掘金面试题总结

<https://juejin.im/post/5daeefc8e51d4524f007fb15#heading-96>

便利蜂面试：

全排列算法

<https://www.jb51.net/article/39291.htm>

/\*

全排列（递归链接）算法

1、设定源数组为输入数组，结果数组存放排列结果（初始化为空数组）；

2、逐一将源数组的每个元素链接到结果数组中（生成新数组对象）；

3、从原数组中删除被链接的元素（生成新数组对象）；

4、将新的源数组和结果数组作为参数递归调用步骤2、3，直到源数组为空，则输出一个排列。

\*/

var count=0;

function show(arr) {

    document.write("P<sub>"+ ++count+"</sub>: "+arr+"<br />");

}

function perm(arr) {

    (function fn(source, result) {

        if (source.length == 0)

            show(result);

        else

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

                fn(source.slice(0, i).concat(source.slice(i + 1)), result.concat(source[i]));

    })(arr, []);

}

perm(["e1", "e2", "e3", "e4"]);

/\*

全排列（递归交换）算法

1、将第一个位置分别放置各个不同的元素；

2、对剩余的位置进行全排列（递归）；

3、递归出口为只对一个元素进行全排列。

\*/

function swap(arr,i,j) {

    if(i!=j) {

        var temp=arr[i];

        arr[i]=arr[j];

        arr[j]=temp;

    }

}

var count=0;

function show(arr) {

    document.write("P<sub>"+ ++count+"</sub>: "+arr+"<br />");

}

function perm(arr) {

    (function fn(n) { //为第n个位置选择元素

        for(var i=n;i<arr.length;i++) {

            swap(arr,i,n);

            if(n+1<arr.length-1) //判断数组中剩余的待全排列的元素是否大于1个

                fn(n+1); //从第n+1个下标进行全排列

            else

                show(arr); //显示一组结果

            swap(arr,i,n);

        }

    })(0);

}

perm(["e1","e2","e3","e4"]);

原子化正则，找到最小单位

2[ab3[c]] 解码成 abcccabccc

最小单位：

var reg = /(\d+)\[([a-z]+)\]/g

cdn缓存清理

<https://blog.csdn.net/wangxin1982314/article/details/51050206>

CDN缓存的缺点：

CDN的分流作用不仅减少了用户的访问延时，也减少的源站的负载。但其缺点也很明显：当网站更新时，如果CDN节点上数据没有及时更新，即便用户再浏览器使用Ctrl +F5的方式使浏览器端的缓存失效，也会因为CDN边缘节点没有同步最新数据而导致用户访问异常。

CDN缓存刷新：

CDN边缘节点对开发者是透明的，相比于浏览器Ctrl+F5的强制刷新来使浏览器本地缓存失效，开发者可以通过CDN服务商提供的“刷新缓存”接口来达到清理CDN边缘节点缓存的目的。这样开发者在更新数据后，可以使用“刷新缓存”功能来强制CDN节点上的数据缓存过期，保证客户端在访问时，拉取到最新的数据。

浏览器缓存刷新

1.  在地址栏中输入网址后按回车或点击转到按钮

浏览器以最少的请求来获取网页的数据，浏览器会对所有没有过期的内容直接使用本地缓存，从而减少了对浏览器的请求。所以，Expires，max-age强缓存依然有效。

2.  按F5或浏览器刷新按钮

浏览器会在请求中附加必要的缓存协商，但不允许浏览器直接使用本地缓存，Last-Modified、ETag依然有效，但是Expires失效。

3.  按Ctrl+F5或按Ctrl并点击刷新按钮

这种方式就是强制刷新，总会发起一个全新的请求，不使用任何缓存。

实现promise.all

有两个面试官说要让这个方法=new Promise(…)

根据mdn的定义：**Promise.all(iterable)** 方法返回一个 [Promise](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Promise) 实例，此实例在 iterable 参数内所有的 promise 都“完成（resolved）”或参数中不包含 promise 时回调完成（resolve）；如果参数中  promise 有一个失败（rejected），此实例回调失败（reject），失败原因的是第一个失败 promise 的结果。

只要让Promise.all方法执行后返回Promise实例就行

https://www.jianshu.com/p/f3413db4f64f

//方法要加在prototype上

**Promise**.prototype.**all** = function(promises) {

    if (!**Array**.**isArray**(promises)) {

        throw new **TypeError**("You must pass array");

    }

    return new **Promise**(function(resolve, reject) {

        var result = [],

            count = promises.length;

        function **resolver**(value) {

**resolveAll**(value);

        }

        function **rejecter**(reason) {

**reject**(reason);

        }

        function **resolveAll**(value) {

            result.**push**(value);

            if (--count == 0) {

**resolve**(result);

            }

        }

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

            promises[i].**then**(resolver, rejecter);

        }

    });

};

promise静态方法

可以使用静态Promise.resolve方法，Promise.resolve(value)方法返回一个以给定值解析后的[Promise](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Promise) 对象。

Promise.resolve("Success").then(function(value) {

console.log(value); // "Success"

}, function(value) {

// 不会被调用

});

和**promise**相关的题目里要注意函数有没有返回**promise**对象，没有返回**thenable**对象的话是没有**then**方法的

fn(…arg) 剩余参数

https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Functions/Rest\_parameters

剩余参数语法允许我们将一个不定数量的参数表示为一个数组

https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/Spread\_syntax

剩余语法(Rest syntax) 看起来和展开语法完全相同，不同点在于展开语法将数组展开为其中的各个元素，而剩余语法则是将多个元素收集起来并“凝聚”为单个元素。

jsBridge

<https://blog.csdn.net/yuzhengfei7/article/details/93468914>

一、什么是JSBridge

是 Native 和非 Native 之间的桥梁，它的核心是构建 Native 和非 Native 间消息通信的通道，而且这个通信的通道是双向的。

二、JSBridge 的实现原理

JavaScript 是运行在一个单独的 JS Context 中（例如，WebView 的 Webkit 引擎、JSCore）。由于这些 Context 与原生运行环境的天然隔离，我们可以将这种情况与 RPC（Remote Procedure Call，远程过程调用）通信进行类比，将 Native 与 JavaScript 的每次互相调用看做一次 RPC 调用。

在 JSBridge 的设计中，可以把前端看做 RPC 的客户端，把 Native 端看做 RPC 的服务器端，从而 JSBridge 要实现的主要逻辑就出现了：通信调用（Native 与 JS 通信） 和句柄解析调用。

三、JSBridge 的通信原理

1.JavaScript 调用 Native的方式

主要是注入API

注入 API 方式的主要原理是，通过 WebView 提供的接口，向 JavaScript 的 Context（window）中注入对象或者方法，让 JavaScript 调用时，直接执行相应的 Native 代码逻辑，达到 JavaScript 调用 Native 的目的。

2.Native 调用 JavaScript 的方式

相比于 JavaScript 调用 Native， Native 调用 JavaScript 较为简单，直接执行拼接好的 JavaScript 代码即可。

Extend class 与 \_\_proto\_\_

class **a** {

    count = 1;

    getCount() {

        return this.count;

    }

}

class **b** extends **a** {

    count = 2;

}

let c = new **a**();

let d = new **b**();

d.\_\_proto\_\_.**getCount**(); //undefined

《你不知道的js-下卷》p190

class **Foo** {

    constructor(a, b) {

        this.x = a;

        this.y = b;

    }

**gimmeXY**() {

        return this.x \* this.y;

    }

}

可以把前面代码中的 class 语法定义粗略理解为下面这个等价前 ES6 代码，之前有过原型 风格编码经验的开发者可能会对此非常熟悉:

function **Foo**(a, b) {

    this.x = a;

    this.y = b;

}

**Foo**.prototype.**gimmeXY** = function() {

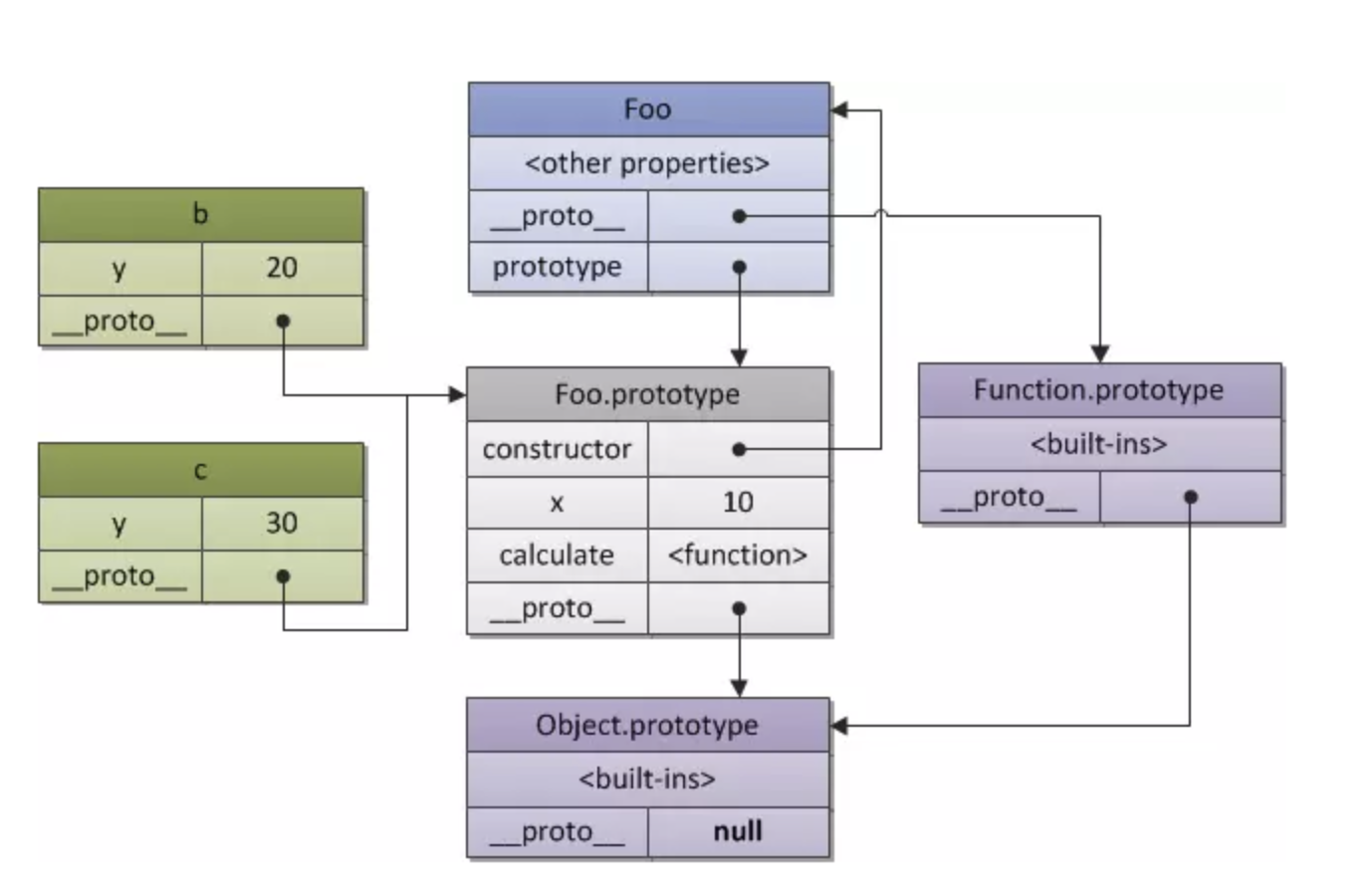
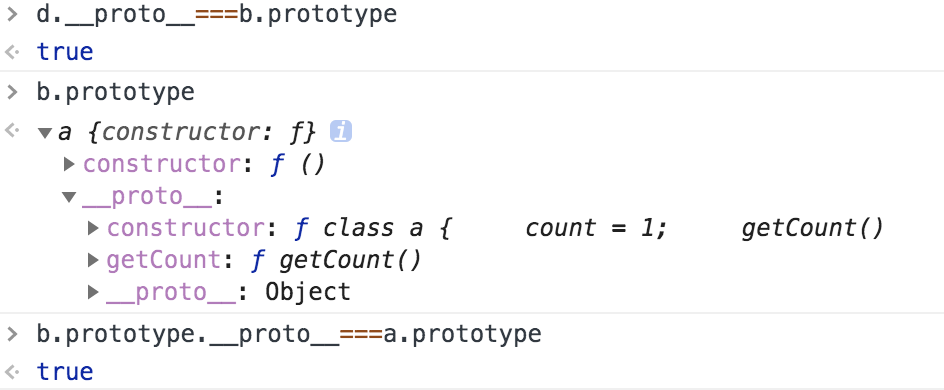
    return this.x \* this.y;

};

Bar extends Foo 的意思当然就是把 Bar.prototype 的 [[Prototype]] 连接到 Foo.prototype。

所以这里

d.\_\_proto\_\_===b.prototype

b.prototype.\_\_proto\_\_===a.prototype

和红红沟通：

微任务清空后做啥

http://xgfe.github.io/2019/04/16/zhangpeng/宏任务和微任务/

当微任务处理完成并且清空的时候，事件循环会检查是否需要更新UI渲染，如果是，则会重新渲染UI视图，至此，当前事件循环结束，之后将回到第一个环节，开启新的一轮的事件循环。

便利蜂网上搜集：

websoket和ajax区别：

https://www.cnblogs.com/zhongzihao/p/7859785.html

1.本质不同

 Ajax,即异步JavaScript和XML，是一种创建交互式网页应用的网页开发技术；

 WebSocket是HTML5一种新的协议，实现了浏览器与服务器全双工通信。其本质是先通过HTTP/HTTPS协议进行握手后创建一个用于交换数据的TCP连接，服务端与客户端通过此TCP连接进行实时通信。

