

## 面试整理-3

- 1、常见的 http 状态码以及代表的意义
- 2、简要描述你对 AJAX 的理解
- 3、请介绍一下 XMLHttpRequest 对象
- 4、介绍一下 XMLHttpRequest 对象的常用方法和属性
- 5、简要描述 JavaScript 的数据类型?
- 6、解释一下 JavaScript 中的局部变量与全局变量的区别
- 7、简述 arguments 对象的作用
- 8、简要描述 JavaScript 中定义函数的几种方式
- 9、列举几个 JavaScript 中常用的全局函数，并描述其作用
- 10、清除缓存方法:
- 11、node:
- 12、ajax 过程
- 13、cookie 来实现购物车功能
- 14、命名规范
- 15、如何理解 html 标签语义化?
- 16、锚点的作用是什么? 如何创建锚点?
- 17、超级链接有哪些常见的表现形式?
- 18、link 和@import 都可以为页面引入 CSS 文件，其区别是?
- 19、什么是跨域，为什么浏览器会禁止跨域，实现跨域的几种方法
- 20、简要描述 JavaScript 中的自有属性和原型属性

## 1、常见的 http 状态码以及代表的意义

200: 请求已成功, 数据返回。302: 临时重定向, 理解为该资源原本确实存在, 但已经被临时改变了位置  
 303: 告知客户端使用另一个 URL 来获取资源。303 的使用场景几乎没有, 参考 302  
 400: 请求格式错误。1) 语义有误 2) 请求参数有误。  
 403: 服务器拒绝请求  
 404: 页面无法找到  
 500: 服务器内部错误  
 502: 服务器网关错误  
 304: 服务器资源未变化

## 2、简要描述你对 AJAX 的理解

AJAX 即异步的 JavaScript 和 XML。它是指一种创建交互式网页应用的网页开发技术, 可以实现页面的异步请求和局部刷新。

## 3、请介绍一下 XMLHttpRequest 对象

AJAX 的核心是 JavaScript 对象 XMLHttpRequest。该对象在 Internet Explorer 5 中首次引入, 它是一种支持异步请求的技术。简而言之, XMLHttpRequest 可以使用 JavaScript 向服务器提出请求并处理响应, 而不阻塞用户。实现页面的局部更新。  
 readyState 属性: 请求的状态, 有 5 个可取值 (0=未初始化, 1=正在加载, 2=以加载, 3=交互中, 4=完成)

## 4、介绍一下 XMLHttpRequest 对象的常用方法和属性

参考答案: open("method", "URL"): 建立对服务器的调用, 第一个参数是 HTTP 请求方式(可以为 GET, POST 或任何服务器所支持的您想调用的方式), 第二个参数是请求页面的 URL; send() 方法: 发送具体请求; abort() 方法: 停止当前请求; readyState 属性: 请求的状态, 有 5 个可取值 (0=未初始化, 1=正在加载, 2=以加载, 3=交互中, 4=完成); responseText 属性: 服务器的响应, 表示为一个串; responseXML 属性: 服务器的响应, 表示为 XML; status 属性: 服务器的 HTTP 状态码。

## 5、简要描述 JavaScript 的数据类型?

JavaScript 的数据类型可以分为原始类型和对象类型。

原始类型包括 string、number 和 boolean 特殊原始值: null (空) 和 undefined (未定义), 其中, 字符串是使用一对单引号

或者一对双引号括起来的任意文本; 而数值类型都采用 64 位浮点格式存储, 不区分整数和小数; 布尔 (逻辑) 只能有两个值: true 或 false。

Object 对象类型如 Function、Array、Date、Object 等。

## 6、解释一下 JavaScript 中的局部变量与全局变量的区别

全局变量拥有全局作用域, 在 JavaScript 代码的任何地方都可以访问; 在函数内声明的变量只在函数体内有定义, 即为局部变量, 其作用域是局部性的。

需要注意的是, 在函数体内声明局部变量时, 如果不使用 var 关键字, 则将声明全局变量。前提函数得调用, 声明提前赋值不提前

## 7、简述 arguments 对象的作用

arguments 可以访问函数的参数。表示函数的参数数组。arguments.length 参数个数, 其次, 可以通过下标 (arguments[index]) 来访问某个参数。

## 8、简要描述 JavaScript 中定义函数的几种方式

JavaScript 中, 有三种定义函数的方式:

1、函数语句: 即使用 function 关键字显式定义函数。如:

```
function f(x){return x+1;}
```

2、函数定义表达式: 也称为“函数直接量”。形如:

```
var f = function(x){return x+1;};
```

3、使用 Function() 构造函数定义, 形如:

```
Var f = new Function("x", "return x+1;");
```

运行结果:

```
function f() {console.log("function")}
function test() {
  console.log(f) //f() {console.log("function")}
  f()           //function
  f = "hello"
  console.log(f) //hello
  f()           //f is not a function
}test();
```

