# 前端的面试

# HTTP

## 1、什么是防抖和节流？有什么区别？如何实现？

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

思路：每次触发事件时都取消之前的延时调用方法

export function debounce(*func*,*delay*){

  let timer=null

  return function(...*args*){

    if(timer)

    clearTimeout(timer)

    timer=setTimeout(() => {

*func*.apply(*this*,*args*)

    },*delay*);

  }

};

function debounce(fn) {

let timeout = null;

// 创建一个标记用来存放定时器的返回值

returnfunction () {

clearTimeout(timeout);

// 每当用户输入的时候把前一个 setTimeout clear 掉

timeout = setTimeout(() => {

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

}, 500)}}

functionsayHi() { console.log('防抖成功'); }

var inp = document.getElementById('inp')

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

高频事件触发，但在n秒内只会执行一次，所以节流会稀释函数的执行频率

思路：每次触发事件时都判断当前是否有等待执行的延时函数

## 2、get请求传参长度的误区、get和post请求在缓存方面的区别（二点）

误区：我们经常说get**请求参数**的大小存在限制，而post请求的参数大小是无限制的。

参考答案实际上HTTP 协议从未规定 GET/POST 的请求长度限制是多少。对**get**请求参数的限制是来源与浏览器或web服务器，浏览器或web服务器限制了url的长度。为了明确这个概念，我们必须再次强调下面几点:

### 1.get请求参数的限制：浏览器和web服务器限url的长度

HTTP 协议 未规定 GET 和POST的长度限制GET的最大长度显示是因为 **浏览器和 web服务器**限制了 URI的长度。不同的浏览器和WEB服务器，限制的最大长度不一样要支持IE，则最大长度为2083byte，若只支持Chrome，则最大长度 8182byte。

### 2.get（无数据库，缓存）和post（数据库，无缓存）

get请求类似于查找的过程，用户获取数据，可以不用每次都与数据库连接，所以可以使用缓存。post不同，post做的一般是修改和删除的工作，所以必须与数据库交互，所以不能使用缓存。因此get请求适合于请求缓存。

1.[Get是](https://zq.zhaopin.com/answer/6161144/)**[不安全](https://zq.zhaopin.com/answer/6161144/)**[的，因为在传输过程，数据被放在请求的](https://zq.zhaopin.com/answer/6161144/)**[URL](https://zq.zhaopin.com/answer/6161144/)**[中；Post的所有操作对用户来说都是](https://zq.zhaopin.com/answer/6161144/)**[不可见](https://zq.zhaopin.com/answer/6161144/)**[的。](https://zq.zhaopin.com/answer/6161144/)

[2. Get传送的](https://zq.zhaopin.com/answer/6161144/)**[数据量较小](https://zq.zhaopin.com/answer/6161144/)**[，这主要是因为受](https://zq.zhaopin.com/answer/6161144/)**[URL长度限制](https://zq.zhaopin.com/answer/6161144/)**[；Post传送的数据量较大，一般被默认为不受限制。](https://zq.zhaopin.com/answer/6161144/)

[3. Get限制Form表单的数据集的值必须为](https://zq.zhaopin.com/answer/6161144/)**[ASCII字符](https://zq.zhaopin.com/answer/6161144/)**[；而Post支持整个](https://zq.zhaopin.com/answer/6161144/)**[ISO10646字符集](https://zq.zhaopin.com/answer/6161144/)**[。](https://zq.zhaopin.com/answer/6161144/)

[4. Get](https://zq.zhaopin.com/answer/6161144/)**[执行效率](https://zq.zhaopin.com/answer/6161144/)**[却比Post方法好。Get是form提交的默认方法。](https://zq.zhaopin.com/answer/6161144/)

1、Http get方法提交的数据大小长度并没有限制，Http协议规范没有对URL长度进行限制。目前说的get长度有限制，是特定的浏览器及服务器对它的限制。

2、get是最常见的请求类型，常用于向服务器查询某些信息，必要时可以将查询字符串参数追加到url的末尾，以便将信息发送给服务器。

3、post使用频率仅次于get请求，通常用于向服务器发送应该被保存的数据，post请求应该把数据作为请求的主体提交，而get请求传统上不是这样。post请求的主体可以包含非常多的数据，而且格式不限。

### get与post的区别，什么时候使用post？

1.get和post在HTTP中都代表着请求数据，其中get请求相对来说更简单、快速，效率高些

2.get相对post安全性低

3.get有缓存，post没有

4.get体积小，post可以无限大

5.get的url参数可见，post不可见

6.get只接受ASCII字符的参数数据类型，post没有限制

7.get请求参数会保留历史记录，post中参数不会保留

8.get会被浏览器主动catch，post不会，需要手动设置

9.get在浏览器回退时无害，post会再次提交请求

10.post一般用于修改服务器上的资源，对所发送的信息没有限制。比如

1. 无法使用缓存文件（更新服务器上的文件或数据库）

2. 向服务器发送大量数据（POST 没有数据量限制）

3. 发送包含未知字符的用户输入时，POST 比 GET 更稳定也更可靠

### 跨域

跨域的条件：**同源策略（协议 域名 端口 三者）三者任一不同就为跨域**

跨域的解决方案

**jsonp** （json padding) (callback({a: 1, b: 2})) 前端定义好这个函数的业务逻辑，利用**script标签**可以引入任意外部的资源

**cors（跨站资源共享）下面作为了解**

在使用时前端正常发起异步请求，后端需要设置允许的请求头 请求方法 请求域名

access-control-allow-origin: 设置请求域名/主机

access-control-allow-headers: 设置请求头

access-control-allow-methods: 设置请求方法

什么是预检请求 options类型，什么情况下发起预检请求

简单请求不需要发起预检，直接传输数据到后台

请求方法：get post head

请求头：accept accept-lunguage content-type content-length last-eventID

发送请求的数据格式：text/plain application/x-www-form-urlencoded multipart/form-data

非简单请求需要先进行预检，如果后台支持的话再携带数据发起请求到后台

反向代理（客户端 => 网页服务器(nginx) => node.js / PHP / java / python)

如果代理服务器代理的是后台的处理程序，那就是反向代理。当前页面www.a.com，想请求一个接口www.b.com/api/test，此时网页正常发起请求www.a.com/api/test到达网页服务器nginx，nginx此时收到请求，重置请求域名到www.b.com下的接口地址上，当真正的接口数据返回后，nginx再把数据返回给前端（前端 => 开发服务器8080 => 8888）

iframe + window.name / window.hash

比如当前页面1www.a.com/abc，请求的接口地址www.b.com/api/test。在页面1下内嵌一个iframe，iframe的src属性赋值www.b.com/abc页面2，在页面2中发起异步请求，拿到数据后写在window.name或者window.hash，重置iframe的src属性（更改页面地址），更改到a域名下的一个页面 3，页面3和页面2是同一个窗口，所以页面3能够取到页面2写在window.name或者window.hash的数据

onMessage和postMessage: IE6 7不支持，ie8支持部分内容

document.domain: 仅适用主域相同跨子域的情况. 页面位于www.a.com下，请求的接口位于api.a.com

1：JSONP  
JSONP是服务器与客户端跨源通信的常用方法。最大特点就是简单适用，老式浏览器全部支持，  
服务器改造非常小。  
JSONP的原理是利用**script标签的src属性可以进行不受同源策略的限制**，进行跨域请求数据的属性，在HTML页面中添加一个script标签，向服务器发送请求,服务器收到请求后，返回数据，  
将数据放在指定的回调函数中，回调函数中可以对数据进行操作

CORS是W3C标准，它允许浏览器向跨源服务器发送**AJAX请求**

GET和POST的区别：  
1.post更安全（不会作为url的一部分，不会被缓存、保存在服务器日志、以及浏览器浏览记录中）  
2.post发送的数据量更大（get有url长度限制）  
3.post能发送更多的**数据类型**（get只能发送ASCII字符）  
4.post比get慢  
5.get是**幂等**的，而post不是幂等的  
6.get可以被缓存，post不可以被缓存  
  
因为get请求的请求参数是放在**header**中（添加在链接后面），而post请求的参数是放在**body**中，其实url的长度限制不是get和post所决定的，而是浏览器有所限制，和get、post没有基本没有关系  
get和post相比，要快，是因为get和post的请求过程不同，不多前三步都是一样的，都是要先经过和服务器的三次握手：  
1.浏览器请求tcp连接（第一次握手）  
2.服务器答应进行tcp连接（第二次握手）  
3.浏览器确认，并发送get请求头和数据（第三次握手，这个报文比较小，所以http会在此时进行第一次数据发送）  
**get请求在第四步就会接收到了服务器返回的数据**，而post请求在第四步是接收到服务器发送的100 continue指令，**客户端在发送数据请求服务端，服务端才会去返回数据，**就是说POST比GET多进行了一次客户端和服务器的打交道，GET请求产生了**一个TCP数据包**，而POST请求产生了**两个**TCP数据包

http是在应用层进行数据（数据流）传送的

传输控制协议（TCP，Transmission Control Protocol）是一种面向连接的、可靠的、基于**字节流**的传输层通信协议

## 模块化发展历程

### IIFE、AMD、CMD、CommonJS、UMD、webpack、ES6

### 模块化：抽离公共代码、隔离作用域、避免变量冲突

可从IIFE、AMD、CMD、CommonJS、UMD、webpack(require.ensure)、ES Module、<script type="module"> 这几个角度考虑。

参考答案模块化主要是用来抽离公共代码，隔离作用域，避免变量冲突等。

IIFE：使用自执行函数来编写模块化，特点：在一个单独的函数作用域中执行代码，避免变量冲突。

(function(){return { data:[] }})()

AMD：使用requireJS 来编写模块化，特点：依赖必须提前声明好。

define(./index.js',function(code){

// code 就是index.js 返回的内容

})

CMD：使用seaJS 来编写模块化，特点：支持动态引入依赖文件。

define(function(require, exports, module) {

var indexCode = require('./index.js');})

CommonJS：nodejs 中自带的模块化。

var fs = require('fs')

UMD：兼容AMD，CommonJS 模块化语法。

webpack(require.ensure)：webpack 2.x 版本中的代码分割。

ES Modules：ES6 引入的模块化，支持import 来引入另一个 js 。

import a from 'a';

## npm 模块安装机制，为什么输入 npm install 就可以自动安装对应的模块？

安装使用fastclick插件步骤

npm install fastclick

在main.js中导入

import FastClick from "fastclick"

Vue.use(toast)

FastClick.attach(document.body)

### 1.安装机制npm install-registry压缩包-node\_modules解压

npm 模块安装机制：

发出npm install命令查询node\_modules目录之中是否已经存在,指定模块npm 向 registry 查询模块压缩包的网址下载压缩包，存放在根目录下的.npm目录里解压压缩包到当前项目的node\_modules目录若存在，不再重新安装

### 2.npm 实现原理

npm install-package.json(运行时依赖，开发时依赖确定版本)->npm查询缓存 有直接用 没有仓库下载->package-lock.json(确定最终版本仓库获取压缩包)

执行模块中的生命周期函数（按照 preinstall、install、postinstall 的顺序）。执行工程自身生命周期当前 npm 工程如果定义了钩子此时会被执行（按照 install、postinstall、prepublish、prepare 的顺序）。

输入 npm install 命令并敲下回车后，会经历如下几个阶段（以 npm 5.5.1 为例）：

执行工程自身 preinstall当前 npm 工程如果定义了 preinstall 钩子此时会被执行。确定首层依赖模块首先需要做的是确定工程中的首层依赖，也就是 dependencies 和 devDependencies 属性中直接指定的模块（假设此时没有添加 npm install 参数）。工程本身是整棵依赖树的根节点，每个首层依赖模块都是根节点下面的一棵子树，npm 会开启多进程从每个首层依赖模块开始逐步寻找更深层级的节点。获取模块获取模块是一个递归的过程，分为以下几步：获取模块信息。在下载一个模块之前，首先要确定其版本，这是因为 package.json 中往往是 semantic version（semver，语义化版本）。此时如果版本描述文件（npm-shrinkwrap.json 或 package-lock.json）中有该模块信息直接拿即可，如果没有则从仓库获取。如 packaeg.json 中某个包的版本是 ^1.1.0，npm 就会去仓库中获取符合 1.x.x 形式的最新版本。获取模块实体。上一步会获取到模块的压缩包地址（resolved 字段），npm 会用此地址检查本地缓存，缓存中有就直接拿，如果没有则从仓库下载。查找该模块依赖，如果有依赖则回到第1步，如果没有则停止。模块扁平化（dedupe）上一步获取到的是一棵完整的依赖树，其中可能包含大量重复模块。比如 A 模块依赖于 loadsh，B 模块同样依赖于 lodash。在 npm3 以前会严格按照依赖树的结构进行安装，因此会造成模块冗余。从 npm3 开始默认加入了一个 dedupe 的过程。它会遍历所有节点，逐个将模块放在根节点下面，也就是 node-modules 的第一层。当发现有重复模块时，则将其丢弃。这里需要对重复模块进行一个定义，它指的是模块名相同且 semver 兼容。每个 semver 都对应一段版本允许范围，如果两个模块的版本允许范围存在交集，那么就可以得到一个兼容版本，而不必版本号完全一致，这可以使更多冗余模块在 dedupe 过程中被去掉。比如 node-modules 下 foo 模块依赖 lodash@^1.0.0，bar 模块依赖 lodash@^1.1.0，则 ^1.1.0 为兼容版本。而当 foo 依赖 lodash@^2.0.0，bar 依赖 lodash@^1.1.0，则依据 semver 的规则，二者不存在兼容版本。会将一个版本放在 node\_modules 中，另一个仍保留在依赖树里。举个例子，假设一个依赖树原本是这样：node\_modules -- foo ---- lodash@version1-- bar ---- lodash@version2假设 version1 和 version2 是兼容版本，则经过 dedupe 会成为下面的形式：node\_modules -- foo-- bar-- lodash（保留的版本为兼容版本）假设 version1 和 version2 为非兼容版本，则后面的版本保留在依赖树中：node\_modules -- foo -- lodash@version1-- bar ---- lodash@version2安装模块这一步将会更新工程中的 node\_modules，并执行模块中的生命周期函数（按照 preinstall、install、postinstall 的顺序）。执行工程自身生命周期当前 npm 工程如果定义了钩子此时会被执行（按照 install、postinstall、prepublish、prepare 的顺序）。最后一步是生成或更新版本描述文件，npm install 过程完成。

## ES5的继承和ES6的继承有什么区别？

ES5:prototype-子类parent.apply(this)

ES6:父类-子类extends(constructor->必须调用super(this)) class定义类

ES5的继承时通过prototype或构造函数机制来实现。ES5的继承实质上是先创建子类的实例对象，然后再将父类的方法添加到this上（Parent.apply(this)）。

ES6的继承机制完全不同，实质上是先创建父类的实例对象this（所以必须先调用父类的super()方法），然后再用子类的构造函数修改this。

具体的：ES6通过class关键字定义类，里面有构造方法，类之间通过extends关键字实现继承。子类必须在constructor方法中调用super方法，否则新建实例报错。因为子类没有自己的this对象，而是继承了父类的this对象，然后对其进行加工。如果不调用super方法，子类得不到this对象。

super关键字指代父类的实例，即父类的this对象。在子类构造函数中，调用super后，才可使用this关键字，否则报错。

## 6、setTimeout、Promise、Async/Await 的区别

setTimeout:执行时先执行当前代码，如果发现setTimeout，将该函数放到浏览器的待执行事件队列里面，等浏览器执行完当前代码，再来执行该函数。

Promise:本身是**同步**的立即执行函数，当在executor中执行resolve或者reject的时候, 此时是异步操作，会先执行then/catch，当主栈完成后，才会去调用resolve/reject中存放的方法执行。

[async](https://so.csdn.net/so/search?q=async&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/weixin_38404899/article/details/_blank):函数返回一个 Promise 对象，当函数执行的时候，一旦遇到 await就会先返回，等到触发的异步操作完成，再执行函数体内后面的语句。可以理解为，是让出了线程，跳出了 async 函数体。

## 7、定时器的执行顺序或机制？

因为js是单线程的，浏览器遇到setTimeout或者setInterval会先执行完当前的代码块，在此之前会把定时器推入浏览器的待执行事件队列里面，等到浏览器执行完当前代码之后会看一下事件队列里面有没有任务，有的话才执行定时器的代码。所以即使把定时器的时间设置为0还是会先执行当前的一些代码。

## 8、map和parseInt结果

Function test(){

var aa = 0;

var testSet = setInterval(function(){ aa++;

if(aa<10){ clearInterval(testSet); } },20);

var testSet1 = setTimeout(function(){  },1000);

for(var i=0;i<10;i++){console.log(test); }}

test()输出结果：

test //10次undefined

['1','2','3'].map(parseInt) 输出什么,为什么?

