

["Bob", 22, "Male", "1996-01-21"]

],

metaData: [

{ name: "name", note: '' },

{ name: "age", note: '' },

{ name: "gender", note: '' },

{ name: "birthday", note: '' }

]

}

rows 是数据，metaData 是对数据的说明。现写一个函数 parseData，将上面的对象转化为期望的数组：

[

{ name: "Lisa", age: 16, gender: "Female", birthday: "2000-12-01" },

{ name: "Bob", age: 22, gender: "Male", birthday: "1996-01-21" },

]

\*/

var data =

{

rows: [

["Lisa", 16, "Female", "2000-12-01"],

["Bob", 22, "Male", "1996-01-21"]

],

metaData: [

{ name: "name", note: '' },

{ name: "age", note: '' },

{ name: "gender", note: '' },

{ name: "birthday", note: '' }

]

}

const parseData = (data) => {

let rows = data.rows;

let metaData = data.metaData;

let res = [];

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

let arr = {};

for(let j = 0; j < metaData.length; j++){

let key = metaData[j].name.replace(/"/g,'');

let value = rows[i][j];

arr[key] = value;

}

res.push(arr);

}

return res;

}

//14.deepflatten array

//

const flatten = (arr) => arr.reduce((acc,val) => acc.concat(Array.isArray(val)?flatten(val):val),[]);

//15.完成 cookieJar 单例，它有三个方法：

// set(name, value, days)：设置 cookie 的值，days 为多少天以后过期。

// get(name)：获取 cookie 的值。

// remove(name)：删除 cookie 的值。

// （本题来源：阿里巴巴前端笔试题）

const cookieJar = {

set (name, value, days) {

let seconds = days \* 24 \* 60 \* 60;

const d = new Date();

d.setDate(d.getDate() + days)

document.cookie = name + "=" + value + "; max-age=" + d;

},

get (name) {

const obj = {};

document.cookie.split(";").forEach( val => {

val = val.trim();

let key = val.slice(0,val.indexOf("="));

let value = val.slice(val.indexOf("=") + 1);

obj[key] = value;

});

return obj[name];

},

remove (name) {

this.set(name,'',-1);

}

}

//cankao

// const cookieJar = {

// set (name, value, days) {

// const d = new Date();

// d.setDate(d.getDate() + days)

// document.cookie = `${name}=${value}; expires=${d}`

// },

// get (name) {

// const obj = {};

// document.cookie.split(';').forEach(s => {

// s = s.trim();

// const key = s.slice(0, s.indexOf('='));

// const value = s.slice(s.indexOf('=') + 1);

// obj[key] = value;

// })

// return obj[name];

// },

// remove (name) {

// this.set(name, '', -1)

// }

// }

//

//

//

// 16

/\*

黄小明是老司机了，他每天任务就是送一群人去上班。但是每天送的人数不一样，这些人上车的时间也不一样。小明总是要等到人齐才能开车。

完成函数 driveCustomers，它接受不定数量的参数，这是参数都是函数，每个函数代表一个人。这些函数都接受一个回调函数作为参数，当回调函数被调用的时候说明这个人已经上车了，回调函数会被传入人名。例如：

const MissLi = (callback) => {

setTimeout(() => {

callback('MissLi')

}, 10) // 上车时间不一定

}

const MrWang = (callback) => {

setTimeout(() => {

callback('MrWang')

}, 3) // 上车时间不一定

}

// ..

driveCustomers(MissLi, MarWang, ...)

请你完成 driveCustomers 函数，它的作用是：当人都到齐以后，按上车的时间顺序把人名放到一个数组里面然后传给 drive 函数，正式开车。例如：drive(['MrWang', 'MissLi'])。

你只需要完成 driveCustomers 函数，drive 函数已经可以直接使用。

\*/

/\* drive 函数已经可以直接使用 \*/

const driveCustomers = (...args) => {

let all = [];

let passers = [];

all = args.reduce((acc,val) => acc.concat(new Promise( (resolve) => {

val((p)=>{

passers.push(p);

resolve();

});

})),[]);

Promise.all(all).then(drive(passers));

}

//other answer

// const driveCustomers = (...args) => {

// let customers = [];

// const promises = args.map(fn=>new Promise((resolve)=>{

// fn((p)=>{

// customers.push(p);

// resolve();

// })

// }));

// Promise.all(promises).then(() => {

// drive(customers)

// })

// }

//

//

// reference answer

//

const driveCustomers = (...fns) => {

const arr = []

fns.forEach((fn) => {

fn((name) => {

arr.push(name)

if (arr.length === fns.length) {

drive(arr)

}

})

})

}

/\*

17

在开发当中，我们经常要处理 url。而 url 上的 query string 是我们重点要处理的对象，完成一个 parseQueryString 函数。它接受一个 url 字符串作为参数，返回一个对象，这个对象包含 query string 上的键值对。例如：

parseQueryString('https://scriptoj.com/problems?offset=100&limit=10')

返回:

{ offset: '100', limit: '10'}

特殊情况说明：如果出现 ?name=&age=12 则返回 { name: '', age: '12' }，如果 ?name&age=12 则返回 { name: null, age: '12' }。

请考虑清楚 query string 可能出现的各种情况，包括可能的出现 hash 的情况（?name=jerry#nice）。

如果需要帮助，可以对照参 URI.js 的执行结果。

（本题来源：阿里巴巴前端笔试题）

\*/

var url = "https://scriptoj.comproblems/?offset=10&limit=100&tag=all?name=lucy?name=jerry#nice";

var url = " https://scriptoj.com/";

const parseQueryString = (url) => {

let start = url.indexOf('?');

let end = url.indexOf('#')!== -1?url.indexOf('#'):url.length;

if (start === -1 || end < start) {

let obj = {};

return obj;