2.生命周期不同。

websocket建立的是长连接，在一个会话中一直保持连接；而ajax是短连接，数据发送和接受完成后就会断开连接。

3.适用范围不同

websocket一般用于前后端实时数据交互，而ajax前后端非实时数据交互。

4.发起人不同

Ajax技术需要客户端发起请求，而WebSocket服务器和客户端可以相互推送信息。

5.用法不同

ajax 以jquery为例子：

$.ajax({

type:"post",

url:"http://localhost:8080/target",

data:"state = yes",

dataType:"json",

success:funciont(data){

}

});

websocket:

var monitor = new WebSocket("ws://"+ip+path)

onOpen()、onMessage()、onClose()

Promise

构造函数传入的参数是什么类型？（我回答的是匿名函数）那么，传入的该函数是会立刻执行的吗？

new Promise( function(resolve, reject) {...} /\* executor \*/ );

**executor**

executor是带有 resolve 和 reject 两个参数的函数 。**Promise**构造函数执行时立即调用**executor** 函数， resolve 和 reject 两个函数作为参数传递给executor

猿辅导面试：

elementui this.$message如何实现

<https://blog.csdn.net/yanggreater/article/details/83278641>

import Message from './components/UI/message/message.js'

Vue.prototype.$message = Message;

message.js:

import Vue from 'vue'

const NoticeConstructor = Vue.extend(require('./message.vue').default) // 直接将Vue组件作为Vue.extend的参数,在这里注意，高版本的vue-loader要添加.default

let nId = 1

const Message = (options) => {

let id = 'notice-' + nId++;

options = options || {};

if (typeof options === 'string') {//如果只传入字符串，将其设置为显示的信息

options = {

message: options//这里的message就是message.vue中data中的message

};

}

const NoticeInstance = new NoticeConstructor({// 实例化一个带有content内容的Notice

data:options//在这里将你传过来的参数匹配到message.vue组件的data

})

//console.log(NoticeInstance);

NoticeInstance.id = id;

NoticeInstance.vm = NoticeInstance.$mount(); // 挂载但是并未插入dom，是一个完整的Vue实例

NoticeInstance.vm.visible = true //这里修改message.vue数据中的visible,这样message组件就显示出来

//console.log(NoticeInstance.vm);

NoticeInstance.dom = NoticeInstance.vm.$el //获取到本实例的dom元素

document.body.appendChild(NoticeInstance.dom) // 将dom插入body

NoticeInstance.dom.style.zIndex = nId + 1001 // 后插入的Notice组件z-index加一，保证能盖在之前的上面

return NoticeInstance.vm

};

['success', 'warning', 'info', 'error'].forEach(type => {//同element-ui一样，在给Message绑定四个方法，直接调用

Message[type] = options => {

if (typeof options === 'string') {

options = {

message: options

};

}

options.type = type;

return Message(options);

};

});

export default Message;

Message.vue:

<template>

<transition name="message-fade">

<div v-if="visible" :class="wrapClasses">

<img class="message\_img" :src="typeImg"/>

<span class="">

{{message}}

</span>

</div>

</transition>

</template>

<script>

const prefixCls = 'message'

export default {

name: 'message',

data () {

return {

visible: false,

type:'info',

message: '',

duration: 3000

}

},

computed:{

typeImg(){

return require(`./assets/${this.type}.svg`);

},

wrapClasses(){

return[

`${prefixCls}`,

`${prefixCls}-${this.type}`

]

}

},

methods: {

setTimer() {

setTimeout(() => {

this.close() // 3000ms之后调用关闭方法

}, this.duration)

},

close() {

this.visible = false

setTimeout(() => {

this.$destroy(true)

this.$el.parentNode.removeChild(this.$el) // 从DOM里将这个组件移除

}, 500)

}

},

mounted() {

this.setTimer() // 挂载的时候就开始计时，3000ms后消失

}

}

</script>

箭头函数与普通函数区别

箭头函数表达式的语法比[函数表达式](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/function)更简洁，并且没有自己的[this](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/this)，[arguments](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Functions/arguments)，[super](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/super)或 [new.target](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/new.target)。这些函数表达式更适用于那些本来需要匿名函数的地方，并且它们不能用作构造函数。

new一个箭头函数会报错：

VM356:1 Uncaught TypeError: a is not a constructor

at <anonymous>:1:7

**new.target**属性允许你检测函数或构造方法是否是通过[new](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/new)运算符被调用的。在通过[new](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/new)运算符被初始化的函数或构造方法中，new.target返回一个指向构造方法或函数的引用。在普通的函数调用中，new.target 的值是[undefined](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/undefined)。

Vue.use

Vue.use( plugin )

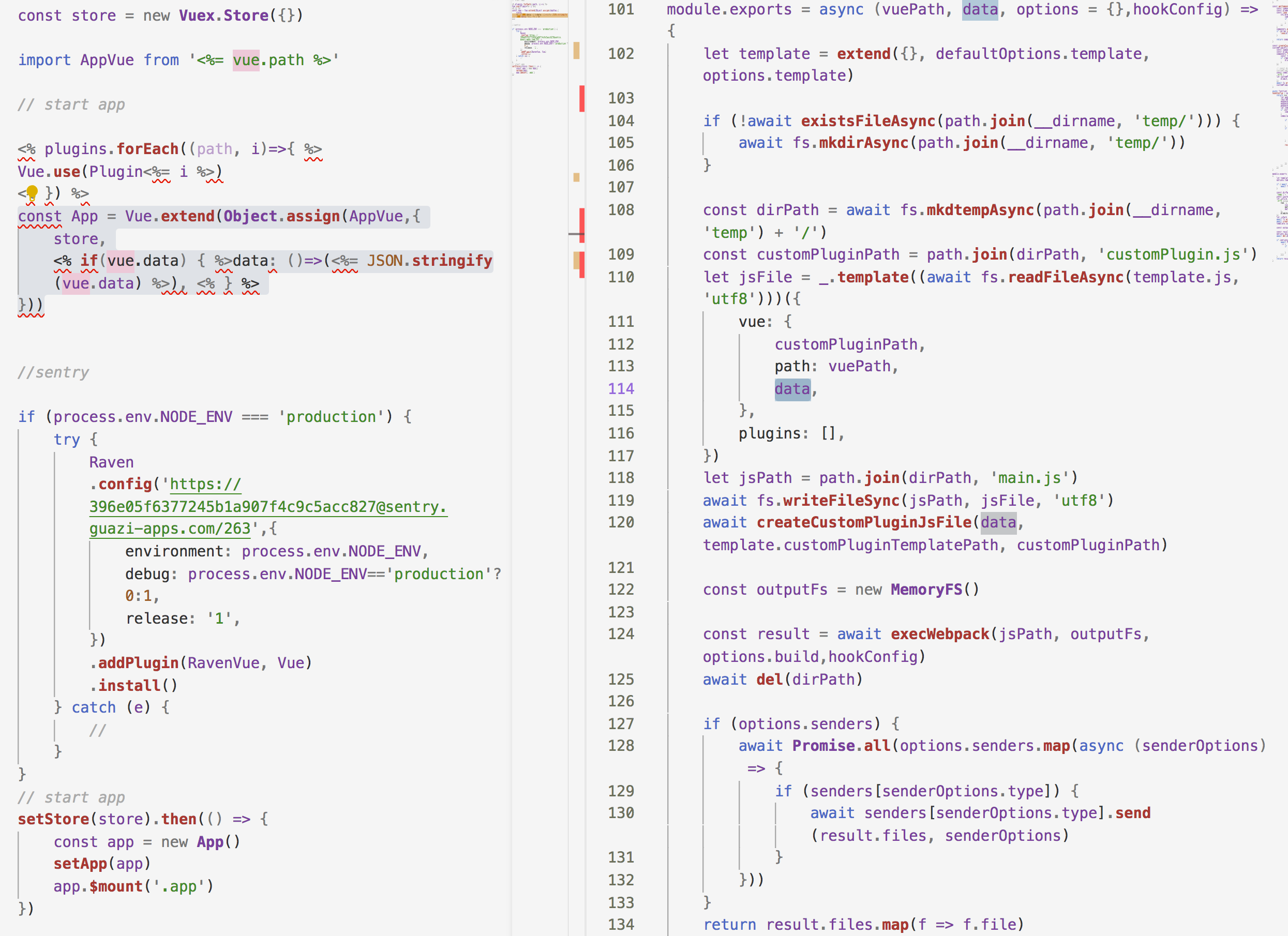
• 参数：

◦ {Object | Function} plugin

**•** 用法：  
安装 Vue.js 插件。如果插件是一个对象，必须提供 install 方法。如果插件是一个函数，它会被作为 install 方法。install 方法调用时，会将 Vue 作为参数传入。  
该方法需要在调用 new Vue() 之前被调用。  
当 install 方法被同一个插件多次调用，插件将只会被安装一次。

json如何生成html文件

分别为：\_.template解析的template文件main.js.ejs distributer.js

Main.js.ejs里AppVue的来源文件：app.vue

\_.template方法处理模板文件main.js.ejs时，往app.vue导出的对象里合并了store与 data 选项，模板中注入的vue对象的data属性就是{page,customPluginMap}，所以解析后 page组件用的page 对象就是发布方法里面传入的pageObj

单链表拆分奇偶链表

https://blog.csdn.net/romeo12334/article/details/84100112

/\*\*

\* Definition for singly-linked list.

\* function ListNode(val) {

\* this.val = val;

\* this.next = null;

\* }

\*/

/\*\*

\* @param {ListNode} head

\* @return {ListNode}

\*/

var oddEvenList = function(head) {

if (!head) return head;

if (!head.next) return head;

let p = head,

q = head.next;

let phead = p,

qhead = q;

let m = q.next;

let isOdd = true;

while (m !== null) {

if (isOdd) {

p.next = m;

p = m;

} else {

q.next = m;

q = m;

}

m = m.next

isOdd = !isOdd

}

if (!isOdd) {

q.next = null; //????

}

p.next = qhead;

return phead;

};

for循环里key设置成index会如何

当 Vue 正在更新使用 v-for 渲染的元素列表时，它默认使用“就地更新”的策略。如果数据项的顺序被改变，Vue 将不会移动 DOM 元素来匹配数据项的顺序，而是就地更新每个元素，并且确保它们在每个索引位置正确渲染。这个类似 Vue 1.x 的 track-by="$index"。

这个默认的模式是高效的，但是只适用于不依赖子组件状态或临时 DOM 状态 (例如：表单输入值) 的列表渲染输出。

为了给 Vue 一个提示，以便它能跟踪每个节点的身份，从而重用和重新排序现有元素，你需要为每项提供一个唯一 key 属性。

key 的特殊属性主要用在 Vue 的虚拟 DOM 算法，在新旧 nodes 对比时辨识 VNodes。如果不使用 key，Vue 会使用一种最大限度减少动态元素并且尽可能的尝试修复/再利用相同类型元素的算法。使用 key，它会基于 key 的变化重新排列元素顺序，并且会移除 key 不存在的元素。

猜想：

key设置成index的话，元素的key可能会变，这样会导致元素update，很可能就多了很多没有必要的工作，明明只需要新增一个元素并重排的事情，变成需要重新渲染很多元素

探探科技：

合并两有序数组，时间o(n)

<https://blog.csdn.net/u012194956/article/details/79388856>

function mergeArray(arr1,arr2){

var ind1=0; //标记arr1的对比元素的初始索引值

var ind2=0; //标记arr2的对比元素的初始索引值

var arr=[]; //作为输出的新数组

while(ind1<arr1.length && ind2<arr2.length){ //当arr1和arr2元素均未全部存入arr中，则从第一个元素开始进行比较，将较小的那个元素存入arr

if(arr1[ind1]<=arr2[ind2]){

arr.push(arr1.slice(ind1,ind1+1)[0]); //若arr1的对比元素小于arr2的对比元素，则将arr1的对比元素存入arr中

ind1++;

}else{

arr.push(arr2.slice(ind2,ind2+1)[0]);

ind2++;

}

}

while(ind1<arr1.length){ //当arr2的元素已全部存入arr中，则直接将arr1剩余的所有元素依次存入arr

arr.push(arr1.slice(ind1,ind1+1)[0]);

ind1++;

}

while(ind2<arr2.length){ //当arr1的元素已全部存入arr中,则直接将arr2剩余的所有元素依次存入arr

arr.push(arr2.slice(ind2,ind2+1)[0]);

ind2++;

}

return arr;

}

手写防抖细节

<https://www.cnblogs.com/momo798/p/9177767.html>

都是利用闭包保存对一个变量的引用，return函数出去

// 防抖

function debounce(fn, wait) {

var timeout = null;

return function() {

if(timeout !== null)  clearTimeout(timeout);

timeout = setTimeout(fn, wait);

}

}

// 节流throttle代码（定时器）：

var throttle = function(func, delay) {

var timer = null;

return function() {

var context = this;

var args = arguments;

if (!timer) {

timer = setTimeout(function() {

func.apply(context, args);

timer = null;

}, delay);

}

}

}

节流throttle代码（时间戳）：

var throttle = function(func, delay) {

　　var prev = Date.now();

　　return function() {

　　　　var context = this;

　　　　var args = arguments;

　　　　var now = Date.now();

　　　　if (now - prev >= delay) {

　　　　　　func.apply(context, args);

　　　　　　prev = Date.now();

　　　　}

　　}

}

猿辅导网上搜集：

动态规划

链接：https://www.nowcoder.com/discuss/95347?type=0&order=0&pos=28&page=1

给出一个实际金额，比如40

以及一个优惠券面额列表比如[30, 50, 100…],每种优惠券数量不限

要求返回能组合成实际金额的最大值，比如：实际金额40 -> 返回30；80 -> 80=30+50；110 -> 110=30+30+50

动态规划 假定给定金额x，能凑出的最大值是f(x)，则f(x) = Math.max(30+f(x-30),50+f(x-50),100+f(x-100)等等等)，若x<30则f(x)=0 这样明白了吗？可以写成递归，也可以写成循环