参考答案输出：[1, NaN, NaN]

首先让我们回顾一下，map函数的第一个参数callback：

var new\_array = arr.map(function callback(currentValue, index, array)

这个callback一共可以接收三个参数，其中第一个参数代表当前被处理的元素，而第二个参数代表该元素的索引。

而parseInt则是用来解析字符串的，使字符串成为指定基数的整数。parseInt(string, radix)接收两个参数，第一个表示被处理的值（字符串），第二个表示为解析时的基数（index）,第三个返回结果数组

了解这两个函数后，我们可以模拟一下运行情况parseInt('1', 0) //radix为0时，且string参数不以“0x”和“0”开头时，按照10为基数处理。这个时候返回1 基数要大于处理值才会解析

parseInt('2', 1) //基数为1（1进制）表示的数中，最大值小于2，所以无法解析，返回NaN

parseInt('3', 2) //基数为2（2进制）表示的数中，最大值小于3，所以无法解析，返回NaN

map函数返回的是一个数组，所以最后结果为[1, NaN, NaN]

## 9、Doctype作用? 严格模式与混杂模式如何区分？它们有何意义?

Doctype声明于文档最前面，告诉浏览器以何种方式来渲染页面，这里有两种模式，严格模式和混杂模式。

严格模式的排版和JS运作模式是以该浏览器支持的最高标准运行。

混杂模式，向后兼容，模拟老式浏览器，防止浏览器无法兼容页面。

## 10、fetch发送2次请求的原因

fetch发送post请求的时候，总是发送2次，第一次状态码是204，第二次才成功？

原因很简单，因为你用fetch的post请求的时候，导致fetch 第一次发送了一个Options请求，询问服务器是否支持修改的请求头，如果服务器支持，则在第二次中发送真正的请求。

## 11、HTTPS 握手过程中，客户端如何验证证书的合法性

首先什么是HTTP协议?http协议是超文本传输协议（hyper text transfer protocol），位于tcp/ip四层模型中的应用层；通过请求/响应的方式在客户端和服务器之间进行通信；但是缺少安全性，http协议信息传输是通过**明文**的方式传输，不做任何加密，相当于在网络上裸奔；容易被中间人恶意篡改，这种行为叫做中间人攻击；

加密通信：为了安全性，双方可以使用**对称加密**的方式**key**进行信息交流，但是这种方式对称加密秘钥也会被拦截，也不够安全，进而还是存在被中间人攻击风险；

于是人们又想出来另外一种方式，使用**非对称加密**的方式；使用**公钥/私钥**加解密；通信方A发起通信并携带自己的公钥，接收方B通过公钥来加密对称秘钥；然后发送给发起方A；A通过**私钥**解密；双发接下来通过对称秘钥来进行加密通信；但是这种方式还是会存在一种安全性；中间人虽然不知道发起方A的私钥，但是可以做到偷天换日，将拦截发起方的公钥key;并将自己生成的一对公/私钥的公钥发送给B；接收方B并不知道公钥已经被偷偷换过；按照之前的流程，B通过公钥加密自己生成的对称加密秘钥key2;发送给A；这次通信再次被中间人拦截，尽管后面的通信，两者还是用key2通信，但是中间人已经掌握了Key2;可以进行轻松的加解密；还是存在被中间人攻击风险；

解决困境：**权威的证书颁发机构CA**来解决；制作证书：作为服务端的A，首先把自己的公钥key1发给证书颁发机构，向证书颁发机构进行申请证书；证书颁发机构有一套自己的公私钥，CA通过自己的私钥来加密key1,并且通过服务端网址等信息生成一个证书签名，证书签名同样使用机构的私钥进行加密；制作完成后，机构将证书发给A；校验证书真伪：当B向服务端A发起请求通信的时候，A不再直接返回自己的公钥，而是返回一个**证书**；说明：各大浏览器和操作系统已经维护了所有的权威证书机构的名称和公钥。B只需要知道是哪个权威机构发的证书，使用对应的机构公钥，就可以解密出证书签名；接下来，B使用同样的规则，生成自己的证书签名，如果两个签名是一致的，说明证书是有效的；签名验证成功后，B就可以再次利用机构的公钥，解密出A的公钥key1;接下来的操作，就是和之前一样的流程了；

中间人是否会拦截发送假证书到B呢？因为证书的签名是**由服务器端网址**等信息生成的，并且通过第三方机构的私钥加密中间人无法篡改；所以最关键的问题是证书签名的真伪；

https主要的思想是在http基础上增加了**ssl安全层(security layer)**，即以上认证过程；

## 12、TCP三次握手（连接）和四次挥手（关闭）

三次握手之所以是三次是保证client和server均让对方知道自己的接收和发送能力没问题而保证的最小次数。

第一次client => server 只能server判断出client具备发送能力 第二次 server => client client就可以判断出server具备发送和接受能力。

此时client还需让server知道自己接收能力没问题于是就有了第三次,第三次 client => server 双方均保证了自己的接收和发送能力没有问题。

第四次握手（挥手），只有等服务端所有的报文的发送完了，服务端才能发送报文FIN，server=>client.

其中，为了保证后续的握手是为了应答上一个握手，每次握手都会带一个标识 seq发送端，后续的ACK接受端都会对这个seq进行加一来进行确认。

## 13、img iframe script 来发送跨域请求有什么优缺点？

iframe优点：跨域完毕之后DOM操作和互相之间的JavaScript调用都是没有问题的

缺点：1.若结果要以URL参数传递，这就意味着在结果数据量很大的时候需要**分割传递**，巨烦。2.还有一个是iframe本身带来的，母页面和iframe本身的交互本身就有**安全性**限制。

script优点：可以直接返回json格式的数据，方便处理

缺点：只接受**GET**请求方式

图片img优点：可以访问任何url，一般用来进行点击追踪，做页面分析常用的方法

缺点：不能访问响应文本，只能监听是否响应

## http和https的区别？五点

http传输的数据都是**未加密**的，也就是**明文**的，网景公司设置了SSL协议来对http协议传输的数据进行加密处理，简单来说https协议是由http和ssl协议构建的可进行加密传输和身份认证的网络协议，比http协议的**安全性**更高。主要的区别如下：

Https协议需要ca证书，费用较高。http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。使用不同的**链接**方式，**端口**也不同，一般而言，http协议的端口为80，https的端口为443,http的连接很简单，是**无状态**的；HTTPS协议是由**SSL+HTTP**协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

## 15、什么是Bom？有哪些常用的Bom属性？

Bom是浏览器对象

location对象

location.href-- 返回或设置当前文档的URL

location.search -- 返回URL中的查询字符串部分。例如 http://www.dreamdu.com/dreamd... 返回包括后面的内容?**id=5&name=dreamdu**

location.hash -- 返回URL#后面的内容，如果没有#，返回空 location.host -- 返回URL中的域名部分，例如www.dreamdu.com

location.hostname -- 返回URL中的主域名部分，例如dreamdu.com

location.pathname -- 返回URL的域名后的部分。例如 http://www.dreamdu.com/xhtml/ 返回/xhtml/

location.port -- 返回URL中的端口部分。例如 http://www.dreamdu.com:8080/xhtml/ 返回8080

location.protocol -- 返回URL中的协议部分。例如 http://www.dreamdu.com:8080/xhtml/ 返回(//)前面的内容http:location.assign -- 设置当前文档的URL

location.replace() -- 设置当前文档的URL，并且在history对象的地址列表中移除这个URL location.replace(url);

location.reload() -- 重载当前页面

history对象

history.go() -- 前进或后退指定的页面数history.go(num); history.back() -- 后退一页

history.forward() -- 前进一页Navigator对象

navigator.userAgent -- 返回用户代理头的字符串表示(就是包括浏览器版本信息等的字符串)

navigator.cookieEnabled -- 返回浏览器是否支持(启用)cookie6

## 16、Cookie、sessionStorage、localStorage的区别

共同点：都是保存在浏览器端，并且是同源的

Cookie：cookie数据始终在同源的http请求中携带（即使不需要），即cookie在浏览器和**服务器**间来回传递。

sessionStorage和localStorage不会自动把数据发给服务器，仅在**本地**保存。cookie数据还有路径（path）的概念，可以限制cookie只属于某个路径下,**存储的大小**很小只有4K左右。（key：可以在浏览器和服务器端来回传递，存储容量小，只有大约4K左右）。sessionStorage：仅在当前**浏览器窗口**关闭前有效，自然也就不可能**持久**保持，localStorage：始终有效，窗口或浏览器关闭也一直保存，因此用作持久数据；cookie只在设置的cookie**过期时间**之前一直有效，即使窗口或浏览器关闭。（key：本身就是一个**回话**过程，关闭浏览器后消失，session为一个回话，当页面不同即使是同一页面打开两次，也被视为同一次回话）

localStorage：localStorage 在所有同源窗口中都是共享的；cookie也是在所有同源窗口中都是共享的。（key：同源窗口都会共享，并且不会失效，不管窗口或者浏览器关闭与否都会始终生效）

补充说明一下cookie的作用：

**保存用户登录状态**。例如将用户id存储于一个cookie内，这样当用户下次访问该页面时就不需要重新登录了，现在很多论坛和社区都提供这样的功能。cookie还可以设置**过期时间**，当超过时间期限后，cookie就会自动消失。因此，系统往往可以提示用户保持登录状态的时间：常见选项有一个月、三个 月、一年等。**跟踪用户行为**。例如一个天气预报网站，能够根据用户选择的地区显示当地的天气情况。如果每次都需要选择所在地是烦琐的，当利用了 cookie后就会显得很人性化了，系统能够记住上一次访问的地区，当下次再打开该页面时，它就会自动显示上次用户所在地区的天气情况。因为一切都是在后台完成，所以这样的页面就像为某个用户所定制的一样，使用起来非常方便定制页面。如果网站提供了换肤或更换布局的功能，那么可以使用cookie来记录用户的选项，例如：背景色、分辨率等。当用户下次访问时，仍然可以保存上一次访问的界面风格。

## 17、Cookie如何防范XSS攻击

XSS（跨站脚本攻击）是指攻击者在返回的HTML中嵌入javascript脚本，为了减轻这些攻击，需要在HTTP头部配上，**set-cookie**：

**httponly**-这个属性可以防止XSS,它会禁止javascript脚本来访问cookie。**secure** - 这个属性告诉浏览器仅在请求为https的时候发送cookie。结果应该是这样的：Set-Cookie=.....

## 18、浏览器和 Node 事件循环的区别？

其中一个主要的区别在于浏览器的**event loop** 和nodejs的event loop 在处理异步事件的顺序是不同的,nodejs中有micro event;其中Promise属于micro event 该异步事件的**处理顺序**就和浏览器不同.

nodejs V11.0以上 这两者之间的顺序就相同了.

## 19、简述HTTPS中间人攻击

https协议由 http + ssl 协议构成，具体的链接过程可参考SSL或TLS握手的概述

中间人攻击过程如下：

服务器向客户端发送公钥。攻击者截获公钥，保留在自己手上。然后攻击者自己生成一个【伪造的】公钥，发给客户端。客户端收到伪造的公钥后，生成加密hash值发给服务器。攻击者获得加密hash值，用自己的私钥解密获得真秘钥。同时生成假的加密hash值，发给服务器。服务器用私钥解密获得假秘钥。服务器用加秘钥加密传输信息防范方法：

服务端在发送浏览器的公钥中加入CA证书，浏览器可以验证CA证书的有效性

## 20、说几条web前端优化策略

**(1). 减少HTTP请求数**

这条策略基本上所有前端人都知道，而且也是最重要最有效的。都说要减少HTTP请求，那请求多了到底会怎么样呢？首先，每个请求都是有成本的，既包 含时间成本也包含资源成本。一个完整的请求都需要经过DNS寻址、与服务器建立连接、发送数据、等待服务器响应、接收数据这样一个“漫长”而复杂的过程。时间成本就是用户需要看到或者“感受”到这个资源是必须要等待这个过程结束的，资源上由于每个请求都需要携带数据，因此每个请求都需要占用带宽。

另外，由于浏览器进行**并发请求**的**请求数**是有上限的，因此请求数多了以后，浏览器需要**分批**进行请求，因此会增加用户的**等待时间**，会给用户造成站点速度慢这样一个印象，即使可能用户能看到的第一屏的资源都已经请求完了，但是浏览器的进度条会一直存在。减少HTTP请求数的主要途径包括：

**(2). 从设计实现层面简化页面**

如果你的页面像百度首页一样简单，那么接下来的规则基本上都用不着了。保持页面简洁、减少资源的使用时最直接的。如果不是这样，你的页面需要华丽的皮肤，则继续阅读下面的内容。

**(3). 合理设置HTTP缓存**

缓存的力量是强大的，恰当的缓存设置可以大大的减少HTTP请求。以有啊首页为例，当浏览器没有缓存的时候访问一共会发出78个请求，共600多K 数据（如图1.1），而当第二次访问即浏览器已缓存之后访问则仅有10个请求，共20多K数据（如图1.2）。（这里需要说明的是，如果直接F5刷新页面 的话效果是不一样的，这种情况下请求数还是一样，不过被缓存资源的请求服务器是**304**响应，只有Header没有Body，可以节省带宽）

怎样才算合理设置？原则很简单，**能缓存越多越好，能缓存越久越好**。例如，很少变化的图片资源可以直接通过HTTP Header中的Expires设置一个很长的**过期头**；变化不频繁而又可能会变的资源可以使用Last-Modifed来做请求验证。尽可能的让资源能够 在缓存中待得更久。

**(4). 资源合并与压缩**

如果可以的话，尽可能的将外部的脚本、样式进行合并，多个合为一个。另外，CSS、Javascript、Image都可以用相应的工具进行压缩，压缩后往往能省下不少空间。

**(5). CSS Sprites**

合并CSS图片，减少请求数的又一个好办法。

**(6). Inline Images**

使用data: URL scheme的方式将图片嵌入到页面或**CSS**中，如果不考虑资源管理上的问题的话，不失为一个好办法。如果是嵌入页面的话换来的是增大了页面的体积，而且无法利用浏览器缓存。使用在CSS中的图片则更为理想一些。

**(7). Lazy Load Images**

这条策略实际上并不一定能减少HTTP请求数，但是却能在某些条件下或者页面刚加载时减少HTTP请求数。对于图片而言，在页面刚加载的时候可以只 加载第一屏，当用户继续往后滚屏的时候才加载后续的图片。这样一来，假如用户只对第一屏的内容感兴趣时，那剩余的图片请求就都节省了。有啊首页曾经的做法 是在加载的时候把第一屏之后的图片地址缓存在Textarea标签中，待用户往下滚屏的时候才“惰性”加载。