## 9、列举几个 JavaScript 中常用的全局函数，并描述其作用

1. parseInt: 解析一个字符串并返回一个整数;
2. parseFloat: 解析一个字符串并返回一个浮点数;
3. isNaN: 检查某个值是否是数字, 表示 不是一个数字, 返回 true 或者 false 不是一个数字如 a=' a' isNaN(a) true
4. encodeURI : 把字符串作为 URI 进行编码;
5. decodeURI : 对 encodeURI() 函数编码过的 URI 进行解码;
6. eval: 计算某个字符串, 以得到结果, 或者用于执行其中的 JavaScript 代码。

instanceof 来检测某个对象是不是另一个对象的实例。

**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。

```
function Car(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year;
}
```

```
const auto = new Car('Honda', 'Accord', 1998);
console.log(auto instanceof Car); //true
console.log(auto instanceof Object); //true
```

语法: object instanceof constructor

object: 某个实例对象

constructor: 某个构造函数

## 10、清除缓存方法:

(1) meta 方法用客户端代码使浏览器不再缓存 Web 页面:

```
<meta http-equiv="Expires" CONTENT="0">
```

```
<meta http-equiv="Cache-Control" CONTENT="no-cache">
```

```
<meta http-equiv="Pragma" CONTENT="no-cache">
```

(2) 用随机数或时间戳方法

在 URL?参数后加上??ran=?+?Math.random(); //当然这里参数?ran 可以任意取了

在?URL?参数后加上??timestamp=?+?New Date().getTime();

开发中: Network-Disable cache

```
<script>document.write('<script src="index.js?t=' + new Date().getTime()
+ '></script>')</script>
```

**11、node:**nodejs 是一个 javascript 的运行环境 运行在服务器,作为 web server 运行在本地,作为打包工具或者构建工具,Nodejs 基于 Javascript 语言,实现前后端统一语言,利于前端代码整合, nodejs 的性能是高于其他后台语言的,可以做缓冲来增加服务器端的总体性能。

NodeJS 集成 npm, 所以 npm 也一并安装了

Npm 是 nodejs 的包管理器 在 github 里进行下载、

应用场景: 做与服务的一些事、网站开发、im 即时聊天 (socket.io)、高并发

api (移动端, pc) \HTTP? Proxy\前端构建工具

## 12、ajax 过程

(1) 创建 XMLHttpRequest 对象, 也就是创建一个异步调用对象. (2) 创建一个新的 HTTP 请求, 并指定该 HTTP 请求的方法、URL 及验证信息. (3) 设置响应 HTTP 请求状态变化的函数. (4) 发送 HTTP 请求. (5) 获取异步调用返回的数据. (6) 使用 JavaScript 和 DOM 实现局部刷新.

## 13、cookie 来实现购物车功能

一、大概思路

1、从 cookie 中取商品列表 2、判断要添加的商品是否存在 cookie 中。3、如果已添加过, 则把对应的商品取出来, 把要添加的商品的数量加上去。4、如果没有添加过, 则把改商品添加到商品列表中。5、再把商品列表序列化, 加入 cookie 中。

22、gzip 优点是减轻了带宽压力, 缺点是加重了服务器的计算压力

## 14、命名规范

可读性~能看懂, 规范性~, 匈牙利命名的: 类型前缀, 首字母大写 a 数组, b 布尔, f 浮点, fn 函数, o 对象

-D 添加开发依赖 在开发周期有用, 发布没用 —— devDependencies

-S 添加 生产依赖 在生产环境下发布以后还需要的依赖

## 15、如何理解 html 标签语义化?

语义化的主要目的在于, 直观的认识标签 (markup) 和属性 (attribute) 的用途和作用。

可以概括为: 用正确的标签做正确的事情。

html 语义化可以让页面的内容结构化, 便于浏览器解析, 便于搜索引擎解析, 并提高代码的可维护度和可重用性。

比如, 尽可能少的使用无语义的标签 div, 使用结构化标签 <header>、<section>、<footer>。

## 16、锚点的作用是什么? 如何创建锚点?

锚点是文档中某行的一个记号, 类似于书签, 用于链接到文档中的某个位置。当定义了锚点后, 我们可以创建直接跳至该锚点 (比如页面中某个小节) 的链接, 这样使用者就无需不停地滚动页面来寻找他们需要的信息了。

在使用 <a> 元素创建锚点时, 需要使用 name 属性为其命名, 代码如下所示:

```
<a name=" anchorname1">锚点一</a><a href="#anchorname1">回到锚点一</a>
```

## 17、超级链接有哪些常见的表现形式？

<a> 元素用于创建超级链接，常见的表现形式有：

1、普通超级链接，语法为：

```
<a href="" target="">文本</a>
```

2、下载链接，即目标文档为下载资源，语法如：

```
<a href="DAY02.zip">下载</a>
```

3、电子邮件链接，用于链接到 email，语法如：

```
<a href="mailto:tarena@tarena.com.cn">联系我们</a>
```