GC回收机制

https://www.cnblogs.com/Yoriluo/p/6861781.html

垃圾回收机制，简称GC（garbage collection），会定期（周期性）地回收那些不再使用的变量，然后释放其内存。

而内存占用的情况有很多：

1、变量

2、字面量对象声明：var obj = {}；

3、数组声明 ： var arr = []；

4、函数声明： function fun(){};

上述情况都会占用内存。

通常的解决方法：标记清除；引用计数。

**1.**标记清除：盒子原理，当变量要进入执行环境，则将该变量标记为“进入环境”，该标记表示不能清除回收该变量，当变量离开执行环境之后，则标记为“离开环境”，进行回收，释放内存。

**2.**引用计数：值占用数，当该变量作为值被赋其他变量之后，则引用计数+1，当其他变量改变被其他赋值之后，则该变量引用计数-1，当引用计数为0的时候，进行回收清除。（不常用）

内存管理优化方法：

例如setTimeout（function(){},1000）,每次执行setTimeout都会创建一个function对象，所以将function函数的功能提取出来。这样不用每次都创建对象，内存占用较少。

原型链

1. //题目

var a = {num:2};

var b = Object.create(a);

//问题,以下顺序执行,值是？

b.num

b.num++

a.num

1. b.num ==> 2

Object.create()方法创建一个新对象，使用现有的对象来提供新创建的对象的\_\_proto\_\_。

也就是b.\_\_proto\_\_ == a

所以虽然b是个空对象，但会从原型上一直找值。

b.num == b.\_\_proto\_\_.num == 2

2. b.num++ ==> 2

——《你不知道的javascript·上卷》P144

下面我们分析一下如果 foo 不直接存在于 myObject 中而是存在于原型链上层时 myObject.foo = "bar" 会出现的三种情况。

1. 如果在[[Prototype]]链上层存在名为foo的普通数据访问属性(参见第3章)并且没 有被标记为只读(writable:false)，那就会直接在 myObject 中添加一个名为 foo 的新 属性，它是屏蔽属性。

2. 如果在[[Prototype]]链上层存在foo，但是它被标记为只读(writable:false)，那么 无法修改已有属性或者在 myObject 上创建屏蔽属性。如果运行在严格模式下，代码会 抛出一个错误。否则，这条赋值语句会被忽略。总之，不会发生屏蔽。

3. 如果在[[Prototype]]链上层存在foo并且它是一个setter(参见第3章)，那就一定会 调用这个 setter。foo 不会被添加到(或者说屏蔽于)myObject，也不会重新定义 foo 这 个 setter。

大多数开发者都认为如果向 [[Prototype]] 链上层已经存在的属性([[Put]])赋值，就一 定会触发屏蔽，但是如你所见，三种情况中只有一种(第一种)是这样的。

如果你希望在第二种和第三种情况下也屏蔽 foo，那就不能使用 = 操作符来赋值，而是使 用 Object.defineProperty(..)(参见第 3 章)来向 myObject 添加 foo。

——《你不知道的javascript·上卷》P161

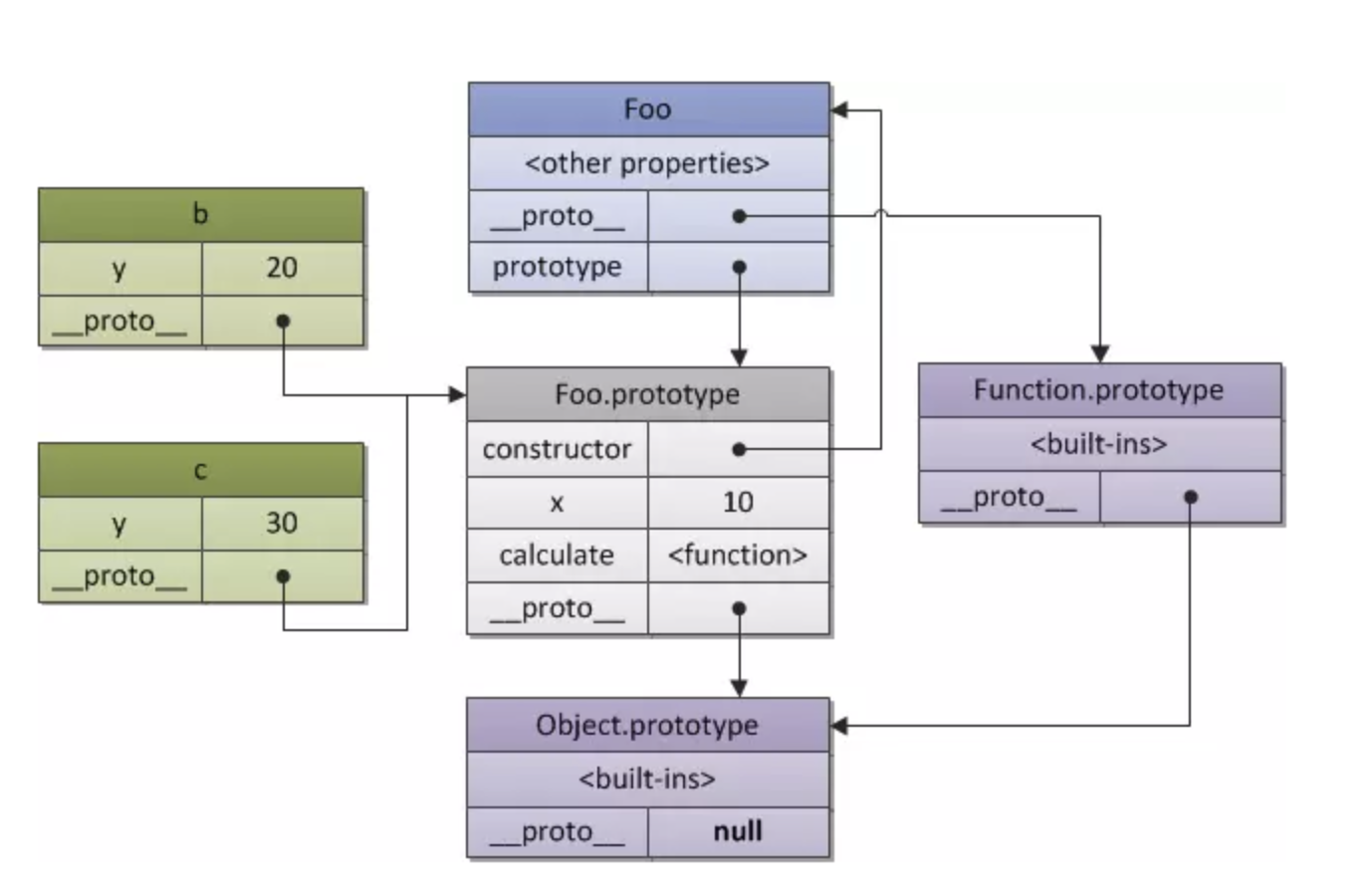
尽管 myObject.a++ 看起来应该(通过委托)查找并增加 anotherObject.a 属性，但是别忘 了 ++ 操作相当于 myObject.a = myObject.a + 1。因此 ++ 操作首先会通过 [[Prototype]] 查找属性 a 并从 anotherObject.a 获取当前属性值 2，然后给这个值加 1，接着用 [[Put]] 将值 3 赋给 myObject 中新建的屏蔽属性 a 。

b.num为2，后置**++**运算符是先用后加，这时**b.num++**返回还是**2**

3. a.num ==> 2

b.num 在执行 ++ 操作之后，这时的b为{num:3},不影响a.

2. const classA = new ClassA() 请问ClassA的prototype是什么？ ClassA.prototype. \_\_proto\_\_是什么？



开平方

开平方算法？

求根号x,精度0.01.x是大于1的正整数

Math.sqrt( x ).toFixed(2) ;

v-for中的 :key 作用

key的作用主要是为了高效的更新虚拟DOM

封装一个 instanceof

https://www.icode9.com/content-4-148601.html

**instanceof** 运算符用于检测构造函数的 prototype 属性是否出现在某个实例对象的原型链上。

检测某个实例的原型链上是否有构造函数的prototype

function instance\_of(I, C) {// I实例 C构造函数

var O = C.prototype;

I = I.\_\_proto\_\_;

while (true) {

if (I === null)

return false;

if (I === O) // 这里重点：当 O 严格等于 L 时，返回 true

return true;

I = I.\_\_proto\_\_;

}

}

网易：

从浏览器地址栏输入url到显示页面的步骤

这次面试直接把三次握手四次挥手忘记了…

连续赋值 赋值顺序从左往右 .的优先级比赋值高

<https://blog.csdn.net/yjd19970908/article/details/80671426>

var a = {n: 1};

var b = a;

a.x = a = {n: 2};

console.log(a.x); //undefined

console.log(b.x); //{n:2}

常用的请求头响应头

要背常用的

<https://blog.csdn.net/qq_30553235/article/details/79282113>

<https://blog.csdn.net/heyue_99/article/details/74689057>

HTTP请求报文由3部分组成 请求行**+**请求头**+**请求体





localstorage sessionstorage cookie 区别

<https://www.cnblogs.com/candy-xia/p/11561542.html>

**1**）存储大小

cookie：一般不超过**4K**（因为每次http请求都会携带cookie、所以cookie只适合保存很小的数据，如会话标识）

sessionStorage：**5M**或者更大

localStorage：5M或者更大

**2**）数据有效期

cookie：一般由服务器生成，可以设置失效时间；若没有设置时间，关闭浏览器cookie失效，若设置了时间，cookie就会存放在硬盘里，过期才失效

sessionStorage：仅在当前浏览器窗口关闭之前有效，关闭页面或者浏览器会被清除

localStorage：永久有效，窗口或者浏览器关闭也会一直保存，除非手动永久清除，因此用作持久数据

**3**）作用域

cookie：在所有同源窗口中都是共享的

sessionStorage：在同一个浏览器窗口是共享的（不同浏览器、同一个页面也是不共享的）

localStorage：在所有同源窗口中都是共享的

**4**）通信

cookie：十种携带在同源的http请求中，即使不需要，故cookie在浏览器和服务器之间来回传递；如果使用cookie保存过多数据会造成性能问题

sessionStorage：仅在客户端（即浏览器）中保存，不参与和服务器的通信；不会自动把数据发送给服务器，仅在本地保存

localStorage：仅在客户端（即浏览器）中保存，不参与和服务器的通信；不会自动把数据发送给服务器，仅在本地保存

**5**）易用性

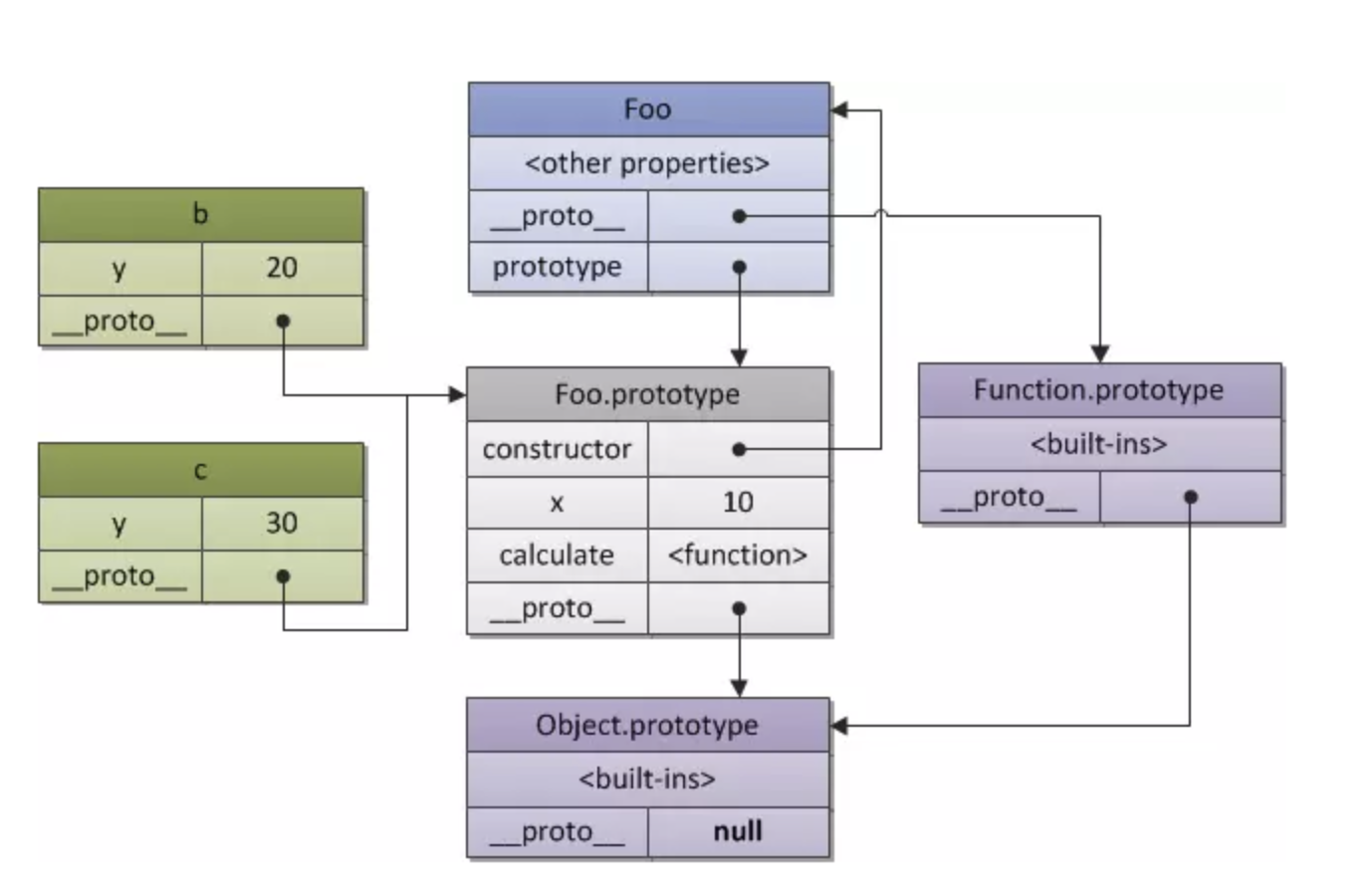
cookie：需要自己进行封装，原生的cookie接口不够友好

sessionStorage：原生接口可以接受，可以封装来对Object和Array有更好的支持

localStorage：原生接口可以接受，可以封装来对Object和Array有更好的支持

function a(){} 在原型链上如何和 Function联系起来

https://www.jianshu.com/p/08c07a953fa0

a的\_\_proto\_\_指向Function.prototype

typeof null === object

手写一个vue组件

面试时都记不得组件的写法了，平时都是复制粘贴…

// 定义一个名为 button-counter 的新组件

Vue.component('button-counter', {

data: function () {

return {

count: 0

}

},

template: '<button v-on:click="count++">You clicked me {{ count }} times.</button>'