## 21、浏览器的重绘和回流导致的性能问题

重绘（Repaint）和回流（Reflow）

重绘和回流是渲染步骤中的一小节，但是这两个步骤对于性能影响很大。

**重绘是当节点需要更改外观而不会影响布局的**，比如改变 color就叫称为重绘

**回流是布局或者几何属性需要改变就称为回流**。回流必定会发生重绘，重绘不一定会引发回流。回流所需的成本比重绘高的多，改变深层次的节点很可能导致父节点的一系列回流。

所以以下几个动作可能会导致性能问题：

改变 window 大小改变字体添加或删除样式文字改变定位或者浮动盒模型很多人不知道的是，重绘和回流其实和 Event loop 有关。

当 Event loop 执行完 Microtasks 后，会判断 document 是否需要更新。因为浏览器是 **60Hz** 的刷新率，每 16ms 才会更新一次。然后判断是否有 resize或者 scroll，有的话会去触发事件，所以 resize和 scroll事件也是至少 16ms 才会触发一次，并且自带**节流**功能。判断是否触发了 media query更新动画并且发送事件判断是否有全屏操作事件执行 requestAnimationFrame回调执行 IntersectionObserver回调，该方法用于判断元素是否可见，可以用于懒加载上，但是兼容性不好更新界面以上就是一帧中可能会做的事情。如果在一帧中有空闲时间，就会去执行 requestIdleCallback回调。

减少重绘和回流

使用 translate 替代 top

<div class="test"></div>

<style> .test { position: absolute; top: 10px; width: 100px; height: 100px; background: red; }</style>

<script> setTimeout(() => { // 引起回流document.querySelector('.test').style.top = '100px' }, 1000)</script>

使用 visibility替换 display: none，因为前者只会引起重绘，后者会引发回流（改变了布局）把 DOM 离线后修改，比如：先把 DOM 给 display:none(有一次 Reflow)，然后你修改100次，然后再把它显示出来,不要把 DOM 结点的属性值放在一个循环里当成循环里的变量for(let i = 0; i < 1000; i++) { // 获取 offsetTop 会导致回流，因为需要去获取正确的值 console.log(document.querySelector('.test').style.offsetTop)}不要使用 table 布局，可能很小的一个小改动会造成整个 table 的重新布局动画实现的速度的选择，动画速度越快，回流次数越多，也可以选择使用 requestAnimationFrameCSS 选择符从右往左匹配查找，避免 DOM 深度过深将频繁运行的动画变为图层，图层能够阻止该节点回流影响别的元素。比如对于 video标签，浏览器会自动将该节点变为图层。

## 22、写 React / Vue 项目时为什么要在列表组件中写 key，其作用是什么？

vue和react都是采用diff算法来对比新旧虚拟节点，从而更新节点。在vue的diff函数中（建议先了解一下diff算法过程）。在交叉对比中，当新节点跟旧节点头尾交叉对比没有结果时，会根据新节点的key去对比旧节点数组中的key，从而找到相应旧节点（这里对应的是一个key => index 的map映射）。如果没找到就认为是一个新增节点。而如果没有key，那么就会采用遍历查找的方式去找到对应的旧节点。一种一个map映射，另一种是遍历查找。相比而言。map映射的速度更快。

## 23、为什么虚拟dom会提高性能?

虚拟dom相当于在js和真实dom中间加了一个缓存，利用dom diff算法避免了没有必要的dom操作，从而提高性能。

具体实现步骤如下：

用 JavaScript 对象结构表示 DOM 树的结构；然后用这个树构建一个真正的 DOM 树，插到文档当中

当状态变更的时候，重新构造一棵新的对象树。然后用新的树和旧的树进行比较，记录两棵树差异

把两所记录的差异应用到步骤1所构建的真正的DOM树上，视图就更新了。

## 24、为什么多个域名来存储网站资源会更有效

1. CDN(content distribution network)缓存更方便。

2. 突破浏览器**并发限制**。

3. **Cookieless, 节省带宽**，尤其是上行带宽 一般比下行要慢。

还有另外两个非常规原因:

4. **对于UGC(user-generated content)的内容和主站隔离，防止不必要的安全问题**。

正是这个原因要求用户内容的域名必须不是自己主站的子域名，而是一个完全独立的第三方域名。

5. 数据做了划分，甚至切到了不同的物理集群，**通过子域名来分流比较省事**.  这个可能被用的不多。

## 25、web开发中会话跟踪的四种方式

1.什么是会话

**客户端打开与服务器**的连接发出**请求**到服务器**响应客户端请求**的全过程称之为会话 。

2.什么是会话跟踪

对同一个用户对服务器的连续的请求和接受响应的**监视** 。

3.为什么需要会话跟踪

浏览器与服务器之间的通信是通过**HTTP协议进行通信**的，而**HTTP协议是”无状态”的协议**，它不能保存客户的信息，即一次响应完成之后连接就断开了，下一次的请求需要重新连接，**这样就需要判断是否是同一个用户，所以才应会话跟踪技术来实现这种要求**

4.介绍

当服务器响应客户端的第一次请求时，将会创建一个新的**session对象**(该对象实现了HttpSession接口)和一个唯一的ID分配给该请求，以后 客户将此会话ID与请求一起传给服务器，此会话ID在后续的请求中会将用户与session对象进行匹配，用于识别不同的客户。

**5.会话跟踪常用的方法:**

①　**URL重写**

URL(统一资源定位符)是Web上特定页面的地址，URL重写的技术就是在**URL结尾添加一个附加数据以标识该会话**,**把会话ID**通过URL的信息传递过去，以便在服务器端进行识别不同的用户 。

如果用户没有禁用cookie，而且又使用URL重写，则：用户在第一次访问EncodeURLServlet时，由于不知道用户是否禁用了 cookie，所以response.encodeURL()方法内部会将JSESSIONID重写在url上，但是一旦第二次访问时，由于用户是带着 cookie来的，所以response.encodeURL()不会将JSESSIONID重写在url上。

但是如果用户禁用cookie，则关闭了浏览器后，重新开启浏览器，则回话失效，无法实现回话跟踪；如果是用户没有禁用cookie，则可以通过设置装载JSESSIONID的cookie的失效时间来控制浏览器关闭后session仍未失效。

**②　隐藏表单域**

**将会话ID添加到HTML表单元素中**提交到服务器,此表单元素并不在客户端显示 。

**③　Cookie**

**Cookie是Web服务器发送给客户端的一小段信息**，客户端请求时可以读取该信息发送到服务器端，进而进行**用户的识别**。对于客户端的每次请求，服务器都会将Cookie发送到客户端,在客户端可以进行保存,以便下次使用。

客户端可以采用两种方式来保存这个**Cookie对象**，一种方式是保存在客户端内存中，称为**临时Cookie**，**浏览器关闭后 这个Cookie对象将消失**。另外一种方式是保存在 客户机的**磁盘上，称为永久Cookie**。以后客户端只要访问该网站，就会将这个Cookie再次发送到服务器上，**前提是 这个Cookie在有效期内**。这样就实现了对客户的跟踪。

**Cookie是可以被禁止的。**

//在服务器端获取Cookie

Cookie[]cookies = request.getCookies();

for (int i = 0; cookies != null && i < cookies.length;i++) {

Cookie[]cookie = cookies[i];

System.put.println(cookie.getName());

}

//在服务器端设置Cookie

String username = "sa";

Cookie usernameCookie = new Cookie("username", username);

response.addCookie(usernameCookie);

如果maxAge为负数，则表示该Cookie仅在本浏览器窗口以及本窗口打开的子窗口内有效，关闭窗口后该Cookie即失效。maxAge为负数的Cookie，**为临时性Cookie，不会被持久化**，不会被写入到**Cookie文件**中。Cookie信息保存在浏览器内存中，因此关闭浏览器该Cookie就消失了。

**Cookie默认的maxAge值为-1。**

**如果maxAge为0，则表示删除该Cookie**。Cookie机制没有提供删除Cookie的方法，因此通过设置该Cookie**即时失效**实现删除Cookie的效果。失效的Cookie会被浏览器从Cookie文件或者内存中删除。

response对象提供的Cookie操作方法只有一个添加操作add(Cookie cookie)。要想修改Cookie只能使用一个同名的Cookie来覆盖原来的Cookie，达到修改的目的。删除时只需要把maxAge修改为0即可。

Cookie并不提供修改、删除操作。如果要修改某个Cookie，只需要新建一个同名的Cookie，并添加到response中覆盖原来的Cookie。

如果要**删除某个Cookie，只需要新建一个同名的Cookie**，并将maxAge设置为0，并添加到**response中覆盖原来的Cookie。**

④　Session

**每一个用户都有一个不同的session**，各个用户之间是不能共享的，是每个用户所独享的，在**session中可以存放信息**。在服务器端会创建一个session对象，产生一个sessionID来标识这个session对象，然后将这个sessionID放入到Cookie中发送到客户端，下一次访问时，sessionID会发送到服务器，在服务器端进行识别不同的用户。**Session是依赖Cookie的**，如果Cookie被禁用，那么session也将失效。因为Session是用Session ID来确定当前对话所对应的服务器Session，而**Session ID是通过Cookie来传递的**，禁用Cookie相当于失去了Session ID，也就得不到Session了。此时可以考虑URL重写和表单隐藏域。

原理：在后台，**系统从cookie或者url中提取用户ID**，这些动作对程序员是透明的，你只需调用getSession，

如果在输入的cookie和url中找不到会话ID，系统会创建一个**新的空会话**。如果使用的是cookie（默认），系统还会创建一个名为

JSESSIONID的输出cookie，在其中存储一个唯一的值表示会话ID。（在服务器端读取的时候就是通过这个来**识别Session的**）

如果不存在会话的时候**不想创建新的会话**，可以使用HttpSession session = request.getSession(false);，如果不存在session对象就得到一个null值，并不创建空的session对象。

removeAttribute(“key”)　　废弃与指定键关联的值

invalidate删除整个会话（在当前web应用中）

logout将用户从web服务器中注销，并删除所有属于他的会

HttpSession session = request.getSession();

设置session的空闲失效时间有两种方式：

1.在web.xml中设置:

<session-config>

<session-timeout>2</session-timeout>

</session-config>

2.设置session的maxInactiveInterval属性，（session的 默认空闲失效时间为30分钟，即30分钟内不访问服务器，session将自动销毁）如下:

session.setMaxInactiveInterval(2\*60);

也可以通过 session.invalidate();销毁session。

## 25、[JS里给数组增加数据的方式](https://www.cnblogs.com/ayaa/p/14732349.html)

第1种：直接给数组的下标通过赋值来增加（数组的下标起始值是0）；

第2种：通过 数组名[数组名.length] 来增加；

第3种：通过 数组名.push(参数) 来增加从数组最后一个数据开始增加，push可以带多个参，带几个参，数组最后就增加几个数据；

第4种：通过 数组名.unshift(参数)来增加从数组第1个数据开始的参数，unshift可以带多个参，带几个参，数组最开始就增加几个数据；

第5种：用 数组名.splice(开始插入的下标数,0,需要插入的参数1,需要插入的参数2,需要插入的参数3……)来增加数组数据；

# CSS

## 24、分析比较 opacity: 0、visibility: hidden、display: none 优劣和适用场景

**结构**：display:none: 会让元素完全从渲染树中**消失**，渲染的时候不占据任何**空间**, 不能**点击**

visibility: hidden:不会让元素从渲染树消失，渲染元素继续占据空间，只是内容不可见，不能点击

opacity: 0: 不会让元素从渲染树消失，渲染元素继续占据空间，只是内容不可见，可以点击

**继承**：display: none：是**非继承**属性，子孙节点消失由于元素从渲染树消失造成，通过修改子孙节点**属性**无法显示。

visibility: hidden：是继承属性，子孙节点消失由于继承了hidden，通过设置visibility: visible;可以让子孙节点显式。

**性能**：display:none : 修改元素会造成文档**回流**,读屏器**不会读取**display: none元素内容，**性能消耗**较大

visibility:hidden: 修改元素只会造成本元素的**重绘**,性能消耗较少,读屏器读取visibility: hidden元素内容

opacity: 0 ：修改元素会造成重绘，性能消耗较少

联系：它们都能让元素不可见

## 25、清除浮动的方式有哪些?比较好的是哪一种?

常用的一般为三种.clearfix,clear:both,overflow:hidden;

比较好是 .clearfix,伪元素万金油版本,后两者有局限性.

.clearfix:after {

visibility: hidden;

display: block;

font-size: 0;

content: " ";

clear: both;

height: 0;}<!--为毛没有 zoom ,\_height 这些,IE6,7这类需要 csshack 不再我们考虑之内了

.clearfix 还有另外一种写法 ,-->

.clearfix:before,.clearfix:after { content:"";display:table;}

.clearfix:after{ clear:both; overflow:hidden;}

.clearfix{ zoom:1;}<!--用display:table 是为了避免外边距margin重叠导致的margin塌陷,内部元素默认会成为 table-cell 单元格的形式-->

clear:both:若是用在同一个容器内相邻元素上,那是贼好的,有时候在容器外就有些问题了, 比如相邻容器的包裹层元素塌陷

overflow:hidden:这种若是用在同个容器内,可以形成 BFC避免浮动造成的元素塌陷

## 26、css sprite 是什么,有什么优缺点

概念：将多个小图片拼接到一个图片中。通过 **background-position** 和元**素尺寸调节**需要显示的**背景**图案。

优点：

**减少 HTTP 请求数**，极大地提高页面**加载速度，**增加图片**信息重复度，**提高**压缩比**，减少**图片大小**，更换**风格**方便，只需在一张或几张图片上修改颜色或样式即可实现

缺点：

**图片合并麻烦维护麻烦**，修改一个图片可能需要重新布局整个图片

## 27、link与@import的区别

link是 HTML 方式， @import是 CSS 方式，

link最大限度支持**并行**下载，@import过多嵌套导致**串行**下载，出现FOUC，link可以通过rel="alternate stylesheet"指定候选样式，浏览器对link**支持**早于@import，可以使用@import对老浏览器隐藏样式，@import必须在样式规则之前，可以在 **css 文件中引用**其他文件

总体来说：link 优于@import

## 28、display: block;和display: inline;的区别

block元素特点：

1. 处于常规流中时，如果width没有设置，会自动填充满父容器
2. 可以应用margin/padding
3. 在没有设置高度的情况下会扩展高度以包含常规流中的子元素

4.处于常规流中时布局时在前后元素位置之间（独占一个水平空间） 5.忽略vertical-align

inline元素特点

1.水平方向上根据direction依次布局

2.不会在元素前后进行换行

3.受white-space控制

4.margin/padding在**竖直**方向上无效，**水平**方向上有效

5.width/height属性对非替换行内元素无效，**宽度由元素内容决定**

6.非替换行内元素的行框高由**line-height**确定，替换行内元素的行框高由**height,margin,padding,border**决定

**7.浮动或绝对定位时会转换为block**

8.**vertical-align**属性生效

## 28、block，inline和inlinke-block细节对比

**display:block**

block元素会独占一行，多个block元素会各自新起一行。默认情况下，block元素宽度自动填满其父元素宽度。

block元素可以**设置width,height属性**。块级元素即使设置了宽度,仍然是**独占一行**。

block元素可以设置**margin和padding属性**。

**display:inline**

inline元素不会独占一行，多个相邻的行内元素会排列在同一行里，直到一行排列不下，才会新换一行，**其宽度随元素的内容而变化**。

inline元素设置**width,height属性无效**。

inline元素的margin和padding属性，**水平方向**的padding-left, padding-right, margin-left, margin-right都产生**边距效果**；但竖直方向的padding-top, padding-bottom, margin-top, margin-bottom不会产生边距效果。