4、空链接，用于返回页面顶部，语法如：

```
<a href="#">...</a>
```

5、链接到 JavaScript，以实现特定的代码功能，语法如：

```
<a href="javascript : ...">JS 功能</a>
```

## 18、link 和@import 都可以为页面引入 CSS 文件，其区别是？

将样式定义在单独的.css 的文件里，link 和@import 都可以在 html 页面引入 css 文件。有 link 和@import 两种方式，导入方式如下

link 方式：<link rel="stylesheet" type="text/css" href="aa.css">

@import 方式:<style type="text/css">@import "aa.css";</style>

## 19、什么是跨域，为什么浏览器会禁止跨域，实现跨域的几种方法

1、什么是跨域

跨域的产生来源于浏览器所的‘同源策略’，所谓同源策略，是指只有在地址的：

1. 协议名 https, http

2. 域名 http://a.study.cn http://study.cn

3. 端口名 http://study.cn:8080/json/jsonp/jsonp.html study.cn/json/jsonp/jsonp.html

均一样的情况下，才允许访问相同的 cookie、localStorage 或是发送 Ajax 请求等等。若在不同源的情况下访问，就称为跨域。

2、为什么浏览器会禁止跨域

跨域的访问会带来许多安全性的问题，比如，cookie 一般用于状态控制，常用于存储登录的信息，如果允许跨域访问，那么别的网站只需要一段脚本就可以获取你的 cookie，从而冒充你的身份去登录网站，造成非常大的安全问题，因此，现代浏览器均推行同源策略。

实现跨域：

1、Jsonp：其背后原理就是利用了 script 标签不受同源策略的限制，在页面中动态插入了 script，script 标签的 src 属性就是后端 api 接口的地址，并且以 get 的方式将前端回调处理函数名称告诉后端，后端在响应请求时会将回调返回，并且将数据以参数的形式传递回去。

基于 script 标签实现跨域

```
<script type="text/javascript">
    var jshow = function (data) { alert(data.s); };
    var url = "https://sp0.baidu.com/5a1Fazu8AA54nxGko9WTAnF6hhy/su?wd=a&cb=jshow";
    var script = document.createElement('script');
    script.setAttribute('src', url);
    document.getElementsByTagName('head')[0].appendChild(script);
</script>
```

2. document.domain

这种方式只适合主域名相同，但子域名不同的 iframe 跨域。

比如主域名是 `http://crossdomain.com:9099`，子域名是 `http://child.crossdomain.com:9099`，这种情况下给两个页面指定一下 `document.domain` 即 `document.domain = crossdomain.com` 就可以访问各自的 window 对象了。

`substr()` 方法可在字符串中抽取从 `start` 下标开始的指定数目的字符。

`stringObject.substr(start, length)`

阻止事件冒泡: `e.stopPropagation, cancelBubble=true`

阻止默认行为: `e.preventDefault returnvalue=false`

`==` `===` 判断值是否相等，后者值和类型是否相等

## 20、简要描述 JavaScript 中的自有属性和原型属性

参考答案:

自有属性是指，通过对象的引用添加的属性，此时，其它对象可能无此属性。对于自有属性，是各个对象所特有的、彼此独立的属性。比如: `empl.job = 'Coder'`; 原型属性是指从原型对象中继承来的属性，一旦原型对象中属性值改变，所有继承自该

原型的对象属性均改变。比如: `Emp.prototype.dept = '研发部'`;

当需要检测对象的自有属性时，可以使用 `hasOwnProperty()` 方法。另，还可以使用 `in`

操作检测对象及其原型链中是否具备指定属性。需要注意的是，在检测对象属性时，先检测自有属性，再检测原型属性。

## ES6

- 1、关于定（声明）变量：
- 2、解构赋值：
- 3、箭头函数
- 4、Array 与 JSON 方法
- 5、字符串模板
- 6、json 写法、JSON 对象
- 7、面向对象
- 8、ES6 模块系统
- 9、Promise
- 10、ES6 新特性及它们作用

## 1、关于定（声明）变量：var、let 和 const

(1)var 是函数作用域。在函数内声明了 var，整个函数内都有效,for 里定义 var 在循环外也可以访问，有声明提前，赋值不提前，

(2)let 是块级作用域。没有声明提前、先定义在使用，不能重复定义声明，赋值可以

(3)const 常量 定义完变量，必须有值，不能后赋值，不能修改，块级作用域，也没有声明提前

在之前 JS 是没有块级作用域的，const 与 let 填补了这方便的空白，const 与 let 都是块级作用域。替代闭包，

不可以：let a=12; let a=5; //不能重复声明定义，报错 has already been declare

可以：let a=12;a=5; alert(a) //5，变量可改

Let 能替代闭包

点 abc 分别 输出 3

```
<input type="button" value='a'>
<input type="button" value='b'>
<input type="button" value='c'>
<script>
for(var i=0;i<aBtn.length;i++){
    aBtn[i].onclick = function(){
        alert(i)
    }
}
Console.log(i)//3
</script>
```

点 abc 分别 输出 0 1 2，

解决方法一:let

```
for(let i=0;i<aBtn.length;i++){
    aBtn[i].onclick = function(){
        alert(i)
    }
}
Console.log(i)//012
}
Console.log(i)//i is not defined
```

解决方法二：闭包

说明：var 作用域是函数，为了强行弄出三个 i 包个函数，for 循环几次就有几个函数执行就有几个 i，这样每个 i 就都属性自己独立函数里的，不是 window.load 的