})

单文件：

<template>

</template>

<script>

export default {

data(){},

methods:{

},

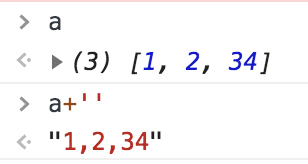
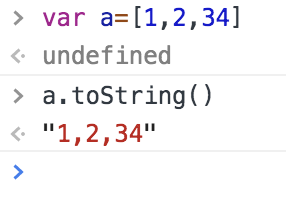
…

}

</script>

新东方面试

数组转字符串的结果

每个对象都有一个 toString() 方法，当该对象被表示为一个文本值时，或者一个对象以预期的字符串方式引用时自动调用。

js下载文件

https://www.cnblogs.com/haojf/p/9407183.html

1.HTML5原生的download属性

如果希望在前端直接出发某些资源的下载，最方便快捷的方法就是使用HTML5原生的download属性， 例如：

<a href="large.jpg" download=“fileName”>下载</a>

但显然，如果纯粹利用HTML属性来实现文件的下载（而不是浏览器打开或浏览），对于动态内容，就无能为力。

2.借助HTML5 Blob实现文本信息文件下载

原理其实很简单，我们可以将文本或者JS字符串借助Blob转换成二进制，然后，作为a元素的href属性，配合download属性，实现下载。

代码也比较简单，如下示例（兼容Chrome和FireFox）：

**URL.createObjectURL()** 静态方法会创建一个 [DOMString](https://developer.mozilla.org/zh-CN/docs/Web/API/DOMString)，其中包含一个表示参数中给出的对象的URL。这个 URL 的生命周期和创建它的窗口中的 [document](https://developer.mozilla.org/zh-CN/docs/Web/API/Document) 绑定。这个新的URL 对象表示指定的 [File](https://developer.mozilla.org/zh-CN/docs/Web/API/File) 对象或 [Blob](https://developer.mozilla.org/zh-CN/docs/Web/API/Blob) 对象。

```

var funDownload = function(content, filename){

// 创建隐藏的可下载链接

var eleLink = document.createElement('a');

eleLink.download = filename;

eleLink.style.display = 'none';

// 字符内容转变成blob地址

var blob = new Blob([content]);

eleLink.href = URL.createObjectURL(blob);

// 触发点击

document.body.appendChild(eleLink);

eleLink.click();

// 然后移除

document.body.removeChild(eleLink);

};

```

3.借助Base64 context.toDataURL('image/jpeg')

```

var funDownload = function(domImg, filename){

// 创建隐藏的可下载链接

var eleLink = document.createElement('a');

eleLink.download = filename;

eleLink.style.display = 'none';

// 图片base64地址

var canvas = document.createElement('canvas');

var context = canvas.getContext('2d');

var width = domImg.natureWidth;

var height = domImg.natureHeight;

context.drawImage(domImg, 0, 0);

// 如果是PNG图片，则context.toDataURL('image/png');

eleLink.href = context.toDataURL('image/jpeg');

// 触发点击

document.body.appendChild(eleLink);

eleLink.click();

// 移除

document.body.removeChild(eleLink);

};

```

js判断奇偶

％

css大括符

<https://www.cnblogs.com/arealy/p/7736856.html>

<div class="quote">

    <div></div>

</div>

.quote {

        position: relative;

        width: 100%;

        height: 40px;

    }

    .quote::before,

    .quote::after,

    .quote ::before,

    .quote ::after {

        content: '';

        display: block;

        position: absolute;

        width: calc(50% - 20px);

        height: 20px;

        border-style: solid;

        border-color: #000;

        border-width: 0;

    }

    .quote ::before,

    .quote ::after {

        top: 0;

        border-bottom-width: 1px;

    }

    .quote::before,

    .quote::after {

        top: 20px;

        border-top-width: 1px;

    }

    .quote ::before {

        left: 0;

        border-bottom-left-radius: 20px;

    }

    .quote ::after {

        right: 0;

        border-bottom-right-radius: 20px;

    }

    .quote::before {

        left: 20px;

        border-top-right-radius: 20px;

    }

    .quote::after {

        right: 20px;

        border-top-left-radius: 20px;

    }

监听对象属性变化

<https://www.cnblogs.com/xiaoyuchen/p/10547696.html>

**Object.defineProperty** 不足

• 无法监听数组的变化： 数组的这些方法是无法触发**set**的**:push, pop, shift, unshift,splice, sort, reverse.**，vue中能监听是因为对这些方法进行了重写

• 只能监听属性，而不是监听对象本身，需要对对象的每个属性进行遍历。对于原本不在对象中的属性难以监听。

**Proxy**优点

• 可以监听数组变化

• 监听的是对象本身

• 有13种拦截方法

<https://www.jianshu.com/p/fce3a6a9f920>

let validator = {

set: function(obj, prop, value) {

if (prop === 'age') {

if (!Number.isInteger(value)) {

throw new TypeError('The age is not an integer');

}

if (value > 200) {

throw new RangeError('The age seems invalid');

}

}

// The default behavior to store the value

obj[prop] = value;

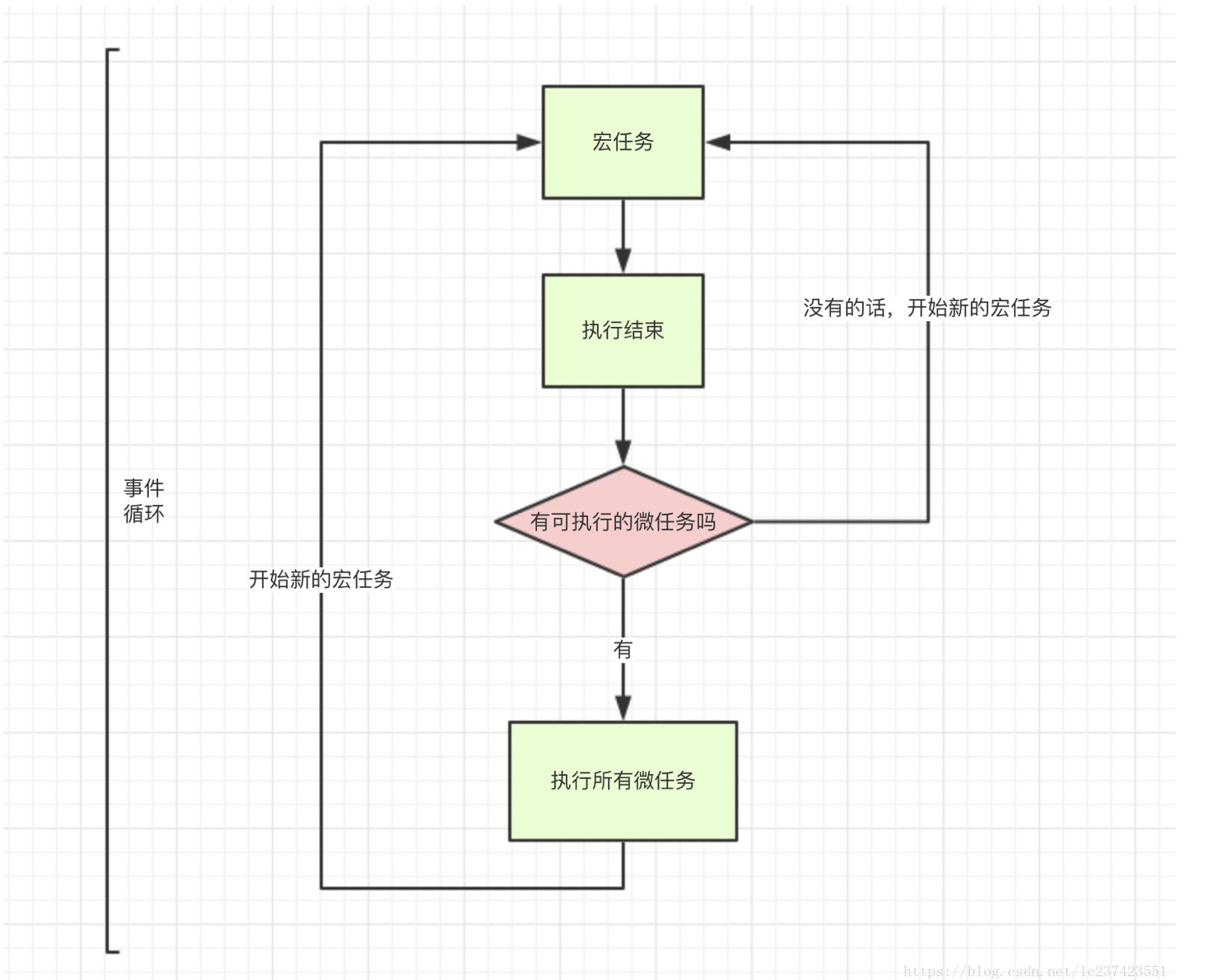
}

};

let person = new Proxy({}, validator);

宏任务 微任务

<https://blog.csdn.net/lc237423551/article/details/79902106>

一个宏任务结束后会把所有可执行的微任务全部执行掉****

https://segmentfault.com/a/1190000018951319

**•** 消息队列：消息队列是一个先进先出的队列，它里面存放着各种消息。

• 事件循环：事件循环是指主线程重复从消息队列中取消息、执行的过程。(浏览器至少有一个事件循环，一个事件循环至少有一个任务队列（macrotask）)

• 微任务：

◦ JavaScript 引擎发起的任务 - JS 引擎级别

◦ **promise**回调，**MutationObserver**，**process.nextTick**，**Object.observe**

• 宏任务

◦ 宿主发起的任务，每次的一段js代码执行过程，其实都是一个宏观任务 - 宿主级别

◦ 整体的js代码，事件回调，XHR回调，定时器（setTimeout/setInterval/setImmediate），IO操作，UI render

• 宏任务和微任务关系：每个macro宏任务会维护一个micro微任务列

判断符号闭合

<http://www.04007.cn/article/664.html>

function checkClose($str){

#定义一个关系

$relate = array('('=>')', '['=>']', '{'=>'}' );

#遍历字符串将对应符号插入栈

$quene = array();

for($i=0; $i< strlen($str); $i++){

#如果是出现了([{类字符，则直接存入栈等待配对

if(isset($relate[$str{$i}])){

array\_push($quene, $str{$i});

}elseif( in\_array($str{$i}, $relate)){

#如果出现了右边的)]}类字符，则必须与栈中最后一个元素相匹配，否则即为不是闭合的

if($str{$i} != $relate[end($quene)]){

return false;

}else{

#匹配的话则消除这对符号

array\_pop($quene);

}

}else{

#非上述6个符号的内容不用考虑

continue;

}

}

#结尾只需要返回对$quene的判断

return (bool)!$quene;

}

如果是架构师如何推动新项目

node事件队列

每次事件循环都包含了6个阶段

◦ **timers** 阶段：这个阶段执行timer（setTimeout、setInterval）的回调

◦ **I/O callbacks** 阶段：执行一些系统调用错误，比如网络通信的错误回调

◦ **idle, prepare** 阶段：仅node内部使用

◦ **poll** 阶段：获取新的I/O事件, 适当的条件下node将阻塞在这里

◦ **check** 阶段：执行 setImmediate() 的回调

◦ **close callbacks** 阶段：执行 socket 的 close 事件回调

动态组件vue

<component v-bind:is="currentTabComponent"></component>

深拷贝循环引用处理

<https://blog.csdn.net/qq_41846861/article/details/102296436>

// 木易杨

function cloneDeep3(source, uniqueList) {

if (!isObject(source)) return source;

if (!uniqueList) uniqueList = []; // 新增代码，初始化数组

var target = Array.isArray(source) ? [] : {};

// ============= 新增代码

// 数据已经存在，返回保存的数据

var uniqueData = find(uniqueList, source);

if (uniqueData) {

return uniqueData.target;

};

// 数据不存在，保存源数据，以及对应的引用

uniqueList.push({

source: source,

target: target

});

// =============

for(var key in source) {

if (Object.prototype.hasOwnProperty.call(source, key)) {

if (isObject(source[key])) {

target[key] = cloneDeep3(source[key], uniqueList); // 新增代码，传入数组

} else {

target[key] = source[key];

}

}

}

return target;