**display:inline-block**

简单来说就是将对象呈现为inline对象，但是对象的内容作为block对象呈现。之后的内联对象会被排列在同一行内。比如我们可以给一个link（a元素）inline-block属性值，**使其既具有block的宽度高度特性又具有inline的同行特性。**

## 29、容器包含若干浮动元素时如何清理浮动

容器元素闭合标签前添加额外元素并设置clear: both，父元素触发块级格式化上下文(见块级可视化上下文部分)设置容器元素伪元素进行清理推荐的清理浮动方法，在标准浏览器下使用 content内容为空格用于修复opera下文档中出现contenteditable属性时，在清理浮动元素上下的空白，使用display使用table而不是block：可以防止容器和子元素top-margin折叠,这样能使清理效果与BFC，IE6/7\* zoom: 1;一致

.clearfix:before,.clearfix:after {

content: " ";

display: table; }

.clearfix:after { clear: both;}/ IE 6/7下使用，

通过触发hasLayout实现包含浮动

.clearfix { zoom: 1;}

第二中方法：

.clearfix:after{

dispplay:block;

visibility:hidden;

content:””;

font-size:0;

clear:both;

}

## 30、PNG,GIF,JPG 的区别及如何选

GIF:

8 位像素，256 色**无损压缩**支持简单动画**支持 boolean 透明，**适合简单**动画**

JPEG：

颜色限于 256**有损压缩**可控制压缩质量不支持透明，适合**照片**

PNG：

有 PNG8 和 truecolor PNG，PNG8 类似 GIF 颜色上限为 256，**文件小，支持 alpha 透明度**，**无动画，**适合**图标、背景、按钮**

## 31、display,float,position 的关系

如果display为 none，那么 position 和 float 都不起作用，这种情况下元素不产生框，否则，如果 position 值为 absolute 或者 fixed，框就是绝对定位的，float 的计算值为 none，display 根据下面的表格进行调整。

否则，如果 float 不是 none，框是浮动的，display 根据下表进行调整。否则，如果元素是根元素，display 根据下表进行调整。其他情况下 display 的值为指定值。总结起来：绝对定位、浮动、根元素都需要调整display

## 32、如何水平居中一个元素

如果需要居中的元素为常规流中 inline 元素，为父元素设置text-align: center;即可实现。

如果需要居中的元素为常规流中 block 元素，1）为元素设置宽度，2）设置左右 margin 为 auto。

IE6 下需在父元素上设置text-align: center;,再给子元素恢复需要的值

<body>

<div class="content"> aaaaaa aaaaaa a a a a a  </div></body>

<style>

body { background: #DDD; text-align: center;  }

.content { width: 500px;

text-align: left;

margin: 0 auto;

background: purple; }

</style>

如果需要居中的元素为浮动元素，1）为元素设置宽度，2）position: relative;，3）浮动方向偏移量（left 或者 right）设置为 50%，4）浮动方向上的 margin 设置为元素宽度一半乘以-1

<body>

<div class="content">aaaaaa aaaaaa a a a a a a a </div>

</body>

<style>

body {background: #DDD; }

.content {

width: 500px;

float: left;

position: relative;

left: 50%;

margin-left: -250px;

background-color: purple; }</style>

如果需要居中的元素为绝对定位元素，1）为元素设置宽度，2）偏移量设置为 50%，3）偏移方向外边距设置为元素宽度一半乘以-1

<body>

<div class="content"> aaaaaa aaaaaa a a a a a a a a </div>

</body>

<style>

body {

background: #DDD;

position: relative; }

.content {

width: 500px;

float: left;

position: absolute;

left: 50%;

margin-left: -250px;

background-color: purple; }

</style>

项目中Toast组件居中

*.toast*{

    position: fixed;

    top:50%;

    left:50%;

    transform: translate(-50%,-50%);

    padding: 8px 10px;

    color:#fff;

    background-color: rgba(0,0,0,.8);

    z-index: 9999;

  }

## position中fixed、relative、absolute区别

relative：生成相对定位的元素，相对于**其正常位置**进行定位。

元素的位置通过left、right、top、button属性进行规定，

可以通过z-index进行层次分级。

元素元素仍保持其未定位前的形状，**原本所占的空间仍将保留。**

如果没有定位偏移量，对元素本身没有任何影响

absolute：生成绝对定位元素。使元素**脱离文档流**，并相对于其[包含块](https://link.zhihu.com/?target=http://www.webforend.com/a/CSSjiaocheng/2016/0712/46.html" \t "_blank)进行定位，包含块可能是文档中的另一个元素或者是初始包含块，

元素原先在正常文档流中所占的空间会会被后面元素占据；

元素定位后生成一个**块级框**，而不论原来它在正常流中生成何种类型的框；

绝对定位元素的包含块由离它最近的 'position' 属性为 'absolute'、**'relative' 或者 'fixed' 的祖先元素**创建。只要父级元素设置了position并且不是static（默认既是static），那么设定了absolute的子元素即以此为包含块（最近的）。如果都没有定义，那么就相对于整个**文档body**定位（注意不是相对于浏览器窗口定位）

**相对定位一般都是配合绝对定位元素使用**

fixed：生成绝对定位元素，相对于**浏览器窗口的定位**。通常配合z-index一起来使用。比如说网页上悬挂的聊天图标或者广告就是用了fixed

## 33、[position的5种定位方式](https://www.cnblogs.com/wang--chao/p/14654111.html)

1.static

position的默认值，没有定位，元素出现在正常的文档流中

2.relative

相对定位，相对于正常位置定位，不会脱离文档流

3.absolute

绝对定位，相对于最近的定位不为static的父元素进行定位。脱离了文档流

4.fixed

固定定位，相对于浏览器窗口定位，不会跟随屏幕滚动。生成绝对定位，脱离文档流

5.sticky

粘性定位，它的行为就像 position:relative和position:fixed的合体，当页面滚动超出目标区域时，它会固定在设定位置。

## 34、margin-top和top的区别

一、功能不同

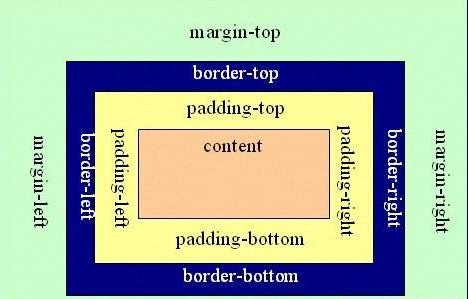
1、margin-top：功能为设置元素的上外边距。

2、top：规定元素的顶部边缘。

二、特点不同

1、margin-top：允许使用负值。定义固定的上外边距。默认值是 0。

2、top：定义了一个定位元素的上外边距边界与其包含块上边界之间的偏移。

[](https://iknow-pic.cdn.bcebos.com/d000baa1cd11728b06cbcc5ed8fcc3cec2fd2ce3)

三、规定不同

1、margin-top：所有主流浏览器都支持 margin-top 属性。

2、top：如果 "position" **属性的值为 "static"，那么设置 "top" 属性不会产生任何效果。只针对脱离文档流的属性起作用。**

top是当css样式为绝对定位（即position:absolute; ）时，left ， right ， top ， bottom 等属性相对于其最接近的一个**最有定位设置的父对象**进行绝对定位，比如：position:absolute;top:0px;

absolute定位下的float：left/right是不起作用的，通常我们会在absolute下用top/right/bottom/left来改变他的位置

relative定位之下这几个属性也是可以使用的，在这种情况下这几个属性和margin-top/right/bottom/left的作用是类似的

# JavaScript

## 34、获取页面中元素的方法

　　1）、document.getElementById('元素的ID')，在整个文档中，通过元素的ID获取到这个元素对象(获取的是一个元素)

注意:如果页面中的id重复了，那么这个方法默认只获得第一个元素

在ie6、7中,会把表单元素（input）的name属性值当做ID来使用

在ie6、7中，不区分ID的大小写

项目实战注意:不要让表单元素的name和其他元素的id重复，不要用id的大小写来区分我们的不同元素

如果没有获取到，返回的 结果为null

我们可以直接的用元素的id来代表当前元素对象（不推荐）

　　2）、通过元素的标签名，来获取一组元素（有几个获取几个）

document.getElementsByTagName('元素的标签名')，在整个文档中，通过元素的标签名获取一组元素。获取的是一个对象数据类型结果，并且是一个**类数组**（以数字作为索引，索引从0开始，逐级递增，索引代表的是当前对应的某一个元素，有一个叫做length的属性代表获取的个数）

document称之为上下文(context)，就是我们自己可以限定当前获取元素的范围，[getElementById的上下文只能是document]

方法一：点击某个li，显示它的索引

<ul>

<li>0</li>

<li>1</li>

<li>2</li>

<li>4</li>

</ul>

<script>

**var** obj=document.getElementsByTagName('li');

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

   obj[i].id=i;

   obj[i].onclick=**function**(){

      alert(**this**.id);

   }

}

</script>

方法二：let和var的区别 块级作用域

<ul>

    <li>0</li>

    <li>1</li>

    <li>2</li>

    <li>4</li>

  </ul>

  <script>

    let obj=document.getElementsByTagName("li");

    for(let i=0;i<obj.length;i++){

      obj[i].onclick=function(){

        console.log(i);

      }

    }

  </script>

　　3）、context.getElementsByName()通过元素的name属性的值获取一组元素

在ie浏览器下只对表单元素起作用

这个方法应用于获取具有同样name的表单元素

　　4）、context.getElementsByClassName()，通过元素的类名（class的值） 是项目中最常用的一种方法，但是在ie6-ie8中会报错。获取多个的这几个方法，即使你获取的只有一个，他也是类数组，也是一个集合，如果想用其中的第一个，你也要通过索引拿出来用。

　　5）、document.documentElement //获取HTML元素

document.body//获取body元素

var curWidth = document.documentElement.clientWidth || document.body.clientWidth//兼容所有浏览器的获取当前屏幕的宽度

var curHeight = document.documentElement.clientHeight || document.body.clientHeight//兼容所有浏览器的获取当前屏幕的高度

1. 在移动端我们回去元素常用的方法（IE6~8下不兼容）

　document.querySelector()获取一个  
　document.querySelectorAll()获取多个

补充：class=”ul” .ul id=”ul” #ul

## 35、js中的点击事件click的实现方式

<!-- 第三种方式-->

<button id="btn" onclick="threeFn()">点我</button>

<script type="text/javascript">

var btn = document.getElementById("btn");

// 第一种 通过点击事件

btn.onclick = function(){

alert("这是第一种点击方式");

}

//第二种 监听点击事件

btn.addEventListener('click', function(){

alert("这是第二中点击方式");

})

// 第三种 通过方法响应点击事件

function threeFn(){

alert("这是第三种点击方式");

}

</script>

## 36、JS有几种数据类型,其中基本数据类型有哪些?

七种数据类型

Boolean，Null，Undefined，Number，String，Symbol (ECMAScript 6 新定义)Object(ES6之前)其中5种为基本类型:string,number,boolean,null,undefined,

ES6出来的Symbol也是**原始数据类型** ，表示独一无二的值

Object为**引用类型**(范围挺大),也包括数组、函数,

## 37、Promise 构造函数是同步执行还是异步执行，那么 then 方法呢？

const promise = new Promise((resolve, reject) => { console.log(1)

resolve()

console.log(2)})

promise.then(() => {

console.log(3)})

console.log(4)

输出结果是：1243

**promise构造函数是同步执行的，then方法是异步执行的**

Promise new的时候会立即执行里面的代码 then是**微任务** 会在本次任务执行完的时候执行 setTimeout是**宏任务** 会在下次任务执行的时候执行

38、JS的四种设计模式

**工厂模式**

简单的工厂模式可以理解为解决**多个相似**的问题;

function CreatePerson(name,age,sex)

{ var obj = new Object();

obj.name = name;

obj.age = age;

obj.sex = sex;

obj.sayName = function()

{return this.name;}

return obj;}

var p1 = new CreatePerson("longen",'28','男');

var p2 = new CreatePerson("tugenhua",'27','女');

console.log(p1.name); // longen

console.log(p1.age); // 28

console.log(p1.sex); // 男

console.log(p1.sayName()); // longen

console.log(p2.name); // tugenhua

console.log(p2.age); // 27

console.log(p2.sex); // 女

console.log(p2.sayName()); // tugenhua

**单例模式**

**只能被实例化(构造函数给实例添加属性与方法)一次**

// 单体模式

var Singleton = function(name){

this.name = name;};

Singleton.prototype.getName = function(){

return this.name;}

// 获取实例对象

var getInstance = (function() {

var instance = null;

return function(name) {

if(!instance) {

//相当于一个一次性阀门,只能实例化一次

instance = new Singleton(name); }

return instance; }})();

// 测试单体模式的实例,所以a===b

var a = getInstance("aa");

var b = getInstance("bb");

**沙箱模式**

**return 返回的结果就是函数的值**

将一些函数放到自执行函数里面,但要用闭包暴露接口,**用变量接收暴露的接口,再调用里面的值**,否则无法使用里面的值

let sandboxModel=(function(){

function sayName(){};

function sayAge(){};

return{sayName:sayName, sayAge:sayAge }})()

**发布者订阅模式**

就例如如我们关注了某一个公众号,然后他对应的有新的消息就会给你推送,

//发布者与订阅模式 var shoeObj = {};

// 定义发布者

shoeObj.list = [];

// 缓存列表 存放订阅者回调函数

// 增加订阅者

shoeObj.listen = function(fn) {

shoeObj.list.push(fn);

// 订阅消息添加到缓存列表 }

// 发布消息

shoeObj.trigger = function() {

for (var i = 0, fn; fn = this.list[i++];) {

fn.apply(this, arguments);

//第一个参数只是改变fn的this, } }

// 小红订阅如下消息

shoeObj.listen(function(color, size) {

console.log("颜色是：" + color);

console.log("尺码是：" + size); });

// 小花订阅如下消息

shoeObj.listen(function(color, size) {

console.log("再次打印颜色是：" + color);

console.log("再次打印尺码是：" + size); });

shoeObj.trigger("红色", 40);

shoeObj.trigger("黑色", 42);

代码实现逻辑是用**数组存贮**订阅者, 发布者**回调函数里面通知**的方式是遍历订阅者数组,并将**发布者内容传入订阅者数组**

## 38、列举出集中创建实例的方法

1.**字面量 对象属性加值**

let obj={'name':'张三'}

2.**Object构造函数创建 创建对象赋值**

let Obj=new Object()

Obj.name='张三'

3.使用**工厂模式**创建对象 传参函数 创建对象赋值 调用传参函数

function createPerson(name){

var o = new Object();

o.name = name;

return o; }

var person1 = createPerson('张三');

1. **使用构造函数创建对象 传参函数(构造函数) 调用传参函数(创建对象)**

function Person(name){

this.name = name;}

var person1 = new Person('张三');

## 39、简述一下前端事件流

HTML中与javascript交互是通过**事件驱动**来实现的，例如鼠标点击事件**onclick**、页面的滚动事件**onscroll**等等，可以向文档或者文档中的元素添加**事件侦听器**来预订事件。想要知道这些事件是在什么时候进行调用的，就需要了解一下“事件流”的概念。

什么是事件流：事件流描述的是从页面中接收事件的顺序,DOM2级事件流包括下面几个阶段。

事件捕获阶段处于目标阶段事件**冒泡阶段**addEventListener：**addEventListener**是DOM2 级事件新增的指定事件处理程序的操作，这个方法接收3个参数：要处理的**事件名**、作为事件处理程序的**函数**和一个**布尔值**。最后这个布尔值参数如果是true，表示在**捕获阶段**调用事件处理程序；如果是false，表示在**冒泡阶段**调用事件处理程序。