点 abc 分别 输出 0 1 2，

```
for(var i=0;i<aBtn.length;i++){
    (function(i){
        aBtn[i].onclick = function(){
            alert(i)
        }
    })(i)
}
```



## 2、解构赋值：

- (1) 两边的结构必须一样
- (2) 右边必须是个合法东西
- (3) 赋值和解构必须同时完成

针对上述错误写法：

```
let {a,b} = [12,6]; console.log(a,b); //X 两边结构不一样,
let {a,b} = {12,5}; console.log(a,b); //X 右边不是 json 也不是数组,
let {a,b};{a,b} = {a:12,b:5} //X 需要同时完成
let {a,b} = {a:12,b:5}
```

**解构扩展**（参数扩展(收集，展开)、数组展开、json 展开）**展开用表示用 ...**

## 3、箭头函数 作用：

- (1) 简写, (2) 修正 this, 固定 this, this=>当前的环境（在哪个对象环境里执行或取决于你这句话所执行时 this 是什么）普通函数 function 写法的 this 跟着执行人走谁执行就是谁，箭头函数的 this 固定不动，取决于在哪声明的这个函数（除非用 call, apply, bind 强行改变 this 指向）普通 function: this 跟着执行人走

箭头函数注意：

- 1. this 问题，定义函数所在的对象或是父级对象是谁，不在是运行时所在的对象如 window
- 2. 箭头函数里面没有 arguments, 用 '...'
- 3. 箭头函数不能当构造函数

## 4、Array 与 JSON 方法

### 原生对象扩展

Array 扩展: map, reduce, filter, forEach \模板字符串\json 写法,

**map()** 映射一一对应: 返回新数组，原数组处理后的值。按原数组顺序依次处理（简单说就是进去 10 个出来还是 10 个，一一对应）

```
let arr = [68, 53, 12, 98, 65];
var result = arr.map((item) => {
  return item > 60 ? '及格' : '不及格'
})
console.log(result) //结果: ["及格", "不及格", "不及格", "及格", "及格"]
```

**reduce()** 作为累加器，数组中的每个值从左到右缩减成一个值。一般用在求和，n>1(多个值变成一个值)

```
let arr = [1, 2, 3, 4];
// 第一次: total 是 1, 当前值是下一个 2, 依次; 结果: 求和 1, 当前 2, 索引 1
let result = arr.reduce((total, curVal, curIdx, arr) => {
  return total + curVal;
})
console.log(result) //结果: 10
```

**filter** 根据条件来筛选过滤，进去多个出来几个不一定

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8];
var result = arr.filter(item => {
  return item % 2 == 0;
})
console.log(result); //结果: [2, 4, 6, 8]
```

forEach 遍历就是循环 突出所有的都走一遍，进去几个出来几个，没返回值，

```
let arr = [1,2,3];
var result = arr.forEach((item,index)=>{
    console.log(`第${index}个是${item}`);
})
/*
//结果:
    第0个是1
    第1个是2
    第2个是3
*/
```

## 5、字符串模板

``${变量}`` 反单引号 代替字符串拼接，可以换行

## 6、json 写法、JSON 对象

JSON.stringify() 序列化，给一个 json，出来一个字符串，字符串写法 JSON.stringify() 结果有单引号  
如： `{ "a":12, "b":5, "name":"blue" }`

JSON.parse() 给一个字符串 进行解析，还原成 json

JSON.stringify({a:12,b:5}) => `{ "a":12, "b":5 }` console.log 中''省略了，出来的里边都有双引号

JSON.parse('{"a":12,"b":5}') => 结果: `{a: 12, b: 5}` parse() 里必须是里层双引外层单引号

### 错误写法

let arr = `{a:12,b:5}`; console.log(JSON.parse(arr)) //报错，里边必须加双引号"a" "b"

JSON.parse(`{a:12,b:5,'name':'blue'}`) //报错，必须是外边单引，里边双引

## 7、面向对象 ES5 与 ES6 的区别?

ES5 面向对象一假的

ES5 没有系统统一的写法，处于自己摸索的状态（例如两个人写自己的库，一互用就会有问题了）ES5 中没有 class 这一说法，它是用函数完成的功能，使用函数声明类，Person 即是类也是构造函数，

### ES6 面向对象-优点

完全解决了统一的问题

提供了四个新的关键字，用于解决上面的问题

1) class : 类声明 2) constructor: 构造函数/构造器 3) extends: 继承 4) super: 超类/父类

//有单独的类声明，构造函数声明

```
class Person{ constructor(){ } }
```

```
继承: Class Worker extends Person { constructor(){ super() } }
```

例：Es5 继承

```
//父类 Person 子类 Worker 有单独的 showJob 在单写，其他的继承 Person
function Person(name, age) {
    this.name = name; this.age = age;
}
Person.prototype.showName = function() { console.log(this.name)}
Person.prototype.showAge = function() { console.log(this.age)}
function Worker(name, age, job) {
    Person.call(this, name, age); //call 方法继承 person 属性和方法
    this.job = job;
}
Worker.prototype = new Person(); // worker 类也就继承了
Worker.prototype.constructor = Worker;
Worker.prototype.showJob = function() {
    console.log(this.job)
}
// 继承父类方法
var w = new Worker('YJUI', 18, '随便');
w.showName();
w.showAge();
w.showJob();
```


例：Es6 继承