}

// 新增方法，用于查找

function find(arr, item) {

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

if (arr[i].source === item) {

return arr[i];

}

}

return null;

}

// 用上面测试用例已测试通过

静态作用域

//xdf

var name='xdf'

    function b(){

        var name='xdf2'

        a()

    }

    function a(){

        alert(name)

    }

    b()

//xdf2

var name='xdf'

    function b(){

        name='xdf2'

        a()

    }

    function a(){

        alert(name)

    }

    b()

红红360图片

头条

cdn拓扑结构

内容路由器负责将用户的请求调度到适当的设备上。内容路由通常通过负载均衡系统来实现，动态均衡各个内容缓存站点的载荷分配，为用户的请求选择最佳的访问站点，同时提高网站的可用性。内容路由器可根据多种因素制定路由，包括站点与用户的临近度、内容的可用性、网络负载、设备状况等。负载均衡系统是整个CDN的核心。负载均衡的准确性和效率直接决定了整个CDN的效率和性能。 [3]

代码：限制promise并发

<https://blog.csdn.net/tonghonglei/article/details/88054116>

//省略代码

function limitLoad(urls, handler, limit) {

// 对数组做一个拷贝

const sequence = [].concat(urls)

let promises = [];

//并发请求到最大数

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

// 这里返回的 index 是任务在 promises 的脚标，

//用于在 Promise.race 之后找到完成的任务脚标

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

return index

});

});

(async function loop() {

let p = Promise.race(promises);

for (let i = limit; i < sequence.length; i++) {

p = p.then((res) => {

promises[res] = handler(sequence[i]).then(() => {

return res

});

return Promise.race(promises)

})

}

})()

}

limitLoad(urls, loadImg, 3)

Xss预防

[https://baike.baidu.com/item/XSS攻击/954065?fr=aladdin#6](https://baike.baidu.com/item/XSS%E6%94%BB%E5%87%BB/954065?fr=aladdin#6)

（1）不信任用户提交的任何内容，对所有用户提交内容进行可靠的输入验证，包括对URL、查询关键字、HTTP头、REFER、POST数据等，仅接受指定长度范围内、采用适当格式、采用所预期的字符的内容提交，对其他的一律过滤。尽量采用POST而非GET提交表单；对“<”，“>”，“；”，“””等字符做过滤；任何内容输出到页面之前都必须加以en-code，避免不小心把htmltag显示出来。

（2）实现Session 标记（session tokens）、CAPTCHA（验证码）系统或者HTTP引用头检查，以防功能被第三方网站所执行，对于用户提交信息的中的img等link，检查是否有重定向回本站、不是真的图片等可疑操作。

（3）cookie 防盗。避免直接在cookie中泄露用户隐私，例如email、密码，等等；通过使cookie和系统IP绑定来降低cookie泄露后的危险。这样攻击者得到的cookie没有实际价值，很难拿来直接进行重放攻击。

（4）确认接收的内容被妥善地规范化，仅包含最小的、安全的Tag（没有JavaScript），去掉任何对远程内容的引用（尤其是样式表和JavaScript），使用HTTPonly的cookie。

csrf预防

跨站请求攻击，简单地说，是攻击者通过一些技术手段欺骗用户的浏览器去访问一个自己曾经认证过的网站并运行一些操作（如发邮件，发消息，甚至财产操作如转账和购买商品）。由于浏览器曾经认证过，所以被访问的网站会认为是真正的用户操作而去运行。这利用了web中用户身份验证的一个漏洞：简单的身份验证只能保证请求发自某个用户的浏览器，却不能保证请求本身是用户自愿发出的。

措施：

检查**Referer**字段

因其完全依赖浏览器发送正确的Referer字段。虽然http协议对此字段的内容有明确的规定，但并无法保证来访的浏览器的具体实现，亦无法保证浏览器没有安全漏洞影响到此字段。并且也存在攻击者攻击某些浏览器，篡改其Referer字段的可能。

添加校验**token**

async defer

1 <script src="script.js"></script>  
没有 defer 或 async，浏览器会立即加载并执行指定的脚本，“立即”指的是在渲染该 script 标签之下的文档元素之前，也就是说不等待后续载入的文档元素，读到就加载并执行。

2 <script async src="script.js"></script>  
有 async，加载和渲染后续文档元素的过程将和 script.js 的加载与执行并行进行（异步）。

3 <script defer src="myscript.js"></script>  
有 defer，加载后续文档元素的过程将和 script.js 的加载并行进行（异步），但是 script.js 的执行要在所有元素解析完成之后

如果async属性存在，脚本将异步执行，只要它是可用的，如果async属性不存在，而defer属性存在，脚本将会在页面完成解析后执行，如果都不存在，那么脚本会在useragent解析页面之前被取出并立刻执行

computed watch的区别

<https://segmentfault.com/a/1190000012948175?utm_source=tag-newest>

data(){

return{

'first':{

second:0

}

}

},

computed:{

secondChange(){

return this.first.second

}

},

watch:{

secondChange(){

console.log('second属性值变化了')

}

},

computed特性：

1.是计算值，

2.应用：就是简化tempalte里面{{}}计算和处理props或$emit的传值

3.具有缓存性，页面重新渲染值不变化,计算属性会立即返回之前的计算结果，而不必再次执行函数

watch特性：

1.是观察的动作，

2.应用：监听props，$emit或本组件的值执行异步操作

3.无缓存性，页面重新渲染时值不变化也会执行

node环境里的 setimmediate与settimeout区别

<https://www.cnblogs.com/jiasm/p/9482443.html>

就单从API层面上来理解，Node新增了两个方法可以用来使用：微任务的process.nextTick以及宏任务的setImmediate。

**setImmediate**与**setTimeout**的区别

在官方文档中的定义，setImmediate为一次Event Loop执行完毕后调用。

setTimeout则是通过计算一个延迟时间后进行执行。

但是同时还提到了如果在主进程中直接执行这两个操作，很难保证哪个会先触发

redux vuex中间件

避免打包到bundle里怎么设置soursemap

hidden-source-map

webpack每个块的大小设置多少比较合理

200k

webpack速度优化

https://blog.csdn.net/a460550542/article/details/82661520

1.减小文件搜索范围

配置 resolve.modules 告诉 webpack 解析模块时应该搜索的目录

设置 test & include & exclude

2.增强代码代码压缩工具

Webpack 默认提供的 UglifyJS 插件，由于采用单线程压缩，速度颇慢 ；推荐采用 [webpack-parallel-uglify-plugin](https://www.npmjs.com/package/webpack-parallel-uglify-plugin) 插件，她可以并行运行 UglifyJS 插件，更加充分而合理的使用 CPU 资源，这可以大大减少的构建时间；当然，该插件应用于生产环境而非开发环境

3.用 Happypack 来加速代码构建

Happypack 的处理思路是：将原有的 webpack 对 loader 的执行过程，从单一进程的形式扩展多进程模式，从而加速代码构建；原本的流程保持不变，这样可以在不修改原有配置的基础上，来完成对编译过程的优化

DllPlugin

Dll

Use the DllPlugin to move code that is changed less often into a separate compilation. This will improve the application's compilation speed, although it does increase complexity of the build process.

移动端1px几种方案

https://www.jianshu.com/p/31f8907637a6

1、 媒体查询利用设备像素比缩放，设置小数像素；

优点：简单，好理解

缺点：兼容性差，目前之余IOS8+才支持，在IOS7及其以下、安卓系统都是显示0px。

2.transform: scale(0.5) 方案 - 推荐: 很灵活

设置height: 1px，根据媒体查询结合transform缩放为相应尺寸。

http2

HTTP2.0并没有改变之前HTTP的语义,也就是说高层的Api并没有改变,它是在底层通过二进制frame来改变性能的.

性能提升的核心在于二进制帧层.它指HTTP消息在客户端和服务端如何封装和传输.

Headers frame data frame

请求和响应的多路复用

在HTTP/2中,新的二进制帧层,解除了这个限制.使得所有的请求和响应多路复用.通过允许客户端和服务端把HTTP消息分解成独立的帧,交错传输,然后在另一端组装.

服务端推送

头部压缩

每个HTTP传输都包含一组描述传输资源及其属性的标题。在HTTP / 1.x中，此元数据始终以纯文本形式发送，并且每次传输的开销都会在任何位置增加500-800字节，如果使用HTTP Cookie，则会增加数千字节。为了减少这种开销并提高性能，HTTP / 2使用两种简单但强大的技术使用HPACK压缩格式(要了解这个算法,可以参考这篇文章[https://imququ.com/post/header-compression-in-http2.html](https://link.jianshu.com/?t=https%3A%2F%2Fimququ.com%2Fpost%2Fheader-compression-in-http2.html))来压缩请求和响应头元数据：

vue Extend mixin区别

mixins 选项接受一个混入对象的数组。这些混入实例对象可以像正常的实例对象一样包含选项，他们将在 Vue.extend() 里最终选择使用相同的选项合并逻辑合并。举例：如果你的混入包含一个钩子而创建组件本身也有一个，两个函数将被调用。

Mixin 钩子按照传入顺序依次调用，并在调用组件自身的钩子之前被调用。

• Vue.extend用于创建vue实例

• mixins可以混入多个mixin，extends只能继承一个

Vue 对数组方法的处理

Vue内部实现了一组观察数组的变异方法，例如：push()，pop()，shift()等。

Object.defineProperty只能把对象属性改为getter/setter，而对于数组的方法就无能为力了，其内部巧妙的使用了数组的属性来实现了数据的双向绑定

移动端适配几种方案

flexible.js的原理

https://github.com/amfe/article/issues/17

**flexible**的实质

flexible实际上就是能过JS来动态改写meta标签，代码类似这样：

var metaEl = doc.createElement('meta');

var scale = isRetina ? 0.5:1;

metaEl.setAttribute('name', 'viewport');

metaEl.setAttribute('content', 'initial-scale=' + scale + ', maximum-scale=' + scale + ', minimum-scale=' + scale + ', user-scalable=no');

if (docEl.firstElementChild) {

document.documentElement.firstElementChild.appendChild(metaEl);

} else {

var wrap = doc.createElement('div');

wrap.appendChild(metaEl);

documen.write(wrap.innerHTML);

}

事实上他做了这几样事情：

• 动态改写<meta>标签

• 给<html>元素添加data-dpr属性，并且动态改写data-dpr的值

• 给<html>元素添加font-size属性，并且动态改写font-size的

前端模块规范有哪些，之间的差异 3分

https://www.jianshu.com/p/480496c3c671

CommonJS

Node.js遵循的规范

1.代码运行在模块作用域，不会污染全局

2.加载模块顺序按照词法解析的顺序加载

3.加载模块是同步的

4.单例加载：也就是加载的模块会缓存起来，再次使用时，会直接用运行结果，不会再加载（除非手动清除）

5.加载模块得到的是结果的拷贝作者：evilrescuer

AMD

RequireJS遵循的规范

AMD用define()定义模块，用require()加载模块。

1.加载模块推崇依赖前置、提前执行

2.加载模块是异步

3. CMD

sea.js遵循的规范

1.加载模块推崇依赖就近、延迟执行

2.加载模块也是异步的

ES6 模块

ES6 模块的设计思想是尽量的静态化，使得编译时就能确定模块的依赖关系

1.编译时确定模块的依赖关系

2.加载模块存储的是值的引用，所以全局只有一份

3.加载模块也是异步的

最开始，JavaScript没有模块。后来：

有CommoneJS规范，最典型的实践就是Node.js，主要使用在服务器端，同步加载模块；

有AMD，最典型的实践就是RequireJS，依赖前置，主要使用在浏览器端，异步加载模块。

有CMD，最典型的实践就是sea.js，依赖就近，主要使用在浏览器端，异步加载模块。

有ES6的Module，在语言层面定义了模块，通过export和import，吸收了CommoneJS和AMD两者的优点，兼容两标准的规范

2、class与es5 function本质上的区别 5分??

3、Promise的链式调用 2分

4、async/await用法，异常捕获的方式 3分

5、浏览器的事件循环模型与node中的事件循环模型的区别 2分

https://blog.csdn.net/Fundebug/article/details/86487117

浏览器和 Node 环境下，microtask 任务队列的执行时机不同

Node 端，microtask 在事件循环的各个阶段之间执行

浏览器端，microtask 在事件循环的 macrotask 执行完之后执行

6、vue的响应式实现原理？3分

7、什么时候使用Vuex和Redux? 4

8、webpack代码分割方式和分割策略？5

有三种常用的代码分离方法：

• 入口起点：使用 [entry](https://www.webpackjs.com/configuration/entry-context) 配置手动地分离代码。

• 防止重复：使用 [CommonsChunkPlugin](https://www.webpackjs.com/plugins/commons-chunk-plugin) 去重和分离 chunk。

• 动态导入：通过模块的内联函数调用来分离代码。

9、webpack treeshaking? 1

*tree shaking* 是一个术语，通常用于描述移除 JavaScript 上下文中的未引用代码(dead-code)。它依赖于 ES2015 模块系统中的[静态结构特性](http://exploringjs.com/es6/ch_modules.html#static-module-structure)，例如 [import](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import) 和 [export](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export)。

「副作用」的定义是，在导入时会执行特殊行为的代码，而不是仅仅暴露一个 *export* 或多个 *export*。举例说明，例如 *polyfill*，它影响全局作用域，并且通常不提供 *export*。

如果你的代码确实有一些副作用，那么可以改为提供一个数组

为了学会使用 *tree shaking*，你必须……

• 使用 ES2015 模块语法（即 import 和 export）。

• 在项目 package.json 文件中，添加一个 "sideEffects" 入口。

• 引入一个能够删除未引用代码(dead code)的压缩工具(minifier)（例如 UglifyJSPlugin）。

10、css模块化？2

A **CSS Module** is a CSS file in which all class names and animation names are scoped locally by default. All URLs (url(...)) and @imports are in module request format (./xxx and ../xxx means relative, xxx and xxx/yyy means in modules folder, i. e. in node\_modules).