IE只支持事件冒泡。

## 40、Function.\_proto\_(getPrototypeOf)是什么？

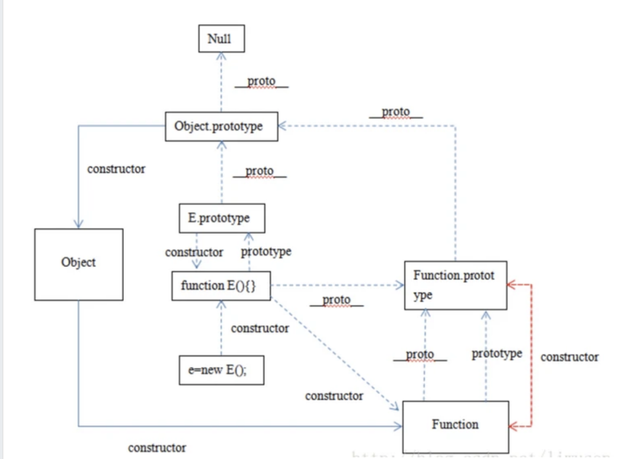
**获取一个对象的原型**，在chrome中可以通过\_\_proto\_\_的形式，或者在ES6中可以通过Object.getPrototypeOf的形式。

那么Function.proto是什么么？也就是说Function由什么对象继承而来，我们来做如下判别。

Function.\_\_proto\_\_==Object.prototype//false Function.\_\_proto\_\_==Function.prototype//true

我们发现Function的原型也是Function。

我们用图可以来明确这个关系：



## 41、简述一下原型 / 构造函数 / 实例

原型(prototype): **一个简单的对象，用于实现对象的属性继承**。可以简单的理解成**对象的爹**。在 Firefox 和 Chrome 中，每个JavaScript对象中都包含一个\_\_proto\_\_(非标准)的属性指向它爹(该对象的原型)，可**obj.\_\_proto\_\_**进行访问。

构造函数: **可以通过new来新建一个对象的函数**。实例: 通过构造函数和new创建出来的对象，便是**实例**。实例通过\_\_proto\_\_指向原型，通过constructor指向构造函数。这里来举个栗子，以Object为例，我们常用的Object便是一个构造函数，因此我们可以通过它构建实例。

// 实例const instance = new Object()则此时，**实例**为instance, **构造函数**为Object，我们知道，构造函数拥有一个prototype的属性指向**原型**，因此原型为:

// 原型const prototype = Object.prototype这里我们可以来看出三者的关系:

实例.\_\_proto\_\_ === 原型.constructor === 构造函数.prototype === 原型

// 这条线其实是基于原型进行获取的，可以理解成一条基于**原型的映射线**

// 例如: // const o = new Object()

// o.constructor === Object --> true

// o.\_\_proto\_\_ = null;

// o.constructor === Object --> false

o.constructor === 构造函数

## 42、简述一下JS继承，并举例

在 JS 中，继承通常指的便是原型链继承，也就是通过**指定原型**，并可以通过原型链继承原型上的**属性或者方法**。

最优化:圣杯模式

var inherit = (function(c,p){

var F = function(){};

return function(c,p){

F.prototype = p.prototype;

c.prototype = new F();

c.prototype = p.prototype;

c.prototype.constructor = c; }})();

使用 ES6 的语法糖 class / extends

## 43、函数柯里化

在函数式编程中，**函数是一等公民**。那么函数柯里化是怎样的呢？

**函数柯里化指的是将能够接收多个参数的函数转化为接收单一参数的函数**，并且返回接收**余下参数**且返回结果的**新函数**的技术。

函数柯里化的主要作用和特点就是**参数复用、提前返回和延迟执行**。

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的技术，称为函数的柯里化。**通常可用于在不侵入函数的前提下，为函数预置通用参数，供多次重复调用**。

const add = function add(x) {

return function (y) {

return x + y }

}

const add1 = add(1) add1(2) === 3 add1(20) === 21

## 44、说说bind、call、apply 区别？

call 和 apply 都是为了解决改变 **this 的指向**。作用都是相同的，只是**传参**的方式不同。

除了第一个参数外，**call 可以接收一个参数列表，apply 只接受一个参数数组**。

let a = { value: 1}

function getValue(name, age){

console.log(name)

console.log(age)

console.log(this.value)}

传参方式如下：

getValue.call(a, 'yck', '24')

getValue.apply(a, ['yck', '24'])

bind和其他两个方法作用也是一致的，只是该方法会**返回一个函数**。并且我们可以通过 **bind实现柯里化**。

（下面是对这三个方法的扩展介绍）

如何实现一个 bind 函数

对于实现以下几个函数，可以从几个方面思考

不传入第一个参数，那么默认为 window改变了 this 指向，让新的对象可以执行该函数。那么思路是否可以变成给新的对象添加一个函数，然后在执行完以后删除？

Function.prototype.myBind = function (context) {

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

throw new TypeError('Error') }

var \_this = this

var args = [...arguments].slice(1)

//arguments 是一个类[数组](https://so.csdn.net/so/search?q=%E6%95%B0%E7%BB%84&spm=1001.2101.3001.7020" \t "https://blog.csdn.net/weixin_43720095/article/details/_blank)对象。代表传给一个function的参数列表 遍历所有的参数列表 扩展操作符可以将 arguments 展开成独立的参数。

// 返回一个函数return function F() {

// 因为**返回了一个函数**，我们可以 new F()，所以需要判断

if (this instanceof F) {

return new \_this(...args, ...arguments) }

return \_this.apply(context, args.concat(...arguments)) }}如何实现一个call函数

Function.prototype.myCall = function (context) {

var context = context || window

// 给 context 添加一个属性

//getValue.call(a, 'yck', '24') => a.fn = getValue

context.fn = this

// 将 context 后面的参数取出来

var args = [...arguments].slice(1)

// getValue.call(a, 'yck', '24') => a.fn('yck', '24')

var result = context.fn(...args)

// 删除 fn delete context.fn return result}

如何实现一个apply函数

Function.prototype.myApply = function (context) {

var context = context || windowcontext.fn = this

var result

// 需要判断是否存储第二个参数

// 如果存在，就将第二个参数展开

if (arguments[1]) { result = context.fn(...arguments[1]) } else { result = context.fn() }

delete context.fn return result}

## 45、箭头函数的特点

Function a() {

return () => {

return () => {

console.log(this) } }}

console.log(a()()())

箭头函数其实是**没有 this**的，这个函数中的 **this只取决于他外面的第一个不是箭头函数的函数的 this**。在这个例子中，因为调用 a符合前面代码中的第一个情况，所以 **this是 window**。并且 this一旦绑定了上下文，就不会被任何代码改变。

## 箭头函数和普通函数的区别

1：写法不一样

普通函数

function greeting(){}

箭头函数

let greeting=()=>{}

2：普通函数存在变量提升的现象

普通函数

greeting(); //正确

function greeting(){}

箭头函数

greeting(); //错误

let greeting=()=>{}

3：箭头函数不能作为构造函数使用

普通函数

function greeting(name){

this.name=name

}

let greet=new greeting(“why”)

console.log(greet.name)//why

箭头函数

let greeting=(name)=>{

this.name=name

}

let object=new greeting(“why”)

console.log(greet.name)//undefined

4：两者this的指向不同

普通函数的this指向的是**谁调用该函数就指向谁**

箭头函数的this指向的是在你书写代码时候的上下文环境对象的this，如果没有上下文环境对象，那么就指向最外层对象window。

普通函数

let one=1;

function greeting(){

console(this)//Number{1}

}

greeting.call(one)

箭头函数

let greeting=()=>{

console(this)//{}

}

greeting.call(one)

5：箭头函数的arguments指向它的父级函数所在作用域的arguments

function greeting(){

console.log(“-”,arguments)

let innerFn=()=>{

console.log(“--”,arguments)

}

innerFn(“innerParms”)

}

greeting(“Hello World”)

- Arguments[“Hello World”]

-- Arguments[“Hello World”]

6：箭头函数没有new.target(返回构造函数的使用)

先说明下new.target是干嘛的，这家伙是用来检测函数是否被当做构造函数使用，他会返回一个指向构造函数的引用。

因为箭头函数不能当做构造函数使用，自然是没有new.target的。

function greeting(){

console.log(new.target)

}

let greeting=()=>{

console.log(new.target)

}

## js实现一个栈的类代码

class StackArray {

constructor() {

this.items = [];//用数组保存栈元素

}

//向栈添加元素

push(element) {

this.items.push(element);

}

//从栈移除元素

pop() {

return this.items.pop();

}

//查看栈顶元素

peek() {

return this.items[this.items.length - 1];

}

//检查栈是否为空

isEmpty() {

return this.items.length === 0;

}

//查看栈的长度

size() {

return this.items.length;

}

//清空栈元素

clear() {

this.items = [];

}

//栈转化为数组

toArray() {

return this.items;

}

//栈转化为字符串

toString() {

return this.items.toString();

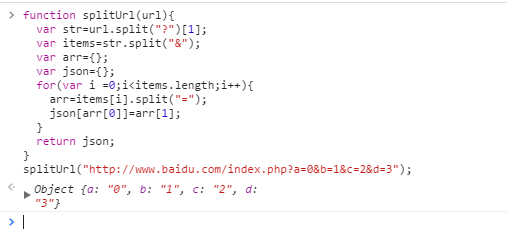
}

}

## 将url网址参数部分提取出来代码

url=”http://www.baidu.com/index.php?a=0&b=1&c=2&d=3”

function splitUrl(url){  
  var str=url.split("?")[1];//a=0&b=1&c=2&d=3  
  var items=str.split("&");//a=0,b=1,d=3  
  var arr={};  
  var json={};  
  for(var i =0;i<items.length;i++){  
    arr=items[i].split("=");//a,0 b,1 d,3  
    json[arr[0]]=arr[1];//a:”0”,b:”1”, d:”3”  
  }  
return json;  
}  
splitUrl("http://www.baidu.com/index.php?a=0&b=1&c=2&d=3");



# 程序阅读题

## 47、let和var

1、下面程序输出的结果是什么？

functionsayHi() {

console.log(name);

console.log(age);

var name = "Lydia";

let age = 21;}

sayHi();

A: Lydia 和 undefined

B: Lydia 和 ReferenceError

C: ReferenceError 和 21

D: undefined 和 ReferenceError

在函数中，我们首先使用var关键字声明了name变量。这意味着变量在**创建阶段**会被提升

（**JavaScript会在创建变量创建阶段为其分配内存空间**），默认值为undefined，直到我们实际执行到使用该变量的行。我们还没有为name变量赋值，所以它仍然保持undefined的值。

使用let关键字（和const）声明的变量也会存在**变量提升**，**但与var不同，初始化没有被提升**。**在我们声明（初始化）它们之前，它们是不可访问的。这被称为“暂时死区”**。当我们在声明变量之前尝试访问变量时，JavaScript会抛出一个ReferenceError。

关于let的是否存在变量提升，我们何以用下面的例子来验证：

let name = 'ConardLi'{

console.log(name)

// Uncaught ReferenceError: name is not definedlet

name = 'code秘密花园'}

let变量如果不存在变量提升，console.log(name)就会输出ConardLi，结果却抛出了ReferenceError，那么这很好的说明了，let也存在变量提升，但是它存在一个“暂时死区”，在变量未初始化或赋值前不允许访问。

变量的赋值可以分为三个阶段：

**创建变量**，在内存中开辟空间初始化变量，将变量初始化为undefined(没有初始化就访问),ReferenceError(没有初始化不可以访问)”暂时死区”

真正赋值关于let、var和function：

let的「创建」过程被提升了，但是初始化没有提升(没有初始化不可以访问)。

var的「创建」和「初始化」都被提升了。

function的「创建」「初始化」和「赋值」都被提升了。

## 48、变量提升原理

//普通函数变量提升，先调用后写函数不妨碍调用函数，先打印后赋值undefined

//箭头函数变量提升，先调用后写函数直接报错，先打印后赋值ReferenceError

var a = 10;

(function () {

console.log(a) //undefined

a = 5

console.log(window.a) //10

var a = 20;

console.log(a)})() //20

依次输出：undefined -> 10 -> 20

**变量提升原理**

在立即执行函数中，var a = 20; 语句定义了一个**局部变量** a，由于js的**变量声明提升机制**，局部变量a的声明会被提升至立即执行函数的**函数体最上方**，且由于这样的提升并不包括赋值，因此第一条打印语句会打印undefined，最后一条语句会打印20。由于变量声明提升，a = 5; 这条语句执行时，局部的变量a已经声明，因此它产生的效果是对局部的变量a赋值，此时window.a 依旧是最开始赋值的10。

## 49、静态方法

class Chameleon {

static colorChange(newColor) {

this.newColor = newColor; }

constructor({ newColor = "green" } = {}) {

this.newColor = newColor; }}

const freddie = new Chameleon({

newColor: "purple" });

freddie.colorChange("orange");}

A: orangeB: purpleC: greenD: TypeError答案: D

colorChange方法是静态的。**静态方法仅在创建它们的构造函数中存在，并且不能传递给任何子级**。由于freddie是一个子级对象，函数不会传递，所以在freddie实例上不存在freddie方法：抛出TypeError。

## 50、==和重写toString方法

var a = ?;if(a == 1 && a == 2 && a == 3){ conso.log(1);}

因为**==会进行隐式类型转换** 所以我们重写**toString方法**就可以了

答案如下

var a = { i: 1,

toString() {

return a

i++; }}

if( a == 1 && a == 2 && a == 3 ) {console.log(1);}

## 51、打印对象数组

var obj = {'2': 3,

'3': 4,

'length': 2,

'splice': Array.prototype.splice,

'push':Array.prototype.push}

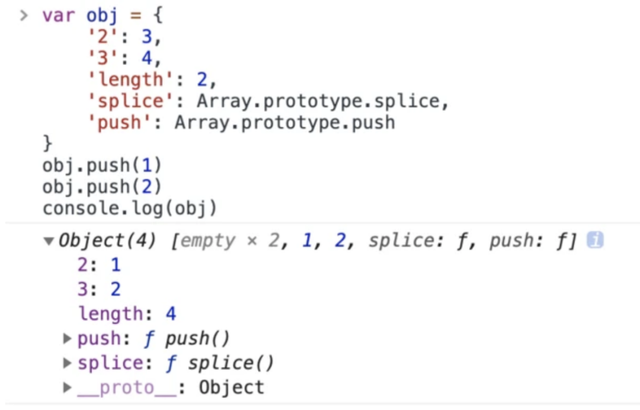
obj.push(1)

obj.push(2)

console.log(obj)

1. 使用第一次push，obj对象的push方法设置 obj[2]=1;obj.length+=1
2. 使用第二次push，obj对象的push方法设置 obj[3]=2;obj.length+=1
3. 使用console.log输出的时候，因为obj具有 length 属性和 splice 方法，故将其作为数组进行打印

4.打印时因为数组未设置下标为 0 1 处的值，故打印为empty，主动 obj[0] 获取为 undefined



## 52、连续赋值(.优先级大于=)

var a = {n: 1};

var b = a;

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

//优先执行a.x 第一步a={n:1,x:undefined} b={n:1,x:undefined}

//第二步从右向左执行a={n:2} a.x=a b={n:1,x:{n:2}}

//最终a={n:2} b={n:1,x:{n:2}}

console.log(a.x)

console.log(b.x)