} else {

let str = url.slice(start + 1,end);

let obj = {};

str.split('&').forEach((val) => {

let symbol\_eq = val.indexOf('=');

if (symbol\_eq !== -1) {

let key = val.slice(0,symbol\_eq);

let value = val.slice(symbol\_eq + 1,val.length);

obj[key] = value;

} else {

let key = val;

let value = null;

obj[key] = null

}

});

return obj;

}

}

//

// 注意：" https://scriptoj.com/"; 返回{}；

// 注意hash

// 注意 "https://scriptoj.comproblems/?offset=10&limit=100&tag=all?name=lucy?name=jerry#nice";

// fanhui {offset: "10", limit: "100", tag: "all", name: "lucy"} wrong

// fanhui {name: "lucy"} wrong

// fanhui {offset: "10", limit: "100", tag: "all"} wrong

// fanhui {}? wrong

//

// answer: {offset: "10", limit: "100", tag: "all?name=lucy"} WTF???????气死我了

//

//

// const parseQueryString = (url) => {

// let obj = {};

// if ((url.indexOf('#')!== -1 && url.indexOf('#') < url.indexOf('?')||url.indexOf('?')=== -1)) {

// return obj;

// }

// let arrs = url.split('?');

// let arr2 = new Array(2);

// arr2[0] = arrs[0];

// arr2[1] = "";

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

// arr2[1] += arrs[i];

// }

// arr2.forEach((arr,index) => {

// if (index === 1) {

// arr = arr.split('#')[0];

// arr.split('&').forEach((val) => {

// let key = val.split('=')[0];

// let value = val.split('=')[1] !== null ? val.split('=')[1] : null;

// obj[key] = value;

// });

// }

// });

// return obj;

// }

/\*

18

观察者模式在前端开发中非常常用，我们经常用的事件就是观察者模式的一种体现。它对我们解耦模块、开发基于消息的业务起着非常重要的作用。Node.js 原生自带 EventEmitter 模块，可见它的重要性。

完成 EventEmitter 模块，它是一个类，它的实例具有以下几个方法：on、emit、off：

on(eventName, func)：监听 eventName 事件，事件触发的时候调用 func 函数。

emit(eventName, arg1, arg2, arg3...)：触发 eventName 事件，并且把参数 arg1, arg2, arg3... 传给事件处理函数。

off(eventName, func)：停止监听某个事件。

使用例子：

const emitter = new EventEmitter()

const sayHi = (name) => console.log(`Hello ${name}`)

const sayHi2 = (name) => console.log(`Good night, ${name}`)

emitter.on('hi', sayHi)

emitter.on('hi', sayHi2)

emitter.emit('hi', 'ScriptOJ')

// => Hello ScriptOJ

// => Good night, ScriptOJ

emitter.off('hi', sayHi)

emitter.emit('hi', 'ScriptOJ')

// => Good night, ScriptOJ

const emitter2 = new EventEmitter()

emitter2.on('hi', (name, age) => {

console.log(`I am ${name}, and I am ${age} years old`)

})

emitter2.emit('hi', 'Jerry', 12)

// => I am Jerry, and I am 12 years old

（本题来源：阿里巴巴前端笔试题）

\*/

//reference answer

//

class EventEmitter {

constructor() {

this.handlers = {}

}

on(eventName, func) {

let callbacks = eventName in this.handlers ? this.handlers[eventName] : []

callbacks.push(func)

this.handlers[eventName] = callbacks

}

emit(eventName, ...args) {

if (!eventName in this.handlers) return

const callbacks = this.handlers[eventName]

callbacks.map(cb => {

cb(...args)

})

}

off(eventName, func) {

if (!eventName in this.handlers) return

let callbacks = this.handlers[eventName]

let index = callbacks.indexOf(func)

callbacks.splice(index, 1)

}

}

// 19.垂直水平居中

/\*position大法\*/

/\*#box {\*/

/\* width: 100px;\*/

/\* height: 100px;\*/

/\* position:absolute;\*/

/\* left:50%;\*/

/\* top:50%;\*/

/\* transform:translate(-50%,-50%);\*/

/\*}\*/

/\*#wrapper{\*/

/\* position:relative;\*/

/\*}\*/

/\*flex大法\*/

// #wrapper{

// display:flex;

// align-items:center;

// justify-content:center;

// }

// #box{

// width:100px;

// height:100px;

// }

//

//

// 20在开发前端框架、模版引擎的时候，经常会需要我们在特定的上下文中，动态分析、执行特定的表达式。例如：在 { x: 1, y: 2, z: 3 } 的上下文中执行表达式 x + y 那么就会得到 3，执行 z - x 就会得到 2。

// 请你完成 execute 函数，接受一个字符串和对象作为参数，它可以在特定的上下文中执行任意的表达式，例如：

// execute(`'My name is ' + name`, { name: 'Jerry' }) // => My name is Jerry

// execute('monkeys.length + 1', { monkeys: [1, 2, 3] }) // => 4

// execute('user.name + user.age', { user: { name: 'Jerry', age: 12 } }) // => Jerry12

// execute('run()', { run: () => 'Good Night' }) // => Good Night

// ...

// （你能否想出不使用 with 的方案？）

//

//

// Function

const execute = (exp, data) => new Function(...Object.keys(data),`return ${exp}`)(...Object.values(data))

// with

const execute = (exp, data) => eval(`with(data){${exp}}`)

const execute = (str,obj) => eval("with(obj){" + str + "}");

/\*

21 DOMcaozuo

循环调节列表

页面上有这么一个列表：

<ul id='adjustable-list'>

<li>

<span>1</span>

<button class='up'>UP</button>

<button class='down'>DOWN</button>

</li>

<li>

<span>2</span>

<button class='up'>UP</button>

<button class='down'>DOWN</button>

</li>

<li>

<span>3</span>

<button class='up'>UP</button>

<button class='down'>DOWN</button>

</li>

...