```
// class 类声明，构造函数声明 constructor
class Person {
    constructor(name, age) {
        this.name = name; this.age = age;
    }
    showName() {console.log(this.name)}
    showAge() {console.log(this.age)}
}
class Worker extends Person {
    constructor(name, age, job) {
        super(name, age); //继承的属性放里
        this.job = job; //子类自己的属性
    }
    // 子类自己的方法
    showJob() {console.log(this.job)}
}
var w = new Worker('YJUI', 18, '随便');
w.showName(); //结果 YJUI
w.showAge(); //结果 18
w.showJob() //结果 随便
```

## 8、ES6 模块系统

export (输出) import(导入)

**注意:**

 必须要写, 因为 webpack 是 nodejs 写的东西, 必须遵循 nodejs 的规定

**多用在** webpack.config.js 中 module.exports = {}

需要对外输出 json ——> {}, CMD 写法主要给 nodejs、nodejs 遵循 CMD,

./ 用 webpack 就必须写, 指当前文件(按 ES6 本身标准可以不写)但现在用 webpack 编译, webpack 是 nodejs 写的要遵循 nodejs 约定, 也就是说 ./ 是 nodejs 的规定,

import \* as mod1 from './mod1'; as 导入所有成员取个共同的名字叫 mod1, 从当前目录下 mod1.js 取

导出 (export) 的几个情况: 可以导出变量、常量、一堆变量、函数, class

例: import 导入 mod1.js 中 export 内容

Export:

mod1.js: export let a=2

index.js : import {a} from '@assets/mod1' 取 a

或 import \* as name from '@assets/mod1' 取 name.a

Export default:

Mod1.js: let a=4; let b=6; export default b;

Index.js: import mod from '@assets/mod1'; console.log(mod) //6

(1) export 和 export default 的区别

Export: import 输出要加 {} 一个也加 {}, export 名与 import 名必须一致对应, 不加 {} 可以起别名用

as, import \* as name from './mod1.js';

export default: import 输出名随便, 一个模块只有一个默认输出, 只能用一次, 所以 import 后不用 {}, 唯一对应

import \* as mod1 from 'xxx' // export 与 export default 都可以 as 后名随意, 区别在于 export 取值 mod1. 变量名; export default 取值 mod1.default. 变量名

import ./ 用 webpack 就必须写, 指当前文件(按 ES6 本身标准可以不写)但现在用 webpack 编译, webpack 是 nodejs 写的要遵循 nodejs 约定,

## 10、Promise: 解决异步回调问题

传统方式解决异步回调问题方式: 大部分用回调函数, 事件, 相当于 ajax 嵌套

异步操作: 同时进行多个操作, 用户体验好(如用户名检查, 输入完就检查). 异步缺点: 好用但写起来麻烦。

同步操作: 一次只能进行一次操作, 用户体验不好, 按顺序执行, 优点: 清晰,

## 9、Promise

即有异步操作优势（不用卡住可同时进行多个操作）也可以像同步一样简单的写法

异步：then 成功回调-> resolve=>解决，失败回调->reject=>拒绝

```
let p = new Promise(function(resolve, reject) {
    $.ajax({
        url: 'data/1.txt',
        dataType: 'json',
        success(arr) { resolve(arr); },
        error(res) { reject(res); }
    })
})
p.then(function(arr) {
    alert('成功')
    console.log(arr) //12, 5, 8
}, function(res) {
    alert('失败')
})
```

总结：Promise 本身不能算是对异步操作的处理只是一个封装，因为不同的异步操作的表现形式不一样，Promise 只是给一个统一的格式，统一模板，按这个封装，不管是什么，都有 resolve 和 reject，调取就可以、Jquery 的 \$.ajax 本身就是一个 Promise，直接用 then 的写法

```
$.ajax({
    "url": 'data/1.txt',
    dataType: "json"
}).then(arr=>{ //成功
    alert(arr)
}, res=>{ //失败
    alert('失败')
})
```

**Promise.all([])**：统一做一个 then。必须全都成功，有一个失败了全失败。要求所有东西都读取完了之后会给一个统一的结果，里边是数组。Promise.all() 虽然好但不能解决所有的问题，

```
Promise.all([
    $.ajax({url: "data/1.txt", dataType: "json"}),
    $.ajax({url: "data/2.txt", dataType: "json"}),
    $.ajax({url: "data/3.txt", dataType: "json"})
]).then(arr=>{
    console.log(arr)
    // alert('成功')
}, res=>{
    alert('失败')
})
```

**Promise.all()不能处理的情况:** Promise.all() 一门心思读到底，这几个全都读下来。并不是说第一个读完判断一下，然后在读后边的，这种 promise.all() 处理不了；

有一种情况不能用 Promise.all()，就是根据前一请求数据来读后一请求数据时不适合，利用前边数据来指导后边的数据:X

如非要用 Promise 处理这样的逻辑只能这样表示，但和原始的 ajax 请求没什么太大的区别