答案undefined {n:2}

首先，a和b同时引用了{n:2}对象，接着执行到a.x = a = {n：2}语句，尽管赋值是从右到左的没错，但是**.的优先级比=要高**，所以这里首先执行a.x，相当于为a（或者b）所指向的{n:1}对象新增了一个属性x，即此时对象将变为{n:1;x:undefined}。之后按正常情况，从右到左进行赋值，此时执行a ={n:2}的时候，a的引用改变，指向了**新对象**{n：2},而b依然指向的是旧对象。之后执行a.x = {n：2}的时候，并不会重新解析一遍a，**而是沿用最初解析a.x时候的a**，也即旧对象，故此时旧对象的x的值为{n：2}，旧对象为 {n:1;x:{n：2}}，它被b引用着。后面输出a.x的时候，又要解析a了，此时的a是指向**新对象**的a，而这个新对象是没有x属性的，故访问时输出undefined；而访问b.x的时候，将输出旧对象的x的值，即{n:2}。

## 53、传参原始类型(值)和对象(内存地址)区别

function checkAge(data) {

if (data === { age: 18 }) {

console.log("You are an adult!"); }

else if (data == { age: 18 }) {

console.log("You are still an adult."); }

else {

console.log(`Hmm..You don't have an age I guess`); }}

checkAge({ age: 18 });

Hmm.. You don't have an age I guess

在比较相等性，**原始类型通过它们的值进行比较，而对象通过它们的引用进行比较**。JavaScript检查对象是否具有对内存中相同位置的引用。

我们作为**参数传递的对象**和我们用于**检查相等性的对象在内存中位于不同位置**，所以它们的引用是不同的。

这就是为什么{ age: 18 } === { age: 18 }和

{ age: 18 } == { age: 18 }返回 false的原因。

## 54、对象和数组的区别

const obj = { 1: "a", 2: "b", 3: "c" };

const set = new Set([1, 2, 3, 4, 5]);

obj.hasOwnProperty("1");

obj.hasOwnProperty(1);

set.has("1");

set.has(1);

答案true` `true` `false` `true

所有**对象键**（不包括Symbols）都会被存储为**字符串**，即使你没有给定字符串类型的**键**。这就是为什么obj.hasOwnProperty（'1'）也返回true。

上面的说法不适用于Set。

在我们的Set中没有“1”：set.has（'1'）返回false。

它有数字类型1，set.has（1）返回true。

## 55、构建方法和产生实例的构建方法

function Foo() {

Foo.a = function() {

console.log(1) }

this.a = function() {

console.log(2) }}

// 以上只是 Foo 的构建方法，没有产生**实例**，此刻也没有执行Foo.prototype.a = function() {

console.log(3)}

// 现在在 Foo 上挂载了原型方法 a ，方法输出值为 3

Foo.a = function() {

console.log(4)}// 现在在 Foo 上挂载了直接方法 a ，输出值为 4

Foo.a();// 立刻执行了 Foo 上的 a 方法，也就是刚刚定义的，所以//输出 4

let obj = new Foo();/\* 这里调用了 Foo 的构建方法。Foo 的构建方法主要做了两件事：1. 将全局的 Foo 上的直接方法 a 替换为一个输出 1 的方法。2. 在**新对象**上挂载直接方法 a ，输出值为 2。obj.a();// 因为有直接方法 a ，不需要去访问原型链，所以使用的是构建方法里所定义的 this.a，// 输出 2

Foo.a();// 构建方法里已经替换了**全局 Foo** 上的 a 方法，所以// 输出 1

# Vue

## 56、v-if 和 v-show 的区别

v-if false**不渲染**，适合更新**不频繁**的，**节省性能**。

v-show false渲染

v-show通过CSS display控制**显示和隐藏**

v-if组件真正的渲染和**销毁**，而不是显示和隐藏，频繁切换显示状态就用v-show,否则用v-if

v-if当条件为false时，压根不会有对应的元素在DOM中。

v-show当条件为false时，仅仅是将元素的display属性设置为none而已。

当需要在显示与隐藏之间切片很频繁时，使用v-show。

当只有一次切换时，通过使用v-if。

## 57、v-for 的 key

因为 VDOM 的 diff 算法，通过判断**新**vnode和**旧**vnode的key是否相等，从而**复用**与新节点对应的老节点，**减少渲染次数**，提升渲染性能,**节约性能**开销

必须用key,且不能是index和random

## 58、父子通信

### props父组件通过props给子组件传值。

父组件通过props(properties)属性向子组件传递数据。

在子组件中，使用选项props来声明需要从父级接受到的数据。

props的值有两种方式：

方式一：字符串数组，数组中的字符串就是传递时的名称。

方式二：对象，对象可以设置传递时的类型，也可以设置默认值等。

<cpn :cmovies=”movies” :cmessage=”message”></cpn>

props:[“cmovies”,”cmessage”]

父组件注册父组件，子组件构造器props用来存储父组件属性，构造器对象模板，使用子组件构造器（将父组件属性赋值给子组件属性）。

props数据验证，支持的数据类型：

方式二：类型限制

String Number Boolean Array Object Date Function Symbol

提供一些默认值

type default required

类型是对象或者数组时，默认值必须是一个函数。

props驼峰标识 v-bind不支持驼峰 props中的驼峰标识cInfo改成 div组件中使用:c-info。

template模板中用div包裹。

app父组件，cpn子组件，div父组件模板，template子组件模板。

### emit子组件通过$emit向父组件发送事件。

子组件通过$emit发送事件向父组件发送消息。

btnClick(item){

this.$emit(“item-click”,item) item-click子传父定义事件名称，item子组件参数

}

在<cpn @item-click=”cpnClick”></cpn> 默认传递event item

在父组件定义事件

cpnClick(item){

console.log(“”,item)

}

自定义事件：

当子组件需要向父组件传递数据时，就要用到自定义事件了。

v-on不仅仅可以用于监听DOM事件，也可以用于组件间的自定义事件。

流程：

在子组件中，通过$emit()来触发事件。

在父组件中，通过v-on来监听子组件事件。

## 59、描述Vue组件的生命周期

**挂载阶段**

**beforeCreate**:此阶段为**实例初始化**之前,此时**数据观察**和**事件机制**还没有形成,不能获取到dom节点；

**created**：此阶段的**vue实例(组件)**已经创建,仍不能获取DOM 节点.把vue 的一个实例给初始化了，只是存在于 js 内存的一个**变量**而已，这个时候并没有开始**渲染，一般用来获取服务器数据**；

**渲染阶段**

**beforeMount**:在这一阶段,我们虽然还不能获取到具体 DOM 元素,但 vue 挂载的**根节点**已经创建,下面 vue 对DOM 的操作将围绕这个根元素继续进行,beforeMount 这个阶段是过渡性的,一般一个项目只能用到一两次；

**mounted**：组件真正绘制完成了，页面已经渲染完了,**数据和DOM** 都已被**渲染**出来,一般我们的**异步请求**都写在这里)

**更新阶段**

**beforeUpdate**: 这一阶段，vue遵循**数据驱动DOM** 的原则，beforeUpdate 函数在数据更新后没有立即更新**服务器数据**，但是**DOM 数据**会改变，这是双向数据绑定的作用，服务器数据和浏览器数据没有同步；

updated：这一阶段，**DOM 会和更改过的内容同步**)，**一般用来获取组件到顶部的位置**

**销毁阶段**

**beforeDestroy**：在上一阶段vue已经成功通过**数据驱动DOM 的修改**，当我们不再需要 **vue 操纵 DOM** 时，就要销毁 vue，也就是**清除vue 实例与 DOM 的关联**，调用destroy方法可以销毁当前组件。**在销毁前**，会触发 beforeDestroy 钩子函数；

**destroyed**：在销毁后，会触发destroyed 钩子函数）

## 60、描述组件渲染和更新过程

渲染组件时，会通过`Vue.extend`方法构建子组件的构造函数(创建组件构造器)，并进行实例化(组件对象)。最终手动调用**`$mount()`进行挂载**(将**组件对象挂载到某个元素**上)。

更新组件时会进行**patchVnode`流程.核心就是diff算法**

因为 VDOM 的 diff 算法，通过判断**新**vnode和**旧**vnode的key是否相等，从而**复用**与新节点对应的老节点，**减少渲染次数**，提升渲染性能,**节约性能**开销

必须用key,且不能是index和random

import Toast from '@/components/common/toast/Toast'

const obj={}

obj.install=function(Vue){

  //console.log("---",Vue);

  //Vue.extend(Toast)//组件构造器

  //document.body.appendChild(Toast.$el)//加载模板内容

  //1.创建组件构造器

  const toastContrustor=Vue.extend(Toast)

  //2.new的方式，根据组件构造器，可以创建出来一个组件对象

  const toast=**new** toastContrustor()

  //3.将组件对象，手动挂载到某个元素上

  toast.$mount(document.createElement("div"))

  //4.toast.$el对应的就是div

  document.body.appendChild(toast.$el)

  Vue.prototype.$toast=toast;//对象加载到原型上

}

export default obj

## 61、Vue实现数据双向绑定 v-model的原理

v-model本质上是 **:value和v-on**的结合体，就是绑定他的value,通过v-on触发，从而**更新数据**

<input type=”text” v-bind:value=”dnumber1” @input=”dnumber1=$event.target.value”>

v-model双向绑定修饰符

lazy，number，trim，

双向绑定得的实现主要依赖于**Object.defineProperty()**,通过这个函数可以监听到**get,set**事件

其中**observer**是最主要的部分，用Object.defineProperty来实现数据的劫持，然后用他的set,get方法来通知**订阅者**，触发update方法，从而实现**更新视图**

订阅者模式：每一个{{name}} **v-model=‘name’都会添加一个订阅者**，从而**监听不同部分的变化**，每一部分变化时都会循环触发相应的订阅者，**更新到页面**中。

## 62、VUE组件中的data必须是函数

Object是引用数据类型，如果不用function返回，每个组件的data都是**内存的同一个地址**，**一个数据改变了其他也改变了**；

JavaScript只有**函数构成作用域**(注意理解作用域，**只有函数{}构成作用域,对象的{}以及if(){}都不构成作用域**),data是一个函数时，**每个组件实例都有自己的作用域**，**每个实例相互独立，不会相互影响**。

组件是一个单独功能模块的封装，这个模块属于自己的HTML模块，也应该有属于自己的数据data。

如果所有的数据都放到在Vue实例中，Vue实例就会变得非常臃肿。

Vue组件应该有自己保存数据的地方。

data(){

return{

}

}

而且这个函数返回一个对象，对象内部保存着数据。

单独的作用域，一个组件的数据变化，对其他组件数据不会影响，函数的对象不是不同的，地址不同，函数的作用域。

**63、computed有何特点**

缓存，data不变不会重新计算，提高性能，一般用来返回获取服务器的数据，不会重复返回。用来解决循环遍历的计算。

而methods中data改变一次就会重新计算一次，一般用来触发事件

## 64、ajax请求应该放在哪个生命周期

mounted 事件总线+防抖动 对服务器请求来的数据不会一个一个频繁调用，只会调用一次，提高了数据加载的性能

js是单线程，**ajax异步获取数据**

放在mounted之前没有用，只会让逻辑更混乱

## 65、如何将组件所有props传递给子组件

$props

<User v-bind='$props'>

## 66、混合多个组件有相同的逻辑，如何抽离

mixin

所以引入混入这个概念

创建common/mixin.js文件

继承是减少类的重复代码

混入是减少对象的重复代码

插件之所以可以直接用就是被定义在原型上面的

混合是针对于共用一个对象

插件是针对于共用一个组件

1、混合

混合 (mixins) 是一种分发 Vue 组件中可复用功能的非常灵活的方式。混合对象可以包含任意组件选项。以组件使用混合对象时，所有混合对象的选项将被混入该组件本身的选项

当组件和混合**对象含有同名选项**时，这些选项将以恰当的方式混合

其调用的**数据和方法**都是重复的 组件还是要挂载到**dom上**

2、插件

Vue.js 的插件应当有一个公开方法 install 。这个方法的第一个参数是 Vue 构造器 , 第二个参数是一个可选的选项对象

将组件作为一个**公共对象**来使用，方便调用它的**方法和属性** 组件挂载到**原型**上

## 67、何时使用异步组件

加载大组件，路由异步加载

## 68、何时需要使用keep-alive

缓存组件，不需要重复渲染，如多个静态tab页的切换，优化性能

## 69、何时需要使用beforeDestory

**解绑自定义事件**event.$off，**清除定时器**，解绑自定义的DOM事件，如window.scroll等，vue不需要操作DOM时候，也就是说vue实例与DOM没有关联，就要摧毁vue。

## 70、插槽

有name的属于具名插槽，没有name属于匿名插槽

普通插槽**渲染**的位置是在它的**父组件**里面，而不是在子组件里面。

作用域插槽渲染是在**子组件**里面

抽取共性，保留不同。

**1.插槽slot**

**在渲染父组件的时候，会将插槽中的先渲染。**

创建组件虚拟节点时，会将组件的儿子的虚拟节点保存起来。当初始化组件时,通过插槽属性将儿子进行分类 {a:[vnode],b[vnode]}

**渲染组件时会拿对应的slot属性的节点进行替换操作**。（插槽的作用域为父组件，插槽中HTML模板显示不显示、以及怎样显示由**父组件**来决定）

有name的父组件通过html模板上的slot属性关联具名插槽。没有slot属性的html模板默认关联匿名插槽。

**2.作用域插槽slot-scope**

作用域插槽在解析的时候，不会作为组件的孩子节点。会解析成**函数**，当子组件渲染时，会调用此**函数进行渲染**。

或者可以说成作用域插槽是**子组件可以在slot标签上绑定属性值**，**在父组件可以拿到子组件的数据，通过子组件绑定数据传递给父组件**。（插槽的作用域为子组件）

### 基本使用

组件的插槽：

组件的插槽也是为了让我们封装的组件更加具有扩展性，可以预留一定的空间，使用同一个组件，可以展示不同的内容，用插槽进行修饰（添加不同内容）。

让使用者可以决定组件内部的一些内容到底是什么。

抽取共性，保留不同。

最好的封装方式就是将共性抽取到组件中，将不同暴露为插槽。

一旦我们预留了插槽，就可以让使用者根据自己的需求，决定插槽中插入什么内容。

是搜索框，还是文字，还是菜单。由调用者自己来决定。

父：<cpn></cpn>组件里面就是插槽的内容(内容替换到插槽里面)，组件模板里面子：<slot></slot>插槽里面是插槽默认的内容，<cpn></cpn>组件里面有内容就替换，没有就是默认内容。

### 具名插槽

组件模板里面有多个插槽，替换第几个插槽默认的内容，给插槽起个名字。

<slot name=”center”></slot>

<cpn></cpn>组件里面<span slot=”center”></span>

### 编译作用域

<cpn v-bind:>一般去 Vue实例里面找，不会去组件里面找。

而在模板中一般去组件里面找。组件的作用域。

父组件模板的所有东西都会在父级作用域内编译；子组件模板的所有东西都会在子级作用域内编译。

### 作用域插槽

父组件替换插槽的标签，但是内容由子组件来提供。可以展示不同的样式。

通过父组件模板插槽获取子组件模板的子组件属性。

<cpn>

<template slot-scope=”slot”> //slot引用了子组件模板

<span v-for=”item in slot.data”>{{item}-}</span>

//<span>slot.data.join(“-”)</span>

</template>

</cpn>

<slot :data=”planguage”> //子组件属性planguage

</slot>

## 71、vuex中action和mutation有何区别

action中处理异步，mutation不可以

mutation做**原子操作**

action可以整合多个mutation

## 72、Vue-router常用的路由模式

Hash: 使用URL的hash值来作为路由。支持所有浏览器。

History: HTML5 History API 和服务器配置。

## 73、监听data的核心API

object.defineProperty 响应式原理

以及深度监听、监听数组

## 74、Vue如何监听数组变化

object.defineProperty不能监听数组变化

**重新定义原型，重写pop push**等方法，实现监听

proxy可以原生支持监听数组变化

## 75、请描述响应式原理

-监听data变化

组件渲染和更新的流程

## 76、diff算法的事件复杂度

o（n)