</ul>

点击 UP 按钮会使得该 li 元素在列表中上升一个位置，点击 DOWN 按钮会使得该 li 元素下降一个位置。点击最后的元素的 DOWN 按钮会使得元素回到第一个位置，点击第一个元素的 UP 按钮会使其回到最后的位置。

页面上已经存在该列表，你只需要完成 initAdjustableList() 函数，给元素添加事件。

\*/

const initAdjustableList = () =>{

const $ = function(id){return document.getElementById(id)}

const insertAfter = (newEle,tarEle) => {

let parent = tarEle.parentNode;

if (tarEle === parent.lastChild) {

parent.appendChild(newEle);

} else {

parent.insertBefore(newEle,tarEle.nextSibling);

}

}

$("adjustable-list").addEventListener('click',(ev)=>{

let event = ev || window.event; //兼容ie！

let target = event.target || event.srcElement;

let \_this;//该li

let ul = $("adjustable-list");

if (target.className.toLowerCase() === 'up') { //其实有问题，如果className不止是up 就无法匹配

\_this = target.parentNode; //最好用target.classList.contains('up')

if (\_this === ul.children[0]) {

ul.appendChild(\_this);

} else {

let targetEle = \_this.previousElementSibling;

ul.insertBefore(\_this,targetEle);

//insertBefore 方法会把元素先从文档树上删除，再挪动！！！

}

}

if (target.className.toLowerCase() === 'down') {

\_this = target.parentNode;

if (\_this === ul.children[ul.children.length - 1]) {

let first = ul.children[0];

//pay attention 上面不要写ul.firstChild. 因为firstCHild是一个textNode；

ul.insertBefore(\_this,first);

} else {

let targetEle = \_this.nextElementSibling;

insertAfter(\_this,targetEle);

}

}

});

}

//reference elegant

const initAdjustableList = ()=>{

let upOrDown = (e)=>{ let ev = e || window.event;

let target = ev.target;

let upBtn = ()=>{ let oli = target.parentElement;

oul.insertBefore(oli,oli.previousElementSibling);

return false;}

let dwBtn = ()=>{ let oli = target.parentElement;

let next = oli.nextElementSibling;

oli === oul.lastElementChild ?

oul.insertBefore(oli,oul.firstElementChild) : oul.insertBefore(next,oli);

return false;}

target.classList.contains('up') ? upBtn() : '';

target.classList.contains('down') ? dwBtn() : '';

return false;}

let oul = document.getElementById('adjustable-list');

oul.addEventListener('click',upOrDown);

return false;}

/\*

22

在前端开发当中，会遇到某个函数被高频率调用的情况。比如说用户疯狂地按住某个按钮，这些事件都会导致回调函数被高频地调用，但是高频调用这些函数可能会导致页面运行效率下降。

于是就有了一种 debounce 的解决方案：如果你疯狂、高频地调用某个函数，而调用之间的时间间隔低于某个时间段，这个函数最后只会被执行一次；如果高于某个时间段，则会执行多次。

请你完成 debounce，它接受两个参数，一个是被封装函数，一个是时间间隔（ms），然后返回一个函数。可以做到函数防抖的效果：

window.addEventListener('resize', debounce(() => {

console.log('Hello')

}, 100))

\*/

//思路：debounce里定义一个局部变量timer，再写一个闭包去控制这个timer。

//timer写在闭包外是只执行一次的，而闭包函数在监听过程中是执行多次的。

const debounce = (fn, duration) => {

let timer = null;

return () => {

clearTimeout(timer)

timer = setTimeout(fn, duration)

}

}

//23 给一个字符串，返回该字符串所有可能组合。比如“abc' =》 【abc,acb,bac,bca,cab,cba】

const anagrams = str => {

if (str.length <= 2) {return str.length === 2?[str,str[1]+str[0]]:[str]};

return str.split('').reduce((acc,val,index) =>

acc.concat(anagrams(str.slice(0,index) + str.slice(index+1)).map(value => val + value)),[]);

}

//24.

/\*

同字母异序指的是两个字符串字母种类和字母的数量相同，但是顺序可能不同。

完成 isAnagram，接受两个字符串作为参数，返回true 或者 false 表示这两个字符串是否同字母异序。例如：

isAnagram("anagram", "nagaram") // => return true.

isAnagram("rat", "car") // => return false.

（本题来源：github, LeetCode）

\*/

巧！

const isAnagram = (str1, str2) => str1.split('').sort().join('') === str2.split('').sort().join('')

//25

/\*

实现一个函数 where，它返回它被调用的时候所在的函数的名字，例如：

function main () {

where() // => 'main'

}

function a () {

function b () {

where() // => 'b'

}

b()

}

main()

a()

\*/

非严格模式下：

function where() {

return where.caller.name

}

答案：

function where () {

try {

throw new Error()

} catch (e) {

return e.stack.split('\n')[2].match(/at\s(.+?)\s/)[1]

}

}

必须用严格模式怎么写：

const where = () => {

let reg = /\s+at\s(\S+)\s\(/g

let str = new Error().stack.toString()

let res = reg.exec(str) && reg.exec(str)

return res && res[1]

}

/\*

26.国家有重要的任务要托付给你：完成间谍函数 spy，它可以潜伏到任何一个函数当中，监听它们的所有活动。spy 接受一个函数作为参数，返回一个被间谍潜伏以后的函数。

let america = (a, b) => a + b

america = spy(america)

america(1, 2) // => 3

spy 返回的函数和原来的函数的功能一样，但是它悄悄记录了每一次执行的参数和执行结果，都存放到一个 calls 数组里面：

america(1, 2)

america(3, 4)

america.calls[0].args // => [1, 2]

america.calls[0].result // => 3

america.calls[1].args // => [3, 4]

america.calls[1].result // => 7

注意，spy 可以支持潜伏到对象方法当中：

let user = {

name: 'Jerry',

getName () {

return this.name

}

}

user.getName = spy(user.getName)

user.getName() // => 'Jerry'

user.getName.calls[0].result // => 'Jerry'