When importing the **CSS Module** from a JS Module, it exports an object with all mappings from local names to global names.

vue css module

首先，CSS Modules 必须通过向 css-loader 传入 modules: true 来开启：

然后在你的 <style> 上添加 module 特性：

这个 module 特性指引 Vue Loader 作为名为 $style 的计算属性，向组件注入 CSS Modules 局部对象。然后你就可以在模板中通过一个动态类绑定来使用它了：

你也可以通过 JavaScript 访问到它：

11、express与KOA的区别？2

https://www.jianshu.com/p/17e82ac53452

koa是一个比express更精简，使用node新特性的中间件框架，相比之前express就是一个庞大的框架

如果你喜欢diy，很潮，可以考虑koa，它有足够的扩展和中间件，而且自己写很简单

如果你想简单点，找一个框架啥都有，那么先express

12、ssr？

https://ssr.vuejs.org/zh/#为什么使用服务器端渲染-ssr-？

it is also possible to render the same components into HTML strings on the server, send them directly to the browser, and finally "hydrate" the static markup into a fully interactive app on the client.

与传统 SPA (单页应用程序 (Single-Page Application)) 相比，服务器端渲染 (SSR) 的优势主要在于：

• 更好的 SEO，由于搜索引擎爬虫抓取工具可以直接查看完全渲染的页面。  
请注意，截至目前，Google 和 Bing 可以很好对同步 JavaScript 应用程序进行索引。在这里，同步是关键。如果你的应用程序初始展示 loading 菊花图，然后通过 Ajax 获取内容，抓取工具并不会等待异步完成后再行抓取页面内容。也就是说，如果 SEO 对你的站点至关重要，而你的页面又是异步获取内容，则你可能需要服务器端渲染(SSR)解决此问题。

• 更快的内容到达时间 (time-to-content)，特别是对于缓慢的网络情况或运行缓慢的设备。无需等待所有的 JavaScript 都完成下载并执行，才显示服务器渲染的标记，所以你的用户将会更快速地看到完整渲染的页面。通常可以产生更好的用户体验，并且对于那些「内容到达时间(time-to-content) 与转化率直接相关」的应用程序而言，服务器端渲染 (SSR) 至关重要。

使用服务器端渲染 (SSR) 时还需要有一些权衡之处：

• 开发条件所限。浏览器特定的代码，只能在某些生命周期钩子函数 (lifecycle hook) 中使用；一些外部扩展库 (external library) 可能需要特殊处理，才能在服务器渲染应用程序中运行。

• 涉及构建设置和部署的更多要求。与可以部署在任何静态文件服务器上的完全静态单页面应用程序 (SPA) 不同，服务器渲染应用程序，需要处于 Node.js server 运行环境。

• 更多的服务器端负载。在 Node.js 中渲染完整的应用程序，显然会比仅仅提供静态文件的 server 更加大量占用 CPU 资源 (CPU-intensive - CPU 密集)，因此如果你预料在高流量环境 (high traffic) 下使用，请准备相应的服务器负载，并明智地采用缓存策略。

13、跨域与cookie？1

https://www.cnblogs.com/nuccch/p/7875189.html

需要从2个方面解决：

1.服务器端使用CROS协议解决跨域访问数据问题时，需要设置响应消息头Access-Control-Allow-Credentials值为“true”。

同时，还需要设置响应消息头Access-Control-Allow-Origin值为指定单一域名（注：不能为通配符“\*”）。

2.客户端需要设置Ajax请求属性withCredentials=true，让Ajax请求都带上Cookie。

var xhr = new XMLHttpRequest();

xhr.open('GET', url);

xhr.withCredentials = true; // 携带跨域cookie

xhr.send();

14、浏览器缓存策略？2

https://www.jianshu.com/p/54cc04190252

• 浏览器每次发起请求，都会先在浏览器缓存中查找该请求的结果以及缓存标识

• 浏览器每次拿到返回的请求结果都会将该结果和缓存标识存入浏览器缓存中

根据是否需要向服务器重新发起HTTP请求将缓存过程分为两个部分，分别是强缓存和协商缓存

强缓存：不会向服务器发送请求，直接从缓存中读取资源，在chrome控制台的Network选项中可以看到该请求返回200的状态码，并且Size显示from disk cache或from memory cache。强缓存可以通过设置两种 HTTP Header 实现：Expires 和 Cache-Control max-age。

协商缓存：协商缓存就是强制缓存失效后，浏览器携带缓存标识向服务器发起请求，由服务器根据缓存标识决定是否使用缓存的过程，主要有以下两种情况：

协商缓存生效，返回304和Not Modified

协商缓存失效，返回200和请求结果

协商缓存可以通过设置两种 HTTP Header 实现：Last-Modified/If-Modified-Since 和 ETag/If-None-Match

——————————————————————————————————

BFC

块格式化上下文（**Block Formatting Context**，**BFC**） 是Web页面的可视化CSS渲染的一部分，是块盒子的布局过程发生的区域，也是浮动元素与其他元素交互的区域。

块格式化上下文对浮动定位（参见 [float](https://developer.mozilla.org/zh-CN/docs/Web/CSS/float)）与清除浮动（参见 [clear](https://developer.mozilla.org/zh-CN/docs/Web/CSS/clear)）都很重要。浮动定位和清除浮动时只会应用于同一个BFC内的元素。浮动不会影响其它BFC中元素的布局，而清除浮动只能清除同一BFC中在它前面的元素的浮动。外边距折叠（[Margin collapsing](https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Box_Model/Mastering_margin_collapsing)）也只会发生在属于同一BFC的块级元素之间。

块级元素的[上外边距](https://developer.mozilla.org/zh-CN/docs/Web/CSS/margin-top)和[下外边距](https://developer.mozilla.org/zh-CN/docs/Web/CSS/margin-bottom)有时会合并（或折叠）为一个外边距，其大小取其中的最大者，这种行为称为外边距折叠（margin collapsing），有时也翻译为外边距合并。注意[浮动元素](https://developer.mozilla.org/zh-CN/docs/Web/CSS/float)和[绝对定位元素](https://developer.mozilla.org/zh-CN/docs/Web/CSS/position#absolute)的外边距不会折叠。

下列方式会创建块格式化上下文：

• 根元素(<html>)

• 浮动元素（元素的 [float](https://developer.mozilla.org/zh-CN/docs/Web/CSS/float) 不是 none）

• 绝对定位元素（元素的 [position](https://developer.mozilla.org/zh-CN/docs/Web/CSS/position) 为 absolute 或 fixed）

• 行内块元素（元素的 [display](https://developer.mozilla.org/zh-CN/docs/Web/CSS/display) 为 inline-block）

• 表格单元格（元素的 [display](https://developer.mozilla.org/zh-CN/docs/Web/CSS/display)为 table-cell，HTML表格单元格默认为该值）

• 表格标题（元素的 [display](https://developer.mozilla.org/zh-CN/docs/Web/CSS/display) 为 table-caption，HTML表格标题默认为该值）

• 匿名表格单元格元素（元素的 [display](https://developer.mozilla.org/zh-CN/docs/Web/CSS/display)为 table、table-row、 table-row-group、table-header-group、table-footer-group（分别是HTML table、row、tbody、thead、tfoot的默认属性）或 inline-table）

• [overflow](https://developer.mozilla.org/zh-CN/docs/Web/CSS/overflow) 值不为 visible 的块元素

• [display](https://developer.mozilla.org/zh-CN/docs/Web/CSS/display) 值为 [flow-root](https://drafts.csswg.org/css-display/#valdef-display-flow-root) 的元素

• [contain](https://developer.mozilla.org/zh-CN/docs/Web/CSS/contain) 值为 layout、content或 paint 的元素

• 弹性元素（[display](https://developer.mozilla.org/zh-CN/docs/Web/CSS/display)为 flex 或 inline-flex元素的直接子元素）

• 网格元素（[display](https://developer.mozilla.org/zh-CN/docs/Web/CSS/display)为 grid 或 inline-grid 元素的直接子元素）

• 多列容器（元素的 [column-count](https://developer.mozilla.org/zh-CN/docs/Web/CSS/column-count) 或 [column-width](https://developer.mozilla.org/zh-CN/docs/Web/CSS/column-width) 不为 auto，包括 column-count 为 1）

• column-span 为 all 的元素始终会创建一个新的BFC，即使该元素没有包裹在一个多列容器中（[标准变更](https://github.com/w3c/csswg-drafts/commit/a8634b96900279916bd6c505fda88dda71d8ec51)，[Chrome bug](https://bugs.chromium.org/p/chromium/issues/detail?id=709362)）。

重绘回流

重绘当元素样式的改变不影响布局时，浏览器将使用重绘对元素进行更新，此时由于只需要UI层面的重新像素绘制回流页面初次渲染浏览器窗口大小改变元素尺寸、位置、内容发生改变元素字体大小变化添加或者删除可见的 dom 元素激活 CSS 伪类（例如：:hover）查询某些属性或调用某些方法（以下属性或方法会刷新渲染队列）clientWidth、clientHeight、clientTop、clientLeftoffsetWidth、offsetHeight、offsetTop、offsetLeftscrollWidth、scrollHeight、scrollTop、scrollLeftgetComputedStyle()getBoundingClientRect()scrollTo()回流必定触发重绘，重绘不一定触发回流。重绘的开销较小，回流的代价较高。

描述清楚重绘和回流并了解之间的关系 大致能说一些 触发方式以及避免方法 3分

JS 基础

写出影响原数组和不影响原数组的方法

影响：pop push unshift shift sort reverse splice

不影响：map forEach filter some every join slice concat indexOf reduce 12个

函数柯里化

定义在一个函数中，首先填充几个参数，然后再返回一个新的函数用处通常可用于在不侵入函数的前提下，为函数 预置通用参数

描述清楚定义以及用法并且讲出一两个实际用法 3分

项目优化

css js webpack 图片 http cache ssr prefetch cdn-prefetch dom回收等 http2 能说出以上7层并言之有物 3分 （http 缓存可以单独问）

什么是防抖和节流？有什么区别？如何实现？ 说清楚2者之间的区别 或 手写代码 说清区别并手写简单实现 3分（手写失败1分）

拷贝 手写代码 写出（考虑完善）3分（能聊聊浅拷贝&&深拷贝更好 3.5-4分）

纯函数&&redux 为什么要把 reducer 设计成纯函数 即相同的输入，永远会得到相同的输出，而且没有任何可观察的副作用（方便测试） 先写一个 && 说清定义 && 在问 reducer问题（简历有 react ) 第一个问题说清楚3分 后面说清楚 3.5分

原型

<https://developer.mozilla.org/zh-CN/docs/Learn/JavaScript/Objects/Object_prototypes>

理解对象的原型（可以通过[Object.getPrototypeOf(obj)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/getPrototypeOf)或者已被弃用的[\_\_proto\_\_](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/proto)属性获得）与构造函数的prototype属性之间的区别是很重要的。前者是每个实例上都有的属性，后者是构造函数的属性。

实例.\_\_proto\_\_ === 原型

原型.constructor === 构造函数

构造函数.prototype === 原型

每个函数function都有一个prototype,每个对象都有一个\_\_proto\_\_

能说清3种之间的关系以及原型链继承3分

当谈到继承时，JavaScript 只有一种结构：对象。每个实例对象（ object ）都有一个私有属性（称之为 \_\_proto\_\_ ）指向它的构造函数的原型对象（**prototype** ）。该原型对象也有一个自己的原型对象( \_\_proto\_\_ ) ，层层向上直到一个对象的原型对象为 null。根据定义，null 没有原型，并作为这个原型链中的最后一个环节。

几乎所有 JavaScript 中的对象都是位于原型链顶端的 [Object](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object) 的实例。原型链 构造函数.prototype 也是一个对象 也有\_\_proto\_\_ 属性也是指向创建它的函数的prototype（原型）object.prototype 每一个对象都有一个—proto\_\_ 对象本质都是Object 创建的

这些含义 && 原型链继承核心是改写原型指向？？ 3分

函数提升

console.log(f1) function f1(){} ; console.log(f2) var f2= function(){} 打印值 打印结果 并问为啥 函数提升与变量提升 3分

从执行上下（词法分析）方向解释3分以上

<https://blog.csdn.net/qq_34178990/article/details/81201495>

对于一个函数执行分为了两个部分：

1. 编译过程——词法分析

2. 执行过程

词法分析需要分析三个内容:

1. 分析函数参数

2. 分析变量声明

3. 分析函数声明

具体步骤：

1.函数运行前的一瞬间， 生成Active Object（活动对象）， 一下简称AO

2.函数声明的参数， 形成AO的属性， 值为实参， 未传值为undefined

3.分析变量声明， 如var age,

1. 如果AO上还没有age属性， 则添加AO属性吗值为undefined

2. 如果**AO**上已经有**age**属性，则不做任何影响

4.分析函数声明， 如function foo(){}, 则把函数赋值给AO.foo属性；如果**foo**存在则无情覆盖掉

var age = 99;

function t() {

console.log(age)

var age = 100

}

t()

在调用t函数时， 形成一个AO

没有参数， 不做参数分析， 此时AO没有属性

分析变量age，将age属性添加到AO上，此时age的值为undefined

分析函数声明， 此时没有函数声明， 跳过

经过分析最终形成了一个AO={age:undefined}

进入执行过程

执行console.log(age), 此时为undefined， 所以打印undefined

执行age = 100

此时AO={age:100}

结束

https://blog.csdn.net/DepressedPrince/article/details/80405018

function add(num) {

var num;

console.log(num); //1

}

add(1);