## 77、简述diff算法过程

在执行Diff算法的过程就是调用名为 **patch 的函数，比较新旧节点**。一边比较一边给真实的 DOM 打补丁。patch 函数接收两个参数 oldVnode 和 Vnode，它们分别代表新的节点和之前的旧节点。这个patch函数会比较 oldVnode 和 vnode 是否是相同的, 即函数 sameVnode(oldVnode, vnode), 根据这个函数的返回结果分如下两种情况：

true：则执行 patchVnode

false：则用 vnode 替换 oldVnode

## 78、vue为何是异步渲染，nextTick何用

**合并data修改**，以提高渲染性能

$nextTick在**DOM更新完**之后，触发回调

## 79、vue常见性能优化

合理使用v-show和v-if

合理使用computed

v-for时加key,以及避免和v-if同时使用

自定义事件、DOM事件及时销毁（内存泄漏）

合理使用异步组件

合理使用keep-alive

data层级不要太深

使用vue-loader在开发环境做模板编译（预编译）

## 80、两种模式的区别，以及两种模式是怎么实现的

**1、hash模式**

hash模式背后的原理是`onhashchange`事件，可以在`window`对象上监听这个事件：

hash发生变化的url都会被浏览器记录下来，从而你会发现浏览器的前进后退都可以用了，同时点击后退时，页面字体颜色也会发生变化。这样一来，尽管浏览器没有请求服务器，但是页面状态和url一一关联起来，后来人们给它起了一个霸气的名字叫前端路由，成为了单页应用标配。

**2、history模式**

随着history **api**的到来，前端路由开始进化了，前面的hashchange，你只能改变**#**后面的url片段，而history api则给了前端完全的自由

history api可以分为两大部分：切换和修改

（1）**切换**历史状态

　　包括**`back、forward`、`go`**三个方法，对应浏览器的前进，后退，跳转操作，有同学说了，(谷歌)浏览器只有前进和后退，没有跳转，嗯，在前进后退上长按鼠标，会出来所有当前窗口的历史记录，从而可以跳转(也许叫跳更合适)：

（2）**修改**历史状态

　　包括了`**pushState、replaceState`**两个方法，这两个方法接收三个参数：stateObj，title，url

　　通过pushstate把页面的状态保存在state对象中，当页面的url再变回这个url时，**可以通event.state取到这个state对象**，从而可以对页面状态进行还原，这里的页面状态就是页面字体颜色，其实滚动条的位置，阅读进度，组件的开关的这些页面状态都可以存储到**state**的里面。

　　通过history api，我们丢掉了丑陋的#，但是它也有个毛病：

　　不怕前进，不怕后退，就怕刷新，f5，（如果后端没有准备的话），因为**刷新是实实在在地去请求服务器的**。

　　在hash模式下，前端路由修改的是**#**中的信息，而浏览器请求时是不带它玩的，所以没有问题。但是在history下，你可以自由的修改**path**，当刷新时，如果**服务器中没有相应的响应或者资源**，会分分钟刷出一个404来。

（3）popstate实现history路由拦截，**监听页面返回事件**

　　当活动**历史记录条目**更改时，将触发**popstate事件**。

　　1、如果被激活的历史记录条目是通过对history.pushState() 的调用创建的，或者受到对 history.replaceState()的调用的影响，popstate事件的state属性包含历史条目的状态对象的副本。

　 2、需要注意的是调用history.pushState()或 history.replaceState()用来在**浏览历史中添加或修改记录**，不会触发popstate事件；

　　只有在做出浏览器动作时，才会触发该事件，如用户点击浏览器的回退按钮（或者在Javascript代码中调用history.back()）

## 81、Vue3 比 Vue2 有什么优势

**性能**比Vue2.x快1.2~2倍

原因1：**diff方法优化**：

vue2中的**虚拟dom是全量的对比**（每个节点不论写死的还是动态的都会比较）

vue3新增了**静态标记（patchflag）**与上次虚拟节点对比时，只对比带有**patch flag**的节点（动态数据 所在的节点）；可通过**flag信息得知当前节点要对比的具体内容**

原因2：**静态提升**

vue2无论元素是否参与更新，每次都会重新创建然后再渲染

vue3对于**不参与更新的元素，会做静态提升**，只会**被创建一次**，在渲染时直接复用即可

原因3：**时间侦听器缓存**

vue2默认情况下onClick会被视为动态绑定，所以每次都会追踪它的变化

vue3但是因为是同一个函数，所以不用追踪变化，直接缓存起来复用即可

原因4：**ssr渲染**

**按需编译**，体积比vue2.x更小

**组合API**（类似react hooks）

更好的Ts支持

暴露了自定义渲染API

更先进的组件

**Fragment**:模板可以有多个根元素

## 深拷贝和浅拷贝区别

Java基本数据类型有**四类八种**，分别为：整型：byte,short,int,long,浮点型：float,double,逻辑型：boolean,字符型：char

**在栈中可以直接分配内存的数据是基本数据类型**。**引用数据类型是数据的引用在栈中，但是他的对象在堆中**。要想学好Java必须知道各种数据的在内存中存储位置。对内存有了很好的了解，可以有助分析程序。

内容：

浅拷贝：如果是基本数据类型，**名字和值都会储存在栈内存**中，

如果是引用数据类型，**名字存在栈内存中，值存在堆内存中**，但是栈内存会提供**一个引用的地址指向堆内存中的值**

深拷贝：**往往需要被赋值的对象生成一个新的对象，开辟新的空间，地址发生改变**

方法：

浅拷贝：**for···in只循环第一层，Object.assign方法**，直接用**=**赋值

深拷贝：采用**递归去拷贝所有层级属性**，通过**JSON对象**来实现深拷贝，通过**jQuery的extend方法**实现深拷贝，**lodash函数库**实现深拷贝，用**slice实现对数组**的深拷贝，用**concat**实现对数组的深拷贝

## 83、ES6---箭头函数()=>{} 与function的区别

this的指向：

使用function定义的函数，this的指向随着调用环境的变化而变化的，而箭头函数中的this指向是固定不变的，一直指向的是定义函数的环境。

构造函数：

function是可以定义构造函数的，而箭头函数是不行的。

变量提升：

由于js的内存机制，function的级别最高，而用箭头函数定义函数的时候，需要var(let const定义的时候更不必说)关键词，而**var所定义的变量不能得到变量提升**，故**箭头函数一定要定义于调用之前**！

## computed和methods的区别

computed:解决任何**复杂重复逻辑**,computed**响应式依赖缓存,**重复次数只有一次，**函数不会被频繁的调用**，代码简洁，性能更高

methods:没有缓存,重复次数多，性能比较低,

## mustache语法

v-html

v-pre

v-once

v-clock

## MVVM理解

MVVM分为Model、View、ViewModel三者。

Model：代表数据模型，数据和业务逻辑都在Model层中定义；

View：代表UI视图，负责数据的展示；

ViewModel：就是与界面(view)对应的Model。因为，数据库结构往往是不能直接跟界面控件一一对应上的，所以，需要再定义一个数据对象专门对应view上的控件。而**ViewModel的职责就是把model对象封装成可以显示和接受输入的界面数据对象**。

前段时间面试蚂蚁金服，面试官问我，比如UI中有一个li列表，它是怎么与我们的数据对应的，当时没明白他在问什么，现在想想应该是考察ViewModel。

Model和View并无直接关联，而是通过ViewModel来进行联系的，Model和ViewModel之间有着双向数据绑定的联系。因此当**Model中的数据改变时会触发View层的刷新**，View中由于用户交互操作而改变的数据也会在Model中同步。

简单的说，ViewModel就是View与Model的连接器，View与Model通过ViewModel实现双向绑定。

## 86、var和let区别以及const

let/var

我们可以将let看成更完美的var，

var可以看作是全局作用域，没有块级作用域引起的问题，if的块级和for的块级的问题，赋值可以被改变，造成了全局污染。解决方案：

闭包处理(function(i){})(i)函数是有作用域的

或者把var i改成let i

而let是有**块级作用域**的，不会与外面**变量名产生冲突**，**局部**变量

const

将某个变量修饰为常量，不可以再次赋值，来保证数据的安全性。建议：在ES6研发中，优先使用const，只有需要改变某个标识符的时候才使用let。

注意一：一旦给const修饰的标识符被**赋值之后，不能修改**。注意二：**必须进行赋值**。注意三：常量的含义是指向的**对象不能修改**（对象内存地址没变，指向属性），但是可以改变**对象内部的属性**（例子app不能改变，可以改变app里面data中的属性）

## 87、Key

Vue在进行DOM渲染时，出于性能考虑，会尽可能的**复用已经存在的元素**，而**不是创建新的元素**。

所以我们需要使用key来给每一个节点做一个唯一标识。

Diff算法就可以正确的识别此节点。

找到正确的位置区插入新的节点。

key的作用主要是为了高效的更新虚拟DOM。

## 88、数组中响应式的方法

pop删除最后一个元素

shift删除第一个元素

unshift在最前面添加一个或者多个元素

push在最后面添加一个或者多个元素

splice(从第几个元素删除，删除几个元素<如果没有传就删除后面所有元素>，替换前面删除元素)

sort排序

reverse反转

function sum(...num){  
}

sum(可以传递任意参数)

注意：通过索引值修改数组中的元素不是响应式的，解决：1.通过splice方法解决，2.Vue.set(修改的对象，索引值，修改后的值)

## 89、遍历数组对象取值

1.for(let i=0;i<this.books.length;i++){}

2.for(let i in this.books){}中i拿到的也是索引值index。

3.for(let item of this.books){}拿到的是一个对象item,item=this.books[i]

调用函数+返回值

## 父子访问

### 父访问子-children-refs

父组件访问子组件：使用$children或$refs(refrence引用)。$children的访问：this.$children[0].调用方法是一个数组类型，它包含所有子组件对象,下标拿子组件对象不好，不知道子组件有多少个，一般拿所有子组件。通过一个遍历，取出所有子组件的message状态。组件一般使用双标签。

一般用$refs，对象类型，默认是一个空的对象。

<cpn ref=”aaa”></cpn>

调用this.$refs.aaa.name。

可以调用子组件的方法

### 子访问父-parent-root

子组件访问父组件：使用$parent，用的比较少，访问的是VueComponent。

this.$root,访问的是Vue实例，用的比较少。

可以调用父组件的方法

## 模块化

解决**全局变量同名**（冲突）的问题。

## 组件化

如果我们将一个页面中所有的处理**逻辑**全部放在一起，处理起来就会变得非常复杂，而且不利于后续的管理以及扩展。

但如果，我们将一个页面拆分成一个个小的**功能块**，**每个功能块完成属于自己这部分独立的功能**，那么之后整个页面的管理和维护就变得非常容易了。

它提供了一种抽象，让我们可以开发出一个个独立可复用的小组件来构造我们的应用。

任何的应用都会被抽象成一颗组件树。

# Webpack

## 93、模块化概念

从本质上来讲，webpack是一个现代的JavaScript应用的**静态模块**打包工具。

webpack其中一个核心就是让我们可能进行模块化开发，并且会帮助我们处理**模块间的依赖关系。(安装各种依赖loader进行配置)**

而且不仅仅是JavaScript文件，我们的**CSS、图片、json**文件等等在Webpack中都可以被当作模块来使用。

## 94、打包概念

就是将webpack中的**各种资源模块**进行打包并合成一个或多个包（Bundle）。

并且在打包的过程中，还可以对资源进行处理，比如**压缩图片，将scss转化成css，将ES6语法转成ES5语法，将TypeScript转化成JavaScript等等操作**。

Webpack模块化打包，为了可以正常的运行，必须依赖node环境，node环境为了可以正常执行很多代码，必须其中包含各种依赖的包，**npm工具**（node package manager）包管理工具。

## 插件和依赖的区别

plugin是**插件**的意思，通常是用于对某个现有的架构进行**扩展。**

webpack中的插件，就是对webpack现有功能的各种扩展，比如打包优化，文件压缩等等。

loader和plugin的区别

loader主要用于**转换某些类型**的模块，它是一个**转换器**。

plugin是插件，它是对webpack本身的扩展，是一个**扩展器**。

plugin的使用过程：

步骤一：通过npm安装需要使用的plugins（某些webpack已经内置的插件不需要要安装）

步骤二：在webpack.config.js中的plugins中配置插件

我们先来使用一个最简单的插件，为打包的文件添加版权声明

该插件名字叫BannerPlugin，属于webpack自带的插件

按照下面的方式来修改webpack.config.js的文件：

const webpack=require(“webpack”)会在node\_modules查找webpack

plugins与modules并列

plugins:{

new webpack.BannerPlugin(“最终版权归coderwhy所有”)

}

重新打包程序：查看bundle.js文件的头部，看到如下信息

/\*最终版权归coderwhy所有\*/

# CLI

## 98、描述

代码目录结构、项目结构和部署、热加载、代码单元测试

CLI是Command-Line Interface,翻译为命令行界面，俗称脚手架。

## 99、runtime-compiler和runtime-only的区别

runtime-compiler中template:”<App/>” components:{App}

template模板传给vue，vue会进行保存，vue保存在vue实例下的options。

进行pass（解析），解析成ast(抽象语法树 abstract syntax tree)，compiler编译成render函数（function）最终翻译成virtual虚拟 dom，进行渲染成真实的dom。

template->ast->render->virtual dom->UI真实dom

runtime-only中render:h=>h(App) render渲染 箭头函数 h(createElement)参数

1.createElement(“标签”,

{标签的属性},

[“Hello World”,createElement(“button”,[“按钮”])  
])

2.传入组件（对象）

箭头函数转换 render:function(h){return h(App)}

将template编译成了render函数

那么.vue文件中的template是由谁处理的了？

是由vue-template-compiler处理的 转成render函数 runtime-only

1.性能更多 2.下面的代码量更少(少6kb就是这么来的)

render->virtual dom->UI 真实dom

## 100、定义函数的方式

箭头函数：也是一种定义函数的方式

1.定义函数的方式：function

const aaa=function(){}

1. 对象字面量中定义函数

const obj={

aaa:function(){},

aaa(){}

}

1. ES6中的箭头函数

const aaa=(参数列表)=>{} 一个参数不需要括号，两个需要

函数代码块中只有一行代码

const mul=(num1,num2)=>num1\*num2 有返回值的

const mul=()=>console.log(“Hello World”)没有返回值,mul()返回值为undefined

结果作为箭头函数的返回值

什么时候使用箭头函数

函数里面使用其他函数时候

结论：箭头函数中的this是如何查找的了？

答案：向外层作用域中，一层层查找this，直到有this的定义。引用的就是最近作用域中的this。

# vue-router

## 描述路由映射关系

路由表本质上就是一个映射表决定了数据包的指向

映射表：{内网ip1:电脑mac1地址 内网ip2:电脑mac2 地址:电脑}

路由组件和组件路径地址形成一一对应的关系

路由器提供了两种机制：路由和传送

路由是决定数据包从**来源到目的地的路径**

传送将**输入端的数据**转移到合适的**输出端**

## replaceState与pushState区别

replaceState与pushState区别，replaceState不能进行前进和后退，不是连续的，不可以点击，替换，不是压栈和出栈，不会处于栈顶中

## 插件重定向

插件，可以通过Vue.use()来安装路由功能

path:””,

//redirect 重定向 默认打开会重定向到localhost:8080/#/home这个路径

redirect:”/home”