另外，不要修改被 spy 的函数（你应该返回一个全新的函数），否则会被敌人发现。

\*/

const spy = (fn) => {

let calls = [];

let a = function(...args){

let res = {};

res.args = args;

res.result = fn.apply(this,args);

calls.push(res);

return res.result;

}

a.calls = calls

return a

}

const f1 = (a,b,c) => a+b+c;

// function a(foo){

// return function(args){

// return foo;

// }

// }

//reference

const spy = function(fn) {

var calls = []

var f = function() {

var obj = {

args: [...arguments],

result: fn.apply(this, arguments)

}

calls.push(obj)

return obj.result

}

f.calls = calls

return f

}

/\*

27.时间停止

pause 函数可以让一个函数暂停运行一段时间（ms）以后继续运行。例如：

async function run () {

console.log('Hello')

await pause(1000) // 续一秒

console.log('World') // 一秒以后继续运行

}

\*/

const pause = async (time) => new Promise((resolve) => setTimeout(resolve,time))

/\*

28 js数据类型的判断

最新的 Javascript 标准规定了六种基本数据类型(number, null, undefined, string, boolean, symbol) 和基于 Object 衍生的其它原生数据类型，写出 type 函数，它传入一个参数，返回它的数据类型（用小写字母），比如: type(new Date)，返回 date。

\*/

const type = (obj) =>

typeof(obj) === "object"

?(obj === null?'null':obj.constructor.name.toLowerCase())

:typeof(obj)

/\*

29 Proxy!!!!!!!

Tomy 非常敏感，不喜欢别人碰他的东西。一旦有人碰他就会大喊 Don't Touch Me.。

完成 tomy 这个对象，禁止对 tomy 的内容进行修改（增加、修改、删除）。一旦有人对 tomy 进行任何的修改，都用 console.log 打印 Don't Touch Me.。

下面是 Proxy 支持的拦截操作一览，一共 13 种。

get(target, propKey, receiver)：拦截对象属性的读取，比如proxy.foo和proxy['foo']。

set(target, propKey, value, receiver)：拦截对象属性的设置，比如proxy.foo = v或proxy['foo'] = v，返回一个布尔值。

has(target, propKey)：拦截propKey in proxy的操作，返回一个布尔值。

deleteProperty(target, propKey)：拦截delete proxy[propKey]的操作，返回一个布尔值。

ownKeys(target)：拦截Object.getOwnPropertyNames(proxy)、Object.getOwnPropertySymbols(proxy)、Object.keys(proxy)，返回一个数组。该方法返回目标对象所有自身的属性的属性名，而Object.keys()的返回结果仅包括目标对象自身的可遍历属性。

getOwnPropertyDescriptor(target, propKey)：拦截Object.getOwnPropertyDescriptor(proxy, propKey)，返回属性的描述对象。

defineProperty(target, propKey, propDesc)：拦截Object.defineProperty(proxy, propKey, propDesc）、Object.defineProperties(proxy, propDescs)，返回一个布尔值。

preventExtensions(target)：拦截Object.preventExtensions(proxy)，返回一个布尔值。

getPrototypeOf(target)：拦截Object.getPrototypeOf(proxy)，返回一个对象。

isExtensible(target)：拦截Object.isExtensible(proxy)，返回一个布尔值。

setPrototypeOf(target, proto)：拦截Object.setPrototypeOf(proxy, proto)，返回一个布尔值。如果目标对象是函数，那么还有两种额外操作可以拦截。

apply(target, object, args)：拦截 Proxy 实例作为函数调用的操作，比如proxy(...args)、proxy.call(object, ...args)、proxy.apply(...)。

construct(target, args)：拦截 Proxy 实例作为构造函数调用的操作，比如new proxy(...args)。

\*/

const tomy = new Proxy({},{

set:function(target,key,value) {

console.log("Don't Touch Me.");

},

deleteProperty:function(target,key) {

console.log("Don't Touch Me.");

return false;//如果这个方法抛出错误或者返回false，当前属性就无法被delete命令删除。

},

defineProperty:function(target,key) {

console.log("Don't Touch Me.");

return false;

}

});

/\*

30 render

模版引擎是在前端是非常常用的一种工具。请你完成一个简单的模版引擎的 render 函数，它可以接受模版字符串和一个数据对象作为参数。函数执行返回渲染以后的模版字符串，例如：

const templateStr = `

<ul class="users">

<% users.forEach((user) => { %>

<li class="user-item">

<%= 'My name is ' + user.name %>

</li>

<% }) %>

</ul>

`

const data = {

users: [

{ name: 'Jerry', age: 12 },

{ name: 'Lucy', age: 13 },

{ name: 'Tomy', age: 14 }

]

}

render(templateStr, data)

/\*返回结果：

<ul class="users">

<li class="user-item">

My name is Jerry

</li>

<li class="user-item">

My name is Lucy

</li>

<li class="user-item">

My name is Tomy

</li>

</ul>

<% 和 %> 之间可以放置任意的 JavaScript 代码，而 <%= 和 %> 之间执行任意的 JavaScript 表达式并且输出在模版上；传入的 data 可以作为模版引擎执行的上下文进行数据的引用，请你完成 render 函数。

\*/

const render = (str,obj) => new Function(...Object.keys(obj),'return ' + str)(...Object.values(obj));

const render = (template, data) => /\* TODO \*/

{

let exp0 = /<%=(.+)%>/g

let exp1 = /<%(.+)%>/g

template = template.replace(exp0,'`);\n echo($1) \n echo(`').replace(exp1,'`);\n $1 \n echo(`')

template = 'echo(`' + template +'`)'

//${template}的结果为 调用若干echo函数

let comp = `

let html = ''

function echo(t){

html+=t

}

${template}

return html

`

return Function(...Object.keys(data),comp)(...Object.values(data))

}

正则实现

const render = (template, data) => {

const result =

`var p=[];with(data){p.push('` + template

.replace(/[\r\n\t]/g,'')

.replace(/<%=(.\*?)%>/g,`');p.push($1);p.push('`)

.replace(/<%/g,`');`)

.replace(/%>/g,`;p.push('`)

+ `');} p.join('');`;

return eval(result);

}

var str = `<ul class="users">

<% users.forEach((user) => { %>

<li class="user-item">

<%= 'My name is ' + user.name %>

</li>

<% }) %>

</ul>`

function a1(data,path){

let p = new Proxy(target,{

get:function(target,key){

if (Object.keys(target).includes(key)) {

return target[key]; //写data.key就不对???