执行上下文

<https://www.cnblogs.com/congxueda/p/10806927.html>

1. 什么是执行上下文

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

2. 执行上下文的类型

执行上下文总共有三种类型：

• 全局执行上下文： 这是默认的、最基础的执行上下文。不在任何函数中的代码都位于全局执行上下文中。它做了两件事：1. 创建一个全局对象，在浏览器中这个全局对象就是 window 对象。2. 将 this 指针指向这个全局对象。一个程序中只能存在一个全局执行上下文。

• 函数执行上下文： 每次调用函数时，都会为该函数创建一个新的执行上下文。每个函数都拥有自己的执行上下文，但是只有在函数被调用的时候才会被创建。一个程序中可以存在任意数量的函数执行上下文。每当一个新的执行上下文被创建，它都会按照特定的顺序执行一系列步骤，具体过程将在本文后面讨论。

• Eval 函数执行上下文： 运行在 eval 函数中的代码也获得了自己的执行上下文，但由于 Javascript 开发人员不常用 eval 函数，所以在这里不再讨论

二、执行上下文的生命周期

执行上下文的生命周期包括三个阶段：创建阶段 → 执行阶段 → 回收阶段，本文重点介绍创建阶段。

1. 创建阶段

当函数被调用，但未执行任何其内部代码之前，会做以下三件事：

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

• 创建作用域链（Scope Chain）：在执行期上下文的创建阶段，作用域链是在变量对象之后创建的。作用域链本身包含变量对象。作用域链用于解析变量。当被要求解析变量时，JavaScript 始终从代码嵌套的最内层开始，如果最内层没有找到变量，就会跳转到上一层父作用域中查找，直到找到该变量。

• 确定 this 指向：包括多种情况

2. 执行阶段

执行变量赋值、代码执行

3. 回收阶段

执行上下文出栈等待虚拟机回收执行上下文

JS 问答题

JS 闭包的原理和使用场景

函数与对其状态即词法环境（**lexical environment**）的引用共同构成闭包（**closure**）。也就是说，闭包可以让你从内部函数访问外部函数作用域。在JavaScript，函数在每次创建时生成闭包。

闭包来定义公共函数，并令其可以访问私有函数和变量。这个方式也称为 [模块模式（module pattern）：](http://www.google.com/search?q=javascript+module+pattern)

JS 的执行队列

JS 的原型链和继承

当谈到继承时，JavaScript 只有一种结构：对象。每个实例对象（ object ）都有一个私有属性（称之为 \_\_proto\_\_ ）指向它的构造函数的原型对象（**prototype** ）。该原型对象也有一个自己的原型对象( \_\_proto\_\_ ) ，层层向上直到一个对象的原型对象为 null。根据定义，null 没有原型，并作为这个原型链中的最后一个环节。

JS 异步编程方法

callback -> promise -> generator -> async + await

前端问答题

HTTPS的连接过程

https://blog.csdn.net/xiaopang\_yan/article/details/78709574

**step1**： “客户”向服务端发送一个通信请求

“客户”->“服务器”：你好

**step2**： “服务器”向客户发送自己的数字证书。证书中有一个公钥用来加密信息，私钥由“服务器”持有

“服务器”->“客户”：你好，我是服务器，这里是我的数字证书

**step3**： “客户”收到“服务器”的证书后，它会去验证这个数字证书到底是不是“服务器”的，数字证书有没有什么问题，数字证书如果检查没有问题，就说明数字证书中的公钥确实是“服务器”的。检查数字证书后，“客户”会发送一个随机的字符串给“服务器”用私钥去加密，服务器把加密的结果返回给“客户”，“客户”用公钥解密这个返回结果，如果解密结果与之前生成的随机字符串一致，那说明对方确实是私钥的持有者，或者说对方确实是“服务器”。

“客户”->“服务器”：向我证明你就是服务器，这是一个随机字符串     //前面的例子中为了方便解释，用的是“你好”等内容，实际情况下一般是随机生成的一个字符串。

“服务器”->“客户”：{一个随机字符串}[私钥|RSA]

**step4**： 验证“服务器”的身份后，“客户”生成一个对称加密算法和密钥，用于后面的通信的加密和解密。这个对称加密算法和密钥，“客户”会用公钥加密后发送给“服务器”，别人截获了也没用，因为只有“服务器”手中有可以解密的私钥。这样，后面“服务器”和“客户”就都可以用对称加密算法来加密和解密通信内容了。

“服务器”->“客户”：{OK，已经收到你发来的对称加密算法和密钥！有什么可以帮到你的？}[密钥|对称加密算法]

“客户”->“服务器”：{我的帐号是aaa，密码是123，把我的余额的信息发给我看看}[密钥|对称加密算法]

“服务器”->“客户”：{你好，你的余额是100元}[密钥|对称加密算法]

浏览器的首次渲染过程，哪些情况会阻塞渲染

https://blog.csdn.net/weixin\_41835977/article/details/88878226

1，当浏览器遇到script的标记，DOM构建将暂停，直到脚本完成执行；

2，CSSOM构建时，JavaScript执行暂停，直到CSSOM就绪。

1，CSS优先，引入顺序上CSS资源优先于JavaScript资源；2，JavaScript尽量少影响DOM构建；

https://juejin.im/post/5b88ddca6fb9a019c7717096

由上所述，我们可以得出以下结论:

1 css加载不会阻塞DOM树的解析

2 css加载会阻塞DOM树的渲染

3 css加载会阻塞后面js语句的执行

因此，为了避免让用户看到长时间的白屏时间，我们应该尽可能的提高css加载速度，比如可以使用以下几种方法:

1.使用CDN(因为CDN会根据你的网络状况，替你挑选最近的一个具有缓存内容的节点为你提供资源，因此可以减少加载时间)

2.对css进行压缩(可以用很多打包工具，比如webpack,gulp等，也可以通过开启gzip压缩)

3.合理的使用缓存(设置cache-control,expires,以及E-tag都是不错的，不过要注意一个问题，就是文件更新后，你要避免缓存而带来的影响。其中一个解决防范是在文件名字后面加一个版本号)

4.减少http请求数，将多个css文件合并，或者是干脆直接写成内联样式(内联样式的一个缺点就是不能缓存)

如果用户无法访问，如何进行排查

框架问答题

Vue组件的生命周期

Vue 的数据绑定过程

当你把一个普通的 JavaScript 对象传入 Vue 实例作为 data 选项，Vue 将遍历此对象所有的属性，并使用 [Object.defineProperty](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty) 把这些属性全部转为 **[getter/setter](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Working_with_Objects#%E5%AE%9A%E4%B9%89_getters_%E4%B8%8E_setters)**。Object.defineProperty 是 ES5 中一个无法 shim 的特性，这也就是 Vue 不支持 IE8 以及更低版本浏览器的原因。

每个组件实例都对应一个 **watcher** 实例，它会在组件渲染的过程中把“接触”过的数据属性记录为依赖。之后当依赖项的 setter 触发时，会通知 watcher，从而使它关联的组件重新渲染。

Vue 异步组件如何加载

Vue 允许你以一个工厂函数的方式定义你的组件，这个工厂函数会异步解析你的组件定义。Vue 只有在这个组件需要被渲染的时候才会触发该工厂函数，且会把结果缓存起来供未来重渲染。

Vue.component(

'async-webpack-example',

// 这个 `import` 函数会返回一个 `Promise` 对象。

() => import('./my-async-component')

)

React SetState 是同步还是异步？

React 组件的生命周期

React 和 Vue 的区别是什么？

<https://segmentfault.com/a/1190000015382068?utm_source=tag-newest>

这里不太清楚

• 两者本质的区别：模板和组件化的区别

◦ Vue本质是MVVM框架，由MVC发展而来；

◦ React是前端组件化框架，由后端组件化发展而来；

• Vue使用模板

• React使用JSX

• React本身就是组件化

• Vue是在MVVM上扩展的

• 共同点：

◦ 都支持组件化，都是数据驱动视图

工程化问答题

如何构建可执行的npm 包

https://www.cnblogs.com/jiasm/p/9857992.html

*npx*可以做更多的事情，在这里先认为它是*./node\_modules/webpack/bin/webpack.js*的简写就好了

*npm scripts*中不仅仅可以执行*NPM*模块，普通的*shell*命令都是支持的

创建执行文件

因为我们这个模块就是用来执行使用的，所以有没有入口文件实际上是没有必要的，我们仅仅需要创建对应的执行文件即可，需要注意的一点是：与普通的**JS**文件区别在于头部一定要写上**#!/usr/bin/env node**

注册执行命令

然后就是修改package.json来告诉NPM我们的执行文件在哪：

{

+ "bin": "./index.js"

}

在只有一个bin，且要注册的命令与package.json中的name字段相同时，则可以写成上边那种形式，如果要注册多个可执行命令，那么就可以写成一个k/v结构的参数：

{

"bin": {

"command1": "./command1.js",

"command2": "./command2.js"

}

}

调用时就是 command1 | command2

webpack 有哪些优化打包体积的方法

webpack 常用的插件和loader设置

使用的构建流水线过程

prepare

run:

      - npm config set registry http://registry.npm.taobao.org/ && yarn install

build

    run:

     - yarn install && yarn run build:production

deploy

    command: yarn run production

技术架构

描述一个复杂组件/功能的设计方案

活动配置平台：

编辑器交互部分

页面数据对象的增删改

组件通信

描述一个系统的架构设计

https://www.cnblogs.com/hofmann/p/10815530.html

大型动态应用系统又可分为几个子系统：

• Web前端系统

• 负载均衡系统

• 数据库集群系统

• 缓存系统

• 分布式存储系统

• 分布式服务器管理系统

• 代码分发系统

框架和库的区别是什么

我的理解：框架是一个系统，库是工具箱

http://www.imooc.com/wenda/detail/572441

根据[马丁·福勒](http://martinfowler.com/bliki/InversionOfControl.html):

A 图书馆本质上是一组你可以调用的函数，现在通常被组织成类。每个调用都做一些工作，并将控制返回给客户端。

A 框架体现了一些抽象的设计，与更多的行为内置。为了使用它，您需要通过子类或插入您自己的类将您的行为插入到框架中的不同位置。然后，框架的代码在这些点上调用您的代码

描述几个设计模式，画出关系图？？

经历过技术难度最大的项目是什么？

技术管理

使用了哪些质量管理方法？

如何提升团队开发质量？

https://baijiahao.baidu.com/s?id=1621149311386570890&wfr=spider&for=pc

第一：按角色管理开发团队。保证每个人都能够胜任自己的工作，岗位工作难度下降则能在一定程度上保障软件的开发质量。

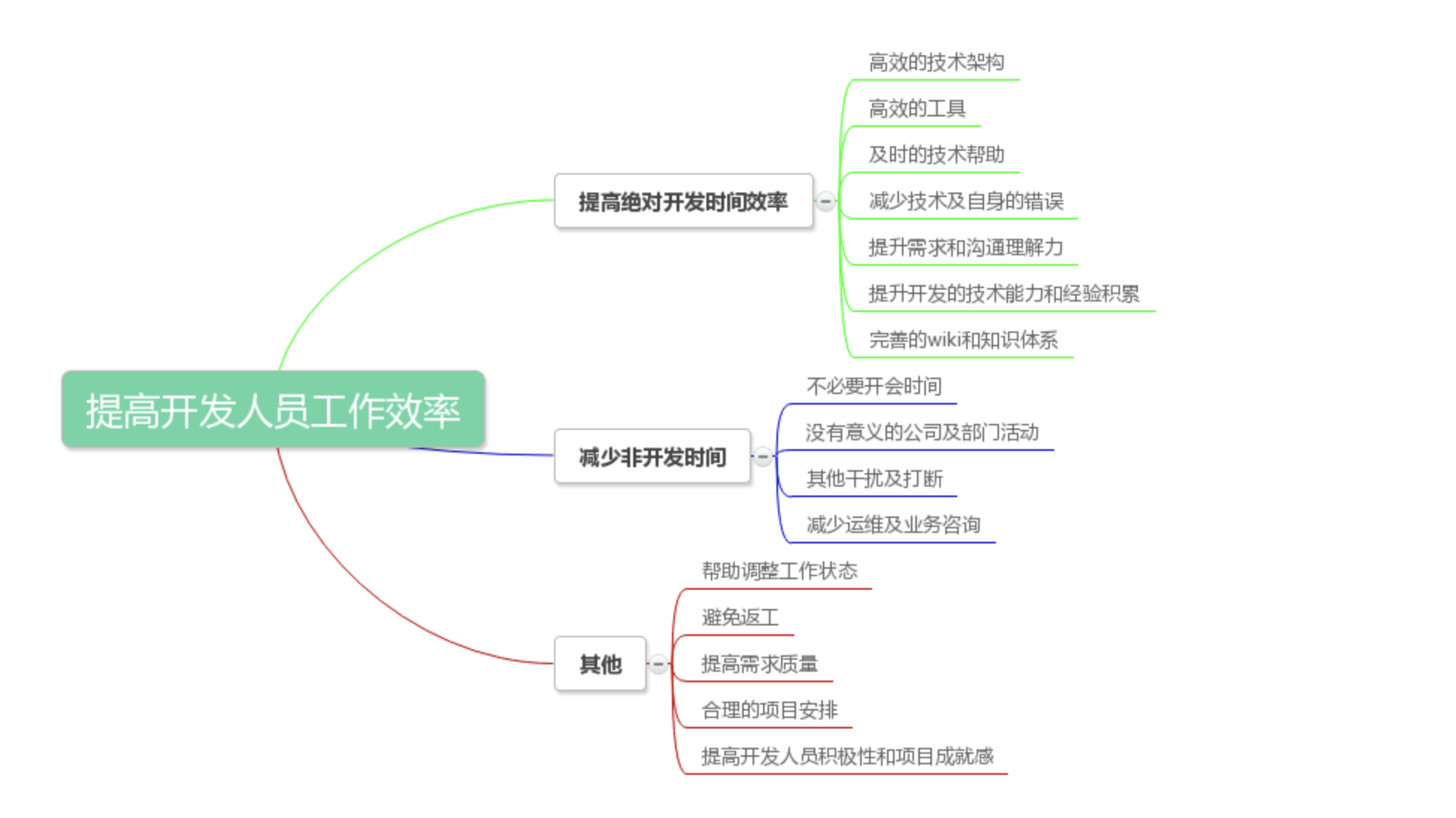
第二：量化开发任务。每天都有非常详细的量化工作指标，每完成一个就提交一个，整个工作过程的管理非常细致，对每个任务的质量也有非常详细的要求，这是保障软件质量的一个有效办法。