<router-view></router-view>路由组件展示

## <router-link>

<router-link>还有一些其他属性

tag:tag可以指定<router-link>之后渲染成什么组件，上面代码渲染成了<button>

replace:replace不会留下history记录，所以指定replace的情况下，后退键返回不能返回到上一个页面中。

active-class:当<router-link>对应的路由匹配成功时，会自动给当前元素设置一个router-link-active的class，设置active-class可以修改默认的名称。

<router-link>：该标签是一个vue-router中已经内置的组件，它会被渲染成一个<a>标签。

<router-view>：该标签会根据当前的路径，动态渲染出不同的组件。

## router和route

this.$router.push(“/home”) //to=”/home”

//this.$router.replace(“/home”)

history.pushState

history.replaceState

location.hash

location.href

{{$route.params.userId)}}

{{$route.query.name}}

router和组件里面this.$router是同一个。

push,replace,go,back,forword（pushState）方法

而route和组件里面this.$route是同一个

哪一个路由组件活跃就赋值给route

$route和$router是有区别的

$router为VueRouter实例，想要导航到不同URL，则使用$router.push方法

$route为当前router跳转对象里面可以获取name、path、query、params等

## 打包js

打包中的js文件我分为了三类

app~.js当前应用程序开发的所有代码（业务代码）

manifest~.js为了打包的代码做底层支撑的 (运行时)

([main.js,mathUtils.js,info.js]) \_\_webpack\_require\_\_(moduleId)

vendor~.js提供商第三方vue vue-router/axios/bs

## query和params

懒加载：用到时，再加载。

component:()=>import (“../components/Home”) //等价于Home变量

传递参数主要有两种类型:params和query

params的类型：

配置路由格式：/router/:id

传递方式：在path后面跟上对应的值

传递后形成的路径：/router/123,/router/abc

query类型：

配置路由格式：/router,也就是普通配置

传递的方式：对象中使用query的key作为传递方式

传递后形成的路径：/router?id=123,/router?id=abc

<router-link :to=”{path:”/profile”,query:{name:”why”,age=18,height:1.88}}”>

</router-link>

//没有v-bind to后面只会当成普通字符串{}解析，有v-bind就会真正当成语法对象{}去解析

## 八个生命周期

created一旦组件被创建出来，就会回调这个生命周期函数

mounted组件创建之后，当我们将template挂载到整个dom上之后，就会回调这个生命周期函数

updated界面{{message}}发生更新的时候，只要发生一次更新，就会回调这个生命周期函数。

destoryed销毁，就会回调这个生命函数

new Vue() ->init Events&Lifecycle(beforeCreate)->init injections&reactivity

创建Vue实例后就会**初始化事件和生命周期**，回调beforeCreate这个函数，**初始化响应式**（注射和反应），回调created这个函数。问是否有el这个属性，没有el这个属性就会vm.$mount(el)进行回调。如果有el这个属性，再问有没有template这个属性，如果有template这个属性就会将el这个属性替换掉，没有这个属性的话，说明很多东西都会编译成了render函数了，如果有template这个属性，**就会编译template属性变成render函数**。beforeMount函数之后，**就会真正把el属性替换掉**（相当于template上的东西被挂载到dom上了），这个时候就会回调mounted（挂载）这个函数，就会进行一种响应式的检测。**数据变化**时候就会回调beforeUpdate这个函数，就会通过**Virtual DOM重新渲染（re-render）并和真实DOM进行对比修改（patch补丁）**，调用updated这个函数进行**循环**。不再停留这个界面了（去下个界面），就会调用beforeDestory这个函数，**回收**很多东西，最后调用destoryed函数，**完全销毁**。

## 导航守卫

**前置守卫**

router.beforeEach(function(to,from,next){})

to：即将要进入目标的路由对象

from：当前导航即将要离开的路由对象

next：调用该方法后，才能进入下一个钩子

如果是后置钩子，也就是afterEach，不需要主动调用next()函数

**路由独享的守卫**

在routes:{

beforeEnter:(to,from,next)=>{next()}

}

**组件内的守卫**

const foo={

template:””,

beforeRouteEnter(to,from,next){},

beforeRouteUpdate(to,from,next){},

beforeRouteLeave(to,from,next){}

}

登录失败next(“./login”)

## vue-router-keep-alive

keep-alive也是Vue内置的一个组件，可以使被包含的组件保留状态，或避免重新渲染。//保证活着

属性：

include 字符串或正则表达式，只有匹配的组件会被缓存

exclude 字符串或正则表达式，任何匹配的组件都不会被缓存。

router-view也是一个组件，如果直接被包在keep-alive里面，所有路径匹配到的视图组件都会被缓存：

在App.vue文件中

<keep-alive exclude=”Profile,User”>

在Profile.vue文件中name:Profile(用的比较少) 正则不要随便加空格

<router-view></router-view> //只会创建一次（调用一次created函数）

</keep-alive>

在Home.vue文件中

activated(){

this.$router.push(this.path) //使用keep-alive才能引用

},

//deactivated(){

//this.path=this.$route.path

//}

将redirect路由映射清除

在data()里面设置path:”/home/news”属性

用组件内导航守卫

beforeRouterLeave(to,from,next){

this.path=this.$route.path;

next()

}

首页中使用path属性记录离开时的路径，在beforeRouteLeave中记录

# Vuex

Vuex是一个专为Vue.js应用程序开发的状态管理模式

多个组件共享的变量全部存储在一个对象里面

然后，将这个对象放在顶层的Vue实例中，让其他组件可以使用

响应式：数据发生改变，界面跟着改变

比如用户的登录状态、用户名称、头像、地理位置信息等等

比如商品的收藏、购物车中的物品等等

const store=new Vuex.store({

state:{

conuter:1000

},

mutations:{}, //commit state,参数 同步

actions:{}, //dispatch context(store), 异步 context.commit

getters:{}, //state,getters

modules:{}

})

actions:{

aUpdateInfo(context,payload){

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

setTimeout(()=>{

context.commit(“updateInfo”)

console.log(payload)

resolve(“111”)

},1000)

})

}

}

共享的状态抽取出来，交给我们的大管家，统一进行管理。

Vuex允许我们将store分割成模块(Module)，而每个模块拥有自己的state、mutations、actions、getters等

mutations唯一的目的就是修改state中状态

mutations中的每个方法尽可能完成的事件比较单一一点

有判断逻辑的代码也应该放到actions中 这样更好跟踪mutations中属性的改变

maGetters辅助函数仅仅是将store中的getters映射到局部计算属性

将getters转成计算属性computed

computed:{

//两种语法

//...mapGetters([“cartList”,”cartCount”])

...mapGetters({

//给方法起名字

count:”cartCount”,

list:”cartList”

})

}

0取返为true，10取返为false

dispatch返回一个Promise

# ES6中高阶函数的使用

filter() 方法创建一个新的数组，新数组中的元素是通过检查指定数组中符合条件的所有元素。

map() 方法返回一个新数组，数组中的元素为原始数组元素调用函数处理后的值。

map() 方法按照原始数组元素顺序依次处理元素.

reduce() 方法接收一个函数作为累加器，数组中的每个值（从左到右）开始缩减，最终计算为一个值。

# axios

JSONP的原理是什么呢？

因为它不能直接访问服务器上面的资料

JSONP的核心在于通过<script>标签的src来帮助我们请求数据

使用JSONP最主要的原因往往是为了解决跨域访问的问题

export function request(config){

return new Promise((resolve,reject)={

//1.创建axios实例

const instance=axios.create({

baseURL:”http://123.207.32.32:8000”,

timeout:5000

})

//发送真正的网络请求

instance(config).then(res=>{

resolve(res)

}).catch(err=>{ //前面都是回调函数方式，resolve reject(Promise方式)

reject(err)

})

})

}

二次封装

request({

url:”/home/multidata”,

}).then(res=>{

console.log(res)

}).catch(err=>{

console.log(err)

})

**拦截器的使用**

instance.interceptors.request.use

instance.interceptors.response.use

请求拦截的作用

//1.比如config中的一些信息不符合服务器的请求

//2.比如每次发送网络请求时，都希望在界面中显示一个请求的图标（动画）

//3.某些网络请求（比如登录（token）），必须携带一些特殊的信息

# Git

git clone 仓库地址

git status 查看状态

方式一

git .add 再执行git commit -m “初始化项目” 这次提交是提交到本地

需要提交到服务器 再执行 git push

方式二

git remote add origin 仓库地址

git push -u origin master

# 项目

修改文件路径 配置别名

脚手架2是在base.config.js文件里面修改

脚手架3创建一个vue.config.js文件

函数里面所有东西都是局部变量 一旦被调用执行完，里面所有定义的变量东西都会被内存回收掉。

ref如果是绑定在组件中的，那么通过this.$refs.refname获取的是一个组件对象

ref如果是绑定在普通的元素中，那么通过this.$refs.refname获取到的是一个元素对象 一定是准确的

修饰符.native修饰什么时候使用？

在我们监听一个组件的原生事件时，必须给对应的事件加上.native修饰符，才能进行监听

父访问子this.$refs调用子组件里面的方法

子访问父this.$parents调用父组件里面的方法

父传子props将父组件属性传递给子组件

子传父$emit将子组件的参数通过事件发送给父组件

因为涉及到非父子组件的通信，所以这里我们选择了事件总线

bus->总线

Vue.probetype.$bus=new Vue()

this.$bus.$emit(“事件名称”,参数)

this.$bus.$on(“事件名称”,参数)

'~'（波浪符号)：更新到当前中间位数字中最新的版本，例："vue": "~2.5.2"这个库会去匹配更新到2.5.x中最新版本，如果存在一个新的版本为2.6.0，则不会升到该版本。

'^'（插入符号)：更新到当前第一位数字中最新的版本。例："vue": "^2.5.2",这个库会去匹配更新到2.x.x中最新的版本，如果存在一个新的版本为3.0.0，则不会升到该版本。

# 刷新频繁的防抖动函数处理

对于refresh非常频繁的问题，进行防抖操作

没有防抖函数refresh需要调用30次

有防抖函数refresh可能只需要一次

防抖debounce/节流throttle

在methods中

debounce(func,delay){

let timer=null

//const refresh=funcrion(...args)就是debounce函数调用的结果

return function(...args){

if(timer){

clearTimeout(timer)

}

timer=setTimeout(()=>{

func.apply(this,args)

//func=this.$refs.scroll.refresh.apply(this,args)

},delay)

}

}

//refresh(“11111”,”222”)//因为...args可以传多个参数 this->vue

定时器是异步操作，所以会被拎出来放到事件队列里(定时器是macrotask)等待执行栈执行完以后再执行 js异步编程：单线程+事件队列 空闲时执行

在mounted中

const refresh=this.debounce(this.$refs.scroll.refresh,500)

//refresh不会被销毁 因为this.$bus.$on函数里面调用了refresh产生了闭包

在this.$bus.$on中调用refresh()

防抖函数起作用的过程：

如果我们直接执行refresh，那么refresh函数会被执行30次

可以将refresh函数传入到debounce函数中，生成一个新的函数

之后在调用非常频繁的时候，就使用新生成的函数，而新生成的函数，并不会非常频繁的调用，如果下一次执行来的非常快，那么会将上一次取消掉

第一次timer为null，不会执行if(timer),将setTimeout函数赋值给了timer

第二次timer有了第一次的timer，执行if(timer),将setTimeout函数赋值给了timer

第三十次就会执行func.apply(this,args),refresh函数

将debounce函数抽取到common/utils.js里面作为一个功能函数

# 图片懒加载

什么是图片懒加载？

图片需要现在在屏幕上时，再加载这张图片

# 响应式原理-依赖技术的分析和学习

面试题：

## 如何理解Vue生命周期

**开始创建、初始化数据、编译模板、挂载Dom、渲染→更新→渲染、销毁**等一系列过程，我们称这是Vue的生命周期。通俗说就是**Vue实例从创建到销毁**的过程，就是生命周期。

每一个组件或者实例都会经历一个完整的生命周期，总共分为三个阶段：**初始化、运行中、销毁**

## **生命周期各个阶段**

（1）**组件实例刚刚被创建：**实例、组件通过new Vue() 创建出来之后会**初始化事件和生命周期**，然后就会执行**beforeCreate**钩子函数，这个时候，**数据还没有挂载**呢，只是一个空壳，无法访问到数据和真实的dom，一般不做操作。

（2）**实例已经创建完成：挂载数据，绑定事件**等等，然后执行**created**函数，这个时候已经可以使用到数据，也可以更改数据,在这里更改数据不会触发updated函数，在这里可以在渲染前倒数第二次更改数据的机会，不会触发其他的钩子函数，一般可以在这里做**初始数据的获取。**

（3）**模板编译之前：**接下来开始找实例或者组件对应的模板，**编译模板为虚拟dom放入到render函数中准备渲染**，然后执行**beforeMount**钩子函数，在这个函数中虚拟dom已经创建完成，马上就要渲染,在这里也可以更改数据，不会触发updated，在这里可以在渲染前最后一次更改数据的机会，不会触发其他的钩子函数，一般可以在这里做初始数据的获取。

（4）**模板编译之后：接下来开始render，渲染出真实dom**，然后执行**mounted**钩子函数，此时，组件已经出现在页面中，数据、真实dom都已经处理好了,事件都已经挂载好了，可以在这里操作真实dom等事情...

（5）**组件更新之前：当组件或实例的数据更改之后**，会立即执行**beforeUpdate**，然后vue的虚拟dom机制会重新构建虚拟dom与上一次的虚拟dom树利用diff算法进行对比之后**重新渲染**，一般不做什么事儿。

（6）**组件更新完毕**：**当更新完成后**，执行**updated**，数据已经更改完成，dom也重新render完成，可以操作更新后的虚拟dom。

（7）**组件销毁前：**当经过某种途径调用**$destroy方法**后，立即执行**beforeDestroy**，一般在这里做一些善后工作，例如**清除计时器、清除非指令绑定的事**件等等。

（8）**组件销毁后：**组件的数据绑定、监听...去掉后只剩下**dom空壳**，这个时候，执行**destroyed**，在这里做善后工作也可以。

## 如何进行非父子组件通信

通过事件总线this.$bus.$on this.$bus.$off this.$emit

## Vue响应式原理

不要认为数据发生改变，界面跟着更新，并不是理所当然

app.message修改数据，Vue内部是如何监听message数据的改变

**Object.defineProperty**->监听对象属性的改变 观察者

name->Dep对象->Subs->[watcher1,watcher2]

const obj={

message:”哈哈哈”,

name:”why”

}

Object.keys(obj),forEach(key=>{

let value=obj[key]

Object.defineProperty(obj,key,{

set(newValue){

//告诉了谁？谁用告诉谁？谁在用了？{{message}}张三李四王五

//根据解析html代码，获取到那些人有用属性 发布

value=newValue

},

get(){

//张三，李四，王五 get 订阅 ->update

return value

}

})

})

//发布者

class Dep{

constructor(){

this.subs=[]

} //2.遍历watcher->通知变化(Dep) <-Observer<-data数据newVue()

addSub(watcher){

this.subs.push(watcher)

}

notify(){ //观察者data数据 订阅者el数据 发布者更新数据

this.subs.forEach(item=>{

item.update()

})

}

}

//订阅者

class Watcher(){

constructor(name){

this.name=name;

} //1.new Vue()el数据->Compile订阅数据变化，绑定更新函数->Watcher->Dep

update(){

}

}

const dep=new Dep()

const w1=new Watcher(“张三”)

dep.addSub(w1)

当数据发生改变，Vue是如何直到要通知哪些人，界面发生刷新

发布订阅者模式