} else {

return 'undefined';

}

}

});

return p.path

}

var data = { a: { b: { c: 'ScriptOJ' } } }

1.31.2018

在文件/home/somebody/workspace/somemodule.js中第一行引用了一个模块：require(‘othermodule‘)，请问required 的查找模块的顺序   
A. /home/somebody/workspace/node\_modules/othermodule/index.js   
B. /home/somebody/workspace/node\_modules/othermodule. Js   
C.CORE MODULES named othermodule   
D./home/somebody/node\_modules/othermodule/index.js

C B A D

首先加载核心模块,不管有没有同名/同目录的情况下,核心模块优先加载.

其次按照相对路径/绝对路径加载文件模块(加载顺序,首先试图按照路径查找 .js 扩展名的文件,如果没有,试图按照路径查找 .json 扩展名的文件,如果还是没有,就按照路径查找 .node 扩展名的c++模块了)

最后搜索 node\_modules 目录下通过npm下载的第三方模块.

注意:首次加载这类模块最慢,因为执行文件所在目录的node\_mondel 文件夹下找不到时,会去父级node\_mondel 文件夹里查找,如果还是找不到会去父级的父级node\_mondel 文件夹里查找.......但是,只要首次加载成功后,node就会缓存起来,它缓存的是编译后的二进制模块,所以以后的加载速度和效率都的有保证的.

下面哪些方法可以用作javascript异步模式的编程？

回调函数

事件监听

发布/订阅

Promises对象

正确答案: A B C D   你的答案: A B D (错误)

1.回调函数  
f1();  
f2();  
function f1(callback){  
setTimeout(function () {  
// f1的任务代码  
callback();  
}, 1000);  
}  
f1(f2);  
  
2.事件监听  
f1.on('done', f2);  
function f1(){  
setTimeout(function () {  
// f1的任务代码  
f1.trigger('done');  
}, 1000);  
}  
  
3.发布/订阅  
jQuery.subscribe("done", f2);  
function f1(){  
setTimeout(function () {  
// f1的任务代码  
jQuery.publish("done");  
}, 1000);  
}  
jQuery.unsubscribe("done", f2);  
  
4.Promises对象  
f1().then(f2);  
function f1(){  
var dfd = $.Deferred();  
setTimeout(function () {  
// f1的任务代码  
dfd.resolve();  
}, 500);  
return dfd.promise;  
}  
指定多个回调函数：  
f1().then(f2).then(f3);  
指定发生错误时的回调函数：  
f1().then(f2).fail(f3);

H5新增的表单元素

datalist规定输入域的选项列表，通过option创建！

keygen提供一种验证用户的可靠方法，密钥对生成器，私钥存于客户端，公钥发到服务器，用于之后验证客户端证书！

output元素用于不同类型的输出！

在HTML中，（ ）可以在网页上通过链接直接打开邮件客户端发送邮件

<a href=”mailto:ming.zhou@nowcoder.com”>发送邮件</a>

a) 置换元素：浏览器根据元素的标签和属性，来决定元素的具体显示内容。   
例如：浏览器会根据<img>标签的src属性的 值来读取图片信息并显示出来，而如果查看(x)html代码，则看不到图片的实际内容；<input>标签的type属性来决定是显示输入 框，还是单选按钮等。 (x)html中 的<img>、<input>、<textarea>、<select>、<object> 都是置换元素。这些元素往往没有实际的内容，即是一个空元素。

置换元素在其显示中生成了框，这也就是有的内联元素能够设置宽高的原因。

In CSS, a  **replaced element**  is an element whose representation is outside the scope of CSS. These are kind of external objects whose representation is independent of the CSS. Typical replaced elements are  [<img>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/img) ,  [<object>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/object) ,  [<video>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/video)  or form elements like[<textarea>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/textarea)  and  [<input>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/input) .

w3c文档的定义，置换元素的意思是  展现不是由CSS来控制的元素。这些外部元素的展现不依赖于CSS规范

b) 不可替换元素：(x)html 的大多数元素是不可替换元素，即其内容直接表现给用户端（如浏览器）。

例如： <label>label中的内容</label> 标签<label>是一个非置换元素，文字label中的内容”将全被显示

下列说法正确的是（）

正确答案: A B   你的答案: A B C (错误)

display: none；不为被隐藏的对象保留其物理空间；

visibility:hidden;所占据的空间位置仍然存在,仅为视觉上的完全透明；

visibility:hidden;产生reflow和repaint(回流与重绘)；

visibility:hidden;与display: none;两者没有本质上的区别；

display: none和visibility:hidden的区别就是visibility:hidden会保留元素的空间

repaint(重绘) ，repaint发生更改时，元素的外观被改变，且在没有改变布局的情况下发生，如改变outline,visibility,background color，不会影响到dom结构渲染。

reflow(渲染)，与repaint区别就是他会影响到dom的结构渲染，同时他会触发repaint，他会改变他本身与所有父辈元素(祖先)，这种开销是非常昂贵的，导致性能下降是必然的，页面元素越多效果越明显。