第三：善于利用管理工具。目前有大量的软件管理工具可以使用，使用软件管理工具一方面可以方便团队成员之间的协作，另一方面也能保证软件代码的质量。当然，工具本身只是起到辅助的作用。

第四：交流。团队之间的定期交流对于提升代码质量还是非常有帮助的，目前不少团队都会做Code Review，这种轻量级的代码评审过程对于提升代码质量还是有较大帮助的。

第五：培训。定期培训可以不断提升程序员自身的代码编写能力，培训的过程会接触到更多的流行技术和发展趋势，对于软件团队来说，一定要重视培训的作用。

如何提升团队开发效率？

<https://www.cnblogs.com/zhangweizhong/p/5137960.html>

近期的技术规划是什么？为什么他们优先级最高？

如何提升同学的开发能力？

https://kb.cnblogs.com/page/522513/

目标应该设定在解决哪一类问题，而不是精通哪一类技术。技术只是手段，不是目的。例如，“我要成为iOS developer中的达人”这个目标，就远不如“我要成为前端应用开发的专家”来得有意义。前者学到深处你可能会去钻研iOS framework里各种奇技淫巧，而后者你会开始关注视觉与交互设计，研究各平台间的差异与共同趋势。显然，后者更有助于你的个人发展。

初入一个领域时，选择主流技术框架；当你有一定经验后，选择技术时更应该关注背后的推动者，我相信优秀的人和团队总能打造优秀的产品，无论是商业公司还是开源社区。不必太在意技术的新旧，因为可能很快都会成为过去时。你真正要学习的是技术背后的思想。

我的另一个学习原则是，在选择学习一门新技术时，最大化它与你现有知识库的差异性。

最后我建议每个程序员都应该经营一款自己的产品，它可以是一款app，一个网站或是一个开源软件。

如何做技术选型？

https://blog.csdn.net/shf4715/article/details/82961602

先测试、研究，再决定

快速搭建原型，不要从博客学习，而要从经验学习。针对新技术提供的功能，在决定采用之前花一两天搭个原型，然后组织大家分析利弊。你可能会遇到若干能彼此替代的技术，可以让团队里不同人用不同的技术来搭原型。

黑客马拉松，这也是不错的办法，它让大家真正感受到不同技术的代价。对所有兼具风险和诱惑力的技术，都让整个团队花一两天来把玩。这会让大家自主做出聪明的选择，根据自己的经验来决策。

算法逻辑题

js二进制

十进制转换为二进制：

var num = 100;

console.log(num.toString(2));

二进制转十进制：

var num = 1100100;

console.log(parseInt(num,2));

https://baike.baidu.com/item/十进制转二进制/393189?fr=aladdin

十进制转换二进制**python**代码

**def** Dec2Bin(dec):

    temp **=** []

    result **=** ''

**while** dec:

        quo **=** dec **%** 2

        dec **=** dec **//** 2

        temp.append(quo)

**while** temp:

        result **+=** str(temp.pop())

**return** result

位运算实现？？

大致思路：

let str;

Let result=[]

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

let left=Str>>i

let curB=left&1

Result.unshift(curB)

}

Return result

js实现大数相乘

<https://blog.csdn.net/weixin_34061482/article/details/88766482>

var multiply = function (num1, num2) {

    if (isNaN(num1) || isNaN(num2)) return '' //判断输入是不是数字

    var len1 = num1.length,

        len2 = num2.length

    var ans = []

    for (var i = len1 - 1; i >= 0; i--) {    //这里倒过来遍历很妙,不需要处理进位了

        for (var j = len2 - 1; j >= 0; j--) {

            var index1 = i + j

            var index2 = i + j + 1

            var mul = num1[i] \* num2[j] + (ans[index2] || 0)

            ans[index1] = Math.floor(mul / 10) + (ans[index1] || 0)

            ans[index2] = mul % 10

        }

    }

    var result = ans.join('')

    //这里结果有可能会是多个零的情况，需要转成数字判断

    //原来写的是return +result === 0 ? '0' : result，result字符串会出现有前置0的情况，感谢评论区小伙伴@nicknice的提醒让我找到了这个错误

    return +result === 0 ? '0' : result.replace(/^0+/, '')

}

二维数组环型遍历

<https://blog.csdn.net/xutian_curry/article/details/80685615>

int main()

{

cin >> row >> col;

for( int i = 1; i <= row ; i++ )

for( int j = 1; j <= col; j++ )

cin >> num[i][j];

int r\_begin = 1, c\_begin = 1;

while( r\_begin <= row || c\_begin <= col )

{

for( int i = r\_begin, j = c\_begin; j <= col && r\_begin <= row; j++ )

cout << num[i][j] << endl;

r\_begin++;

for( int i = r\_begin, j = col; i <= row && c\_begin <= col; i++ )

cout << num[i][j] << endl;

col--;

for( int i = row, j = col; j >= c\_begin && r\_begin <= row; j-- )

cout << num[i][j] << endl;

row--;

for( int i = row, j = c\_begin; i >= r\_begin && c\_begin <= col; i-- )

cout << num[i][j] << endl;

c\_begin++;

}

return 0;

三角形打印数字

js三角形

https://www.cnblogs.com/TreeCTJ/p/9691752.html

将一个多维数组拍平

<https://www.cnblogs.com/guan-shan/p/10165737.html>

1.递归

function beatFlat(arr) {

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

        if (!Array.isArray(arr[i])) {// 或者 Array.isArray(arr[i])

            arrResult.push(arr[i]);

        } else {

            beatFlat(arr[i]);

        }

    }

    return arrResult;

}

function flatten(arr){

 return arr.reduce(function(pre,cur){

  if(!Array.isArray(cur)){

   return [...pre,cur];

  }else{

   return [...pre,...flatten(cur)]

  }

 },[])

}

2.arr.toString().split(‘,’)

实现Array 的 forEach / filter / map /reduce

forEach

// Production steps of ECMA-262, Edition 5, 15.4.4.18

// Reference: http://es5.github.io/#x15.4.4.18

if (!Array.prototype.forEach) {

Array.prototype.forEach = function(callback, thisArg) {

var T, k;

if (this == null) {

throw new TypeError(' this is null or not defined');

}

var O = Object(this);

// 3. Let len be toUint32(lenValue).

var len = O.length >>> 0;

if (typeof callback !== "function") {

throw new TypeError(callback + ' is not a function');

}

// 5. If thisArg was supplied, let T be thisArg; else let

// T be undefined.

if (arguments.length > 1) {

T = thisArg;

}

// 6. Let k be 0

k = 0;

// 7. Repeat, while k < len

while (k < len) {

var kValue;

if (k in O) {

kValue = O[k];

callback.call(T, kValue, k, O);

}

// d. Increase k by 1.

k++;

}

// 8. return undefined

};

}

Filter:

if (!Array.prototype.filter){

Array.prototype.filter = function(func, thisArg) {

'use strict';

if ( ! ((typeof func === 'Function' || typeof func === 'function') && this) )

throw new TypeError();

var len = this.length >>> 0,

res = new Array(len), // preallocate array

t = this, c = 0, i = -1;

if (thisArg === undefined){

while (++i !== len){

// checks to see if the key was set

if (i in this){

if (func(t[i], i, t)){

res[c++] = t[i];

}

}

}

}

else{

while (++i !== len){

// checks to see if the key was set

if (i in this){

if (func.call(thisArg, t[i], i, t)){

res[c++] = t[i];

}

}

}

}

res.length = c; // shrink down array to proper size

return res;

};

}

reduce:

// Production steps of ECMA-262, Edition 5, 15.4.4.21

// Reference: http://es5.github.io/#x15.4.4.21

// https://tc39.github.io/ecma262/#sec-array.prototype.reduce

if (!Array.prototype.reduce) {

Object.defineProperty(Array.prototype, 'reduce', {

value: function(callback /\*, initialValue\*/) {

if (this === null) {

throw new TypeError( 'Array.prototype.reduce ' +

'called on null or undefined' );

}

if (typeof callback !== 'function') {

throw new TypeError( callback +

' is not a function');

}

// 1. Let O be ? ToObject(this value).

var o = Object(this);

// 2. Let len be ? ToLength(? Get(O, "length")).

var len = o.length >>> 0;

// Steps 3, 4, 5, 6, 7

var k = 0;

var value;

if (arguments.length >= 2) {

value = arguments[1];

} else {

while (k < len && !(k in o)) {

k++;

}

// 3. If len is 0 and initialValue is not present,

// throw a TypeError exception.

if (k >= len) {

throw new TypeError( 'Reduce of empty array ' +

'with no initial value' );

}

value = o[k++];

}

// 8. Repeat, while k < len

while (k < len) {

if (k in o) {

value = callback(value, o[k], k, o);

}

// d. Increase k by 1.

k++;

}

// 9. Return accumulator.

return value;

}

});

}

实现extend 方法

var obj1 = {'a': 'obj2','b':'2'};

var obj2 = {name: 'obj3'};

function extend() {

var length = arguments.length;

var target = arguments[0] || {};

if (typeof target!="object" && typeof target != "function") {

target = {};

}

if (length == 1) {

target = this;

i--;

}

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

var source = arguments[i];

for (var key in source) {

// 使用for in会遍历数组所有的可枚举属性，包括原型。

//JavaScript 并没有保护 hasOwnProperty 属性名，因此某个对象 是有可能存在使用这个属性名的属性，使用外部的 hasOwnProperty 获得正确的结果是需要的

if (Object.prototype.hasOwnProperty.call(source, key)) {

target[key] = source[key];

}

}

}

return target;

}

console.log(extend(obj1,obj2));

Object.assign

if (typeof Object.assign != 'function') {

// Must be writable: true, enumerable: false, configurable: true

Object.defineProperty(Object, "assign", {

value: function assign(target, varArgs) { // .length of function is 2

'use strict';

if (target == null) { // TypeError if undefined or null

throw new TypeError('Cannot convert undefined or null to object');

}

let to = Object(target);

for (var index = 1; index < arguments.length; index++) {

var nextSource = arguments[index];

if (nextSource != null) { // Skip over if undefined or null

for (let nextKey in nextSource) {

// Avoid bugs when hasOwnProperty is shadowed

if (Object.prototype.hasOwnProperty.call(nextSource, nextKey)) {

to[nextKey] = nextSource[nextKey];

}

}

}

}

return to;

},

writable: true,

configurable: true

});

}

实现bind 方法

if (!Function.prototype.bind) {

Function.prototype.bind = function(oThis) {

if (typeof this !== 'function') {

// closest thing possible to the ECMAScript 5

// internal IsCallable function

throw new TypeError('Function.prototype.bind - what is trying to be bound is not callable');

}

var aArgs = Array.prototype.slice.call(arguments, 1),

fToBind = this,

fNOP = function() {},

fBound = function() {

// this instanceof fBound === true时,说明返回的fBound被当做new的构造函数调用

return fToBind.apply(this instanceof fBound

? this

: oThis,

// 获取调用时(fBound)的传参.bind 返回的函数入参往往是这么传递的

aArgs.concat(Array.prototype.slice.call(arguments)));

};

// 维护原型关系

if (this.prototype) {

// 当执行Function.prototype.bind()时, this为Function.prototype

// this.prototype(即Function.prototype.prototype)为undefined

fNOP.prototype = this.prototype;

}

// 下行的代码使fBound.prototype是fNOP的实例,因此

// 返回的fBound若作为new的构造函数,new生成的新对象作为this传入fBound,新对象的\_\_proto\_\_就是fNOP的实例

fBound.prototype = new fNOP();

return fBound;

};

}

Debounce 和 Throttle

实现表格的点击表头排序功能

https://www.cnblogs.com/zczhangcui/p/6329736.html

function makeSortable(table) {

var headers=table.getElementsByTagName("th");

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

(function(n){

var flag=false;

headers[n].onclick=function(){

// sortrows(table,n);

var tbody=table.tBodies[0];//第一个<tbody>

var rows=tbody.getElementsByTagName("tr");//tbody中的所有行

rows=Array.prototype.slice.call(rows,0);//真实数组中的快照

//基于第n个<td>元素的值对行排序

rows.sort(function(row1,row2){

var cell1=row1.getElementsByTagName("td")[n];//获得第n个单元格

var cell2=row2.getElementsByTagName("td")[n];

var val1=cell1.textContent||cell1.innerText;//获得文本内容

var val2=cell2.textContent||cell2.innerText;

if(val1<val2){

return -1;

}else if(val1>val2){

return 1;

}else{

return 0;

}

});

if(flag){

rows.reverse();

}

//在tbody中按它们的顺序把行添加到最后

//这将自动把它们从当前位置移走，故没必要预先删除它们

//如果<tbody>还包含了除了<tr>的任何其他元素，这些节点将会悬浮到顶部位置

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

tbody.appendChild(rows[i]);

}

flag=!flag;

}

}(i));

}

}