所以display:none才会产生reflow

visibility:hidden只会出发repaint

英文字母全部转为大写正确的是（text-transform: uppercase;）

在引入线程的操作系统中，通常都是把进程作为分配资源的基本单位，而把线程作为独立运行和独立调度的基本单位 。

下面那个页面调度算法,当进程分配到的页面数增加时,缺页中断的次数可能增加也可能减少

正确答案: A   你的答案: B (错误)

FIFO算法

LRU算法

Clock算法

LFU算法

在操作系统的内存管理中，有一类很重要的算法就是**内存页面置换算法**（包括FIFO，LRU,LFU等几种常见页面置换算法）。事实上，Cache算法和内存页面置换算法的核心思想是一样的：都是在给定一个限定大小的空间的前提下，设计一个原则如何来更新和访问其中的元素。

**LRU**全称是Least Recently Used，即最近最久未使用的意思。 下面说一下LRU算法的核心思想，LRU算法的设计原则是：如果一个数据在最近一段时间没有被访问到，那么在将来它被访问的可能性也很小。也就是说，当限定的空间已存满数据时，应当把最久没有被访问到的数据淘汰。

**FIFO**（First in First out），先进先出。在FIFO Cache设计中，核心原则就是： 如果一个数据最先进入缓存中，则应该最早淘汰掉 。也就是说，当缓存满的时候，应当把最先进入缓存的数据给淘汰掉。

**LFU**（Least Frequently Used）最近最少使用算法。它是基于“如果一个数据在最近一段时间内使用次数很少，那么在将来一段时间内被使用的可能性也很小”的思路。

注意LFU和LRU算法的不同之处，LRU的淘汰规则是基于访问时间，而LFU是基于访问次数的。

发表于 2016-08-18 10:06:26

排序算法：

不稳定：快选堆希

稳    定：插冒归基

/定义一个log函数 使之可以代理console.log的功能

const log = (...args) => console.log(args);//返回的好像是一个数组 和console。log出来的不太一样

function log2(){

console.log.apply(console, arguments);

};//这个方法可以

plus 注意 ...args之后 args是数组

而是用arguments的话，它是一个伪数组。需要转化成数组之后才能是用数组的方法。

let args = [].slice.apply(arguments);

args.unshift('app'); //往队首添加！

1.31

各种排序算法时间复杂度

各种排序算法比较

1、冒泡排序是一种用时间换空间的排序方法，n小时好

2、最坏情况是把顺序的排列变成逆序，或者把逆序的数列变成顺序，最差时间复杂度O(N^2)只是表示其操作次数的数量级

3、最好的情况是数据本来就有序，复杂度为O(n)

注：

1、归并排序每次递归都要用到一个辅助表，长度与待排序的表长度相同，虽然递归次数是O(log2n)，但每次递归都会释放掉所占的辅助空间，

2、快速排序空间复杂度只是在通常情况下才为O(log2n)，如果是最坏情况的话，很显然就要O(n)的空间了。当然，可以通过随机化选择pivot来将空间复杂度降低到O(log2n)。

相关概念：

1、时间复杂度

时间复杂度可以认为是对排序数据的总的操作次数。反映当n变化时，操作次数呈现什么规律。

常见的时间复杂度有：常数阶O(1),对数阶O(log2n),线性阶O(n), 线性对数阶O(nlog2n),平方阶O(n2)

时间复杂度O(1)：算法中语句执行次数为一个常数，则时间复杂度为O(1),

2、空间复杂度

空间复杂度是指算法在计算机内执行时所需存储空间的度量，它也是问题规模n的函数

空间复杂度O(1)：当一个算法的空间复杂度为一个常量，即不随被处理数据量n的大小而改变时，可表示为O(1)

空间复杂度O(log2N)：当一个算法的空间复杂度与以2为底的n的对数成正比时，可表示为O(log2n)

ax=N，则x=logaN，

空间复杂度O(n)：当一个算法的空间复杂度与n成线性比例关系时，可表示为0(n).

如何判断一个js对象是否是Array,arr为要判断的对象，其中最准确的方法是？

正确答案: D 你的答案: B (错误)

typeof(arr)

arr instanceof Array

arr.toString==='[object Array]';

Object.prototype.toString.call(arr) === '[object Array]';

typeof(arr) 返回的是 Object

instanceof 在跨 frame 对象构建的场景下会失效

arr.toString 没有这种用法，正确的用法是 arr.toString() 返回的是数组的内容

关于javascript，以下选项描述错误的是

正确答案: D 你的答案: D (正确)

在原型上扩展的可枚举方法，会被for in循环出来

使用object.defineProperty可向对象添加或者修改属性

每个对象都有prototype属性，返回对象类型原型的引用

通过hasOwnProperty可判断一个对象以及其原型链上是否具有指定名称的属性

原型链是JS实现继承的一种模型

For循环是按顺序的，for in 循环是不一定按顺序的

下列代码，页面打开后能够弹出alert(1)的是？

正确答案: A B C 你的答案: B C (错误)

<iframe src=”javascript: alert(1)”></iframe>

<img src=”” onerror=”alert(1)”/>

IE下<s style=”top:expression(alert(1))”></s>

<div onclick=”alert(1)”></div>

A 加载页面的时候触发；

B onerror 事件 当图片不存在时,将触发；

C 在ie 7下会连续弹出， IE5及其以后版本支持在CSS中使用expression，用来把CSS属性和Javascript表达式关联起来，

这里的CSS属性可以是元素固有的属性，也可以是自定义属性。就是说CSS属性后面可以是一段Javascript表达式，

CSS属性的值等于Javascript表达式计算的结果。 在表达式中可以直接引用元素自身的属性和方法，也可以使用其他浏览器对象。

这个表达式就好像是在这个元素的一个成员函数中一样。参考资料 http://www.blueidea.com/tech/site/2006/3705.asp

var myObject = {

foo: "bar",

func: function() {

var self = this;

console.log(this.foo);

console.log(self.foo);

(function() {

console.log(this.foo);

console.log(self.foo);

}());

}

};

myObject.func();

程序的输出是什么？

bar

bar

undefined //立即执行函数表达式的this指向window！

bar

假设 output 是一个函数，输出一行文本。下面的语句输出结果是什么？

output(typeof (function() {output(“Hello World!”)})());

Hello World! undefined

1.先立即执行匿名函数，输出Hello World！

2.函数执行后无返回值，则输出未定义

1.Ajax的优势：1.可搜索性 2.开放性 3.费用 4.易用性 5.易于开发。

2.Flash的优势：1.多媒体处理 2.兼容性 3.矢量图形 4.客户端资源调度

3.Ajax的劣势：1.它可能破坏浏览器的后退功能 2.使用动态页面更新使得用户难于将某个特定的状态保存到收藏夹中 ，不过这些都有相关方法解决。

4.Flash的劣势：1.二进制格式 2.格式私有 3.flash 文件经常会很大，用户第一次使用的时候需要忍耐较长的等待时间 4.性能问题

w3c 制定的 javascript 标准事件模型，以下正确的顺序以及描述是

事件捕获->事件处理->事件冒泡

2/2

noscript 元素用来定义在脚本未被执行时的替代内容（文本）。

<body>

...

...

<script type="text/javascript">

<!--

document.write("Hello World!")

//-->

</script><noscript>Your browser does not support JavaScript!</noscript>...

...

</body>

请阅读以下代码：

<div style=”width:400px;height:200px;”>

<span style=”float:left;width:auto;height:100%;”>

<i style=”position:absolute;float:left;width:100px;height:50px;”>hello</i>

</span>

</div>

问题：span标签的width和height分别为多少？

width = 0px，height = 200px

这题考点其实有几个：

1.span标签是无法设置宽高的；

2.float会把浮动元素变成块级元素；

3.绝对定位脱离了文档流

解释性语言和编译性语言的定义：

计算机不能直接理解高级语言，只能直接理解机器语言，所以必须要把高级语言翻译成机器语言，计算机才能执行高级语言编写的程序。

翻译的方式有两种，一个是编译，一个是解释。两种方式只是翻译的时间不同。

解释性语言的定义：

解释性语言的程序不需要编译，在运行程序的时候才翻译，每个语句都是执行的时候才翻译。这样解释性语言每执行一次就需要逐行翻译一次，效率比较低。

现代解释性语言通常把源程序编译成中间代码，然后用解释器把中间代码一条条翻译成目标机器代码，一条条执行。

编译性语言的定义：

编译性语言写的程序在被执行之前，需要一个专门的编译过程，把程序编译成为机器语言的文件，比如exe文件，以后要运行的话就不用重新翻译了，直接使用编译的结果就行了（exe文件），因为翻译只做了一次，运行时不需要翻译，所以编译型语言的程序执行效率高。

非独立：JavaScript语言依赖执行环境，对于客户端来说是浏览器，对于服务端来说是node。

效率低：执行前不需要编译，执行时才编译，因此效率低

History对象

length 返回浏览器历史列表中的URL数量

back() 加载 history 列表中的前一个URL

forward() 加载 history 列表中的下一个URL

go() 加载history列表中的某个具体页面。

void是一元运算符，它出现在操作数之前，操作数可以是任意类型，操作数会照常计算，但忽略计算结果并返回undefined。由于void会忽略操作数的值，因此在操作数具有副作用的时候使用void来让程序更具语义

console.log(void 0); // undefined

console.log(void(0)); // undefined

常见的作用:

1.替代undefined

由于undefined并不是一个关键字，其在IE8-浏览器中会被重写，在高版本函数作用域中也会被重写；所以可以用void 0 来替换undefined

var undefined = 10;

console.log(undefined);//IE8-浏览器下为10，高版本浏览器下为undefined

function test(){

var undefined = 10;

console.log(undefined);

}

console.log(test());//所有浏览器下都是10

2.客户端URL

这个运算符最常用在客户端URL——javascript:URL中，在URL中可以写带有副作用的表达式，而void则让浏览器不必显示这个表达式的计算结果。例如，经常在HTML代码中的<a>标签里使用void运算符

<a href="javascript:void window.open();">打开一个新窗口</a>

3.阻止默认事件

阻止默认事件的方式是给事件置返回值false

//一般写法

<a href="http://example.com" onclick="f();return false;">文字</a>

使用void运算符可以取代上面写法

<a href="javascript:void(f())">文字</a>

void();

上面表达式的结果是：

SyntaxError

void 作为运算符后面接的是表达式，void expression。而void(0)也是被当做void 0。

如果直接void()，那么我觉得应该是把void当做函数使用了，但是此时void并没有定义。不过void不能当函数使用所以才报SyntaxError（语法错误）

下面关于DNS说法正确的是（）

正确答案: B C 你的答案: A C D (错误)

DNS的作用是域名和IP地址的相互映射

DNS协议运行在UDP协议之上

DNS协议端口号为53

DNS的默认缓存时间为1小时

A：DNS就是将域名翻译成IP地址。

B：主要用UDP，但是当请求字节过长超过512字节时用TCP协议，将其分割成多个片段传输。

C：DNS协议默认端口号是53。

D：操作系统的DNS缓存：windows DNS缓存的默认值是 MaxCacheTTL，它的默认值是86400s，也就是一天。macOS 严格遵循DNS协议中的TTL。

游览器的DNS缓存：chrome对每个域名会默认缓存60s；IE将DNS缓存30min；Firefox默认缓存时间只有1分钟；Safari约为10S。

＝>综上所述，选择BC。

console.log(([])?true:false);

console.log(([]==false?true:false));

console.log(({}==false)?true:false)

true true false

答案选C：“true、true、false”。此题考察类型转换，三元运算符先**“分清是非”**，再决定今后该走哪条路，“==”运算符比较**“喜欢”**Number类型。（我就不严谨地这么一说，若要刨根问底，请看《JavaScript权威指南》）

下面是题目的类型转换结果：

Boolean([]); //true

Number([]); //0

Number({}); // NaN

Number(false); //0

因此：

console.log(([])?true:fasle);// => console.log((true)?true:false);

console.log([]==false?true:false); // => console.log(0==0?true:false);

console.log(({}==false)?true:false); // => console.log((NaN==0)?true:false);

#### 《JavaScript权威指南》的部分相关知识点

**“==”运算符（两个操作数的类型不相同时）**

* 如果一个值是null，另一个值是undefined，则它们相等
* 如果一个值是数字，另一个值是字符串，先将字符串转换为数学，然后使用转换后的值进行比较。
* 如果其中一个值是true，则将其转换为1再进行比较。如果其中的一个值是false，则将其转换为0再进行比较。
* 如果一个值是对象，另一个值是数字或字符串，则将对象转换为原始值，再进行比较。

**对象到数字的转换**

* 如果对象具有valueOf()方法，后者返回一个原始值，则JavaScript将这个原始值转换为数字（如果需要的话）并返回一个数字。
* 否则，如果对象具有toString()方法，后者返回一个原始值，则JavaScript将其转换并返回。（对象的toString()方法返回一个字符串直接量（作者所说的原始值），JavaScript将这个字符串转换为数字类型，并返回这个数字）。
* 否则，JavaScript抛出一个类型错误异常。

**空数组转换为数字0**

* 数组继承了默认的valueOf()方法，这个方法返回一个对象而不是一个原始值，因此，数组到数学的转换则调用toString()方法。空数组转换为空字符串，空字符串转换为数字0.

说一说常见的请求头和相应头都有什么呢？

1)请求(客户端->服务端[request])   
    GET(请求的方式) /newcoder/hello.html(请求的目标资源) HTTP/1.1(请求采用的协议和版本号)   
    Accept: \*/\*(客户端能接收的资源类型)   
    Accept-Language: en-us(客户端接收的语言类型)   
    Connection: Keep-Alive(维护客户端和服务端的连接关系)   
    Host: localhost:8080(连接的目标主机和端口号)   
    Referer: http://localhost/links.asp(告诉服务器我来自于哪里)   
    User-Agent: Mozilla/4.0(客户端版本号的名字)   
    Accept-Encoding: gzip, deflate(客户端能接收的压缩数据的类型)   
    If-Modified-Since: Tue, 11 Jul 2000 18:23:51 GMT(缓存时间)    
    Cookie(客户端暂存服务端的信息)

    Date: Tue, 11 Jul 2000 18:23:51 GMT(客户端请求服务端的时间)

2)响应(服务端->客户端[response])  
    HTTP/1.1(响应采用的协议和版本号) 200(状态码) OK(描述信息)  
    Location: http://www.baidu.com(服务端需要客户端访问的页面路径)   
    Server:apache tomcat(服务端的Web服务端名)  
    Content-Encoding: gzip(服务端能够发送压缩编码类型)   
    Content-Length: 80(服务端发送的压缩数据的长度)   
    Content-Language: zh-cn(服务端发送的语言类型)   
    Content-Type: text/html; charset=GB2312(服务端发送的类型及采用的编码方式)  
    Last-Modified: Tue, 11 Jul 2000 18:23:51 GMT(服务端对该资源最后修改的时间)  
    Refresh: 1;url=http://www.it315.org(服务端要求客户端1秒钟后，刷新，然后访问指定的页面路径)  
    Content-Disposition: attachment; filename=aaa.zip(服务端要求客户端以下载文件的方式打开该文件)  
    Transfer-Encoding: chunked(分块传递数据到客户端）    
    Set-Cookie:SS=Q0=5Lb\_nQ; path=/search(服务端发送到客户端的暂存数据)  
    Expires: -1//3种(服务端禁止客户端缓存页面数据)  
    Cache-Control: no-cache(服务端禁止客户端缓存页面数据)    
    Pragma: no-cache(服务端禁止客户端缓存页面数据)     
    Connection: close(1.0)/(1.1)Keep-Alive(维护客户端和服务端的连接关系)

    Date: Tue, 11 Jul 2000 18:23:51 GMT(服务端响应客户端的时间)

**在服务器响应客户端的时候，带上Access-Control-Allow-Origin头信息，解决跨域的一种方法。**

**页面的性能指标详解：**

白屏时间（first Paint Time）——用户从打开页面开始到页面开始有东西呈现为止

首屏时间——用户浏览器首屏内所有内容都呈现出来所花费的时间

用户可操作时间(dom Interactive)——用户可以进行正常的点击、输入等操作，默认可以统计**domready**时间，因为通常会在这时候绑定事件操作

总下载时间——页面所有资源都加载完成并呈现出来所花的时间，即页面 onload 的时间

先触发first paint，后触发dom ready

<SCRIPT LANGUAGE="JavaScript">

var bb = 1;

function aa(bb) {

    bb = 2;

    alert(bb);

};

aa(bb);

alert(bb);

</SCRIPT>

//2 1

**1.操作符**

**（1）typeof操作符**

格式：result=typeof variable

返回值：

undefined 值未定义

boolean 布尔值

string 字符串

number 数值

object 对象、null

function 函数

**（2）instanceof操作符**

格式：result=variable instanceof constructor

返回值：

true

false

**2.** **方法**

String.prototype.isString=function(str){

return ((str instanceof String) || (typeof str).toLowerCase() == 'string');

}

**3.原理**

var str1=new String('str1');

var str2='str2';

console.log(typeof str1);//object

console.log(typeof str2);//string

console.log(str1 instanceof String);//true

console.log(str2 instanceof String);//false