```
ajax('http://taobao.com/api/user').then(user_data=>{
  if(user_data.vip){
    ajax('http://taobao.com/api/vip_items').then()
  }
}, error=>{
  alert('error')
})
```

#### async/await 特点:

async 虽然本身很特殊但调用的时候当普通函数用就行

async 是函数的一个特殊形式，是一个语法，表示声明函数中是包含异步操作的，

await 哪个是异步的哪个是同步的，程序不知道，只要加上 await 表明就是异步的，是等待的意思，有 await 标注那一行要等待操作结束后再往下走，也可以顺便把数据收集起来 let data=await \$.ajax();

只是给函数加了个修饰，告诉编译器这不是普通的函数，是有一些需要暂停的操作，会用 await 标记出来，现在写代码习惯中所有的异步操作都用了 async/await

普通函数：一旦开始运行就不会停，直到代码执行完，

async 函数-能够“暂停”，是语法糖，会把这个大函数拆分成很多的小函数，执行第一个也是从头到尾，执行完要等着操作完成再执行第二个小函数，

async 会暂停在执行，碰着 await 就暂停一会，到哪执行到哪等待是由开发人决定的

语法糖：往往给程序员提供了更实用的编码方式，有益于更好的编码风格，更易读。

写法和同步一样方便，但是异步的，\$.ajax 执行完之后也执行了 then

```
async function show(){
  xxx; xxx;
  let data = await $.ajax();
  xxx;
} show();
```

Object.assign(target, source);

一个或多个源对象分配到目标对象，第一个值是目标对象，最后合并值都放 target 里

let json = Object.assign({}, defaults, options); //{} 是 target，后两个值合并后还是以前值

```
const target = { a: 1, b: 2 };
const source = { b: 4, c: 5 };
const result = Object.assign(target, source);
console.log(target, result)//都是 {a: 1, b: 4, c: 5}
console.log(source)// { b: 4, c: 5 };
Object.is(null, null); // true
Object.is([], []); // false
Object.is(0, -0); // false
Object.is('foo', 'foo'); // true
```

## 10、ES6 新特性及它们作用

A) 类 继承 : (class, constructor, extends, super) 让 JavaScript 的面向对象编程变得更加简单和易于理解

B) 模块化: 主要由 export 和 import 组成, 每一个模块都有自己单独的作用域, 模块之间的相互调用关系是通过 export 来规定模块对外暴露的接口, 通过 import 来引用其它模块提供的接口

C) 箭头函数=>function 简写, 箭头函数与包围它的代码共享同一个 this, 能帮你很好的解决 this 的指向问题. 借助=>, 就不需要诸如 var self = this; 或 var that = this 这种模式了

D) 模板字符串: 使得字符串的拼接更加的简洁、直观。`\${变量}` 完成字符串的拼接,

E) 解构赋值: JavaScript 的一种表达式, 可以方便的从数组或者对象中快速提取值赋给定义的变量。

F) 延展操作符(展开操作符): ... 可以在函数调用/数组构造时, 将数组表达式或者 string 在语法层面展开;  
函数调用: myFunction(...iterableObj);

数组构造或字符串: [...iterableObj, '4', ...'hello', 6];

构造对象时, 进行克隆或者属性拷贝: let objClone = { ...obj};

G) 对象属性简写: 在 ES6 中允许我们在设置一个对象的属性时不指定属性名。

不用 es6: const student = { name: name, age: age, city: city};

用 es6: const student = {name, age, city};

H) Promise: Promise 是异步编程的一种解决方案, 比传统的解决方案 callback 更加的优雅, ES6 将其写进了语言标准, 统一了用法, 原生提供了 Promise 对象。

不使用 ES6, 嵌套两个 setTimeout 回调函数: hello 1s 后在 hi, 很慢

```
setTimeout(function () {
  console.log('Hello'); // 1 秒后输出"Hello"
  setTimeout(function () {
    console.log('Hi'); // 2 秒后输出"Hi" }, 1000);
  }, 1000)
})
```

使用 ES6, 先 hello 在 hi, 很快

用两个 then 来进行异步编程串行化, 避免了回调地狱:

```
var waitSecond = new Promise(function (resolve, reject) {
  setTimeout(resolve, 1000);
});
waitSecond.then(function () {
  console.log("Hello"); // 1 秒后输出"Hello"
  return waitSecond;
}).then(function () {
  console.log("Hi"); // 2 秒后输出"Hi"
});
```

J) Let 与 Const

在之前 JS 是没有块级作用域的, const 与 let 填补了这方便的空白, const 与 let 都是块级作用域。替代闭包,

Node. js

### 1、 介绍

nodejs 运行在服务器端，前后端代码整合

Node. js 有自己的模块系统，因为 Node. js 早于 ES6 出现，nodejs 是遵循 CMD 规范，所以对 CMD 比较熟悉的就好上手很多【cmd 有 requirejs 等】

使用：安 node. js 自带 npm,

`npm install -g xxx`

### 2、 npm 和 cnpm 的区别？

npm 的源在国外

cnpm 在国内 【c 不是 china】

npm 和 cnpm 装的包是不能混用的，同一个项目就用其中一个，用注了，一混用就会有问题



## VUE

- 1、vue 生命周期:
- 2、vue 指令
- 3、事件修饰符
- 4、filters--过滤器
- 5、computed ——计算后的数据
- 6、watch ——监听数据的修改-没 return
- 7、组件
- 9、:class 的几种写法
- 10、sync
- 11、slot 插槽应用扩展: 占位符
- 12、Router

**1、vue 生命周期：**

beforeCreate 组件实例刚刚被创建, 属性都没有

created 实例已经创建完成, 属性已经绑定

beforeMount 模板编译之前 (准备)

mounted 模板编译之后, 代替之前 ready \*

beforeUpdate 组件更新之前 data 数据变了 (用在\$.watch('a',function) {})

updated 组件更新完毕 \* (用在\$.watch('a',function) {})

beforeDestroy 组件销毁前

destroyed 组件销毁后

Vue 钩子函数: created mounted updated Destroy

页面初次加载执行生命周期 beforeCreate created beforeMount mounted

**2、vue 指令**

**v-bind:** 用于属性的单向绑定, 可简写为 ':' v-bind:title="a"

**v-on:** 用于事件的绑定, 可简写为 '@'; v-on:click ==@click

**v-model** 双向绑定 多用于 input select

**v-text:** 同插值表达是作用一样, 但是会覆盖原本的内容;

**v-html:** 将内容以 html 元素渲染;

**v-for** 循环输出 v-for="(val,key) in data", key 作用: 区分元素、提高性能

**v-if/v-else/v-else-if**

**v-once** 只会渲染解析一次

**v-pre** 预编译指令让 vue 跳过这个节点不编译原样输出。写书文档能用到

**v-if** 元素真的被删掉, 只剩下一行注释, <!-->占位符

**v-show** 元素只是隐藏了, display

**v-cloak** 防止页面加载时出现 vuejs 的变量名而设计。解决 vue 代码加载闪烁问题

```
[v-cloak]{
  display: none;
}
```

2、v-if 与 v-show 区别?

**v-if** 元素真的被删掉, 只剩下一行注释, <!-->占位符

**v-show** 元素只是隐藏了, display

**v-show** 用于频繁显示隐藏, 【隐藏在显示比直接删除显示快】**v-if** 用的更多。大量的隐藏-也会影响性能。

**v-show** 某些元素隐藏了也会起作用-比如: 表单, 【v-if 不会有这个问题】

**事件**

**v-on** v-on:click="xxx" 等价于 @click="xxx"

**3、事件修饰符**

(1). stop—阻止冒泡 @click.stop

(2). prevent—阻止默认事件:

按键盘这个行为要干掉它, @keydown.prevent

如表单提交按钮 submit 或在页面上点右键, 自动出现下拉菜单

(3). self—只接收自身的事件 (冒泡上来的不要) 嵌套点击中父级加.self

阻止冒泡二选一像.self 放 div 或.stop 放 button 上

```
<div @click.self='divFn'>
  <button @click.stop='btnFn'></button>
```

```
</div>
```

总结: `.self` 和 `.stop`

一个事情有多种实现方法, 结果一样, 一是在 `.self` 外边加, 一个在 `.stop` 里边加

`.once`: 事件只触发一次;

`.passive` 告诉浏览器你不想阻止事件的默认行为

`.capture`: 捕获冒泡

加,outer inner middle ,都不加 inner middle outer, 加的从外向里顺序输出

```
<div @click.capture="outer">
  <div @click="middle">
    <button @click.capture="inner">点击</button>
  </div>
</div>
```

#### 4、filters——过滤器

作用: 接收输入的数据→转换→输出的结果 有 `return`

```
filters: { } { { a | Upper } }
Upper: function (value) { return value.toUpperCase() },
```

例: 时间戳转日期格式、千位分隔符、转成万单位、首字母大写、保留两位小数

解答: 为什么这里用 `filter` 要比用 `methods` 好?

(1) `filters` 本身专事专用, 仅是用来做转换的, 用 `filters` 有个预期是要开始转数据了; `methods` 本身可以做任何事情,

(2) `filters` 语法更简洁

#### 5、computed ——计算后的数据——在真实的数据之外包一层有 `return`

应用场景

(1) 简单的一些小计算可以直接用模板内的表达式计算, 比较复杂一点的就建议使用“计算属性来运算了”, 也方便后期的维护;

(2) `computed` 适合比较单纯的数据改动, 处理完后返回一个新的数据 `return`, 页中使用新变量

(3) 每个计算属性都有一个 `getter` 函数 和 `setter` 函数, 下面的示例只是用了 `computed` 的唯一默认属性, 就是 `getter`, `setter` 一般用来手动修改数据

默认的写法是 `get()`, `set()` 可以改变新数据值

(4) 写法:

```
computed 与 watch 都可以这样表示: a() {} 同于 a:function() {} 同于 'a'() {} 同于
'a':function() {}
```

作用:

1、可以控制对数据的操作

2、缓存: 当数据不变, `get` 每次都不用重新计算, 如算税有很多没变用上, 快很多(避免无效的计算), 只有在相数据发生改变时才会重新求值执行函数。

3、想改变 `a` 的值, 要设置 `set()`, 不加 `set` 的 `a` 只有 `get()` 是只读的。例中加 `set` 后 `vm.a` 设置成 5 倍数 `{{a}}` 就会变了。不加 `set` 改 `vm.a` 值会提示没设 `setting` 报错。

直接这样写 `a() {return this.true_a+5}`, 同于 `get()` 里的,

应用场景: 加 `set` 才能改变 `a` 值, 在 `console.log` 改 `vm.a` 只有 5 的倍数才能改变页中 `a`

```

<div id = "app">{{a}}</div>
var vm =new Vue({
  el:'#app', data: {true_a:5},
  computed:{
    a(){return this.true_a+5 } //同于 get() 里的
  },
  a:{
    get(){ return this.true_a+5 },
    set(val){ if(val%5==0){ this.true_a = val; }}
  }
})

```

缓存: Price 不变不执行 total 函数这就叫缓存{{total}}。改 vm.price 可与页面同步。改 vm.total 报错, 加上 set 可以 set(val){this.price=val;}能同步 vm.total

```

var vm =new Vue({
  el:'#app',
  data:{ price:10,},
  computed:{ total:function(){return this.price+100; }}
})

```

#### 6、watch ——监听数据的修改-没 return

watch 函数名必须和 data 名一样, 监听 data 里数据。接收两个参数值, (newval,oldval) 变化后值和变化前值

类似于事件——当某个数据被修改了, 可以得到通知, **有深度监听和浅度监听**

浅度监听: watch 在默认的情况下只能监听表层 vm.json=?会出现变了, vm.json.a 监听不到不现在变了,

data:{ json:{a:12,b:5} },watch:{ json(){ console.log('json 变了') } }

深度监听 vm.json 与 vm.json.a 都触发 watch: deep:true; 也监听内部; 性能不高

immediate 讲解作用: true 在程序初始化之后 watch 就立即发生一次执行 “**json 变了**”

用在页码上初始 handler(){ ajax(...) //请求数据操作 }

```

watch:{
  json:{
    deep:true,
    immediate:true,
    handler(){
      console.log('json 变了')
    }
  }
}

```

**如果即希望深度监听也希望性能高些:** 选择性去做, 浅层的够用就用浅层的, 精确盯着某个要修改的值, 用浅度监听找监听的值, vm.json 与 vm.json.a 都会监听到

```

Watch:{
  'json.a':function(){
    console.log('json 变了')
  }
}

```

**watch 监听数组**

data: { arr: [1, 2, 3, 4, 5] }, watch: { 'arr.2' () { console.log('数组变了') } },

watch 监听数组某个下标值 vm.arr.2=? 2 是下标也不会触发 watch, 只有 vm.arr.push(6) 有变化。因为 vue 对 json 【可以精准某个值用浅度写法 vm.json.a】和数组 【精准浅度写法也无用】。要用 vm.\$set(数据, key, val) Vue.set(数据, key, val) 方法

vm.\$set(vm.arr, 2, 33) -> [1, 2, 33, 4, 5]

**watch 需要注意的: 不要循环 watch**

watch: { a() { this.a++; } }, 这样不可

**7、组件**

全局组件——任何地方: Vue.component('xxx', { data() { return {}; }, })

局部组件——父组件之内 components: {}

类声明组件——var cmp = Vue.extend({}) 挂局部上/全局上 Vue.component('cmp', cmp)

动态组件——is

组件传参

组件注意:

- 1、组件要写在 vm 实例之上,
- 2、中划线组件名 页中 <login-dialog></login-dialog> 组件 js 引入可以 loginDialog
- 3、new Vue 能用的, 组件也能用——filters、computed、watch、\*
- 4、只有在全局组件中能用 props

注意 页中不能是 <myButton></myButton> 当然对应的 js 中 myButton 与 my-button 都不行

页中能是 <my-button></my-button> 对应 js 中 myButton、my-button 都行

总结 html 中组件名不能是中间大写的形式 (如: myButton) 要用横岗代替 (如: my-button)。

对应 js 都行。

全局与局部组件写法:

```
<login-dialog :aa='a'><aaa></aaa></login-dialog>
结果: 这是全局组件 12——我是局部组件 12
Vue.component('loginDialog', {
  props: ['aa'],
  template: `<h1>这是全局组件 {{aa}}<aaa :bb = "aa"></aaa></h1>`,
  components: {
    'aaa': {
      props: ['bb'],
      template: `<span>——我是局部组件 {{bb}}</span>`,
    }
  }
})
```

局部组件写法:

1、局部组件放 vm 实例的 components 中

2、全局组件里放局部组件数据用:aa='a' 的 props 传, 全局与局部组件都能用 props

3、用类的方式 var cmp = Vue.extend({}) 把 cmp 放 vm 的局部 components

```
let vm = new Vue({
  el: '#root',
```

```

        data:{ a:12},
        components:{
          'aaa':{ props:['aa'],
                  template:`<span>我是局部组件{{aa}}</span>`,
                }
        }
      })
    var child={
      template:`<button @click="add">我是局部组件{{a}}</button>`,
      methods:{ add(){ this.a++;}}
    };

    let vm = new Vue({
      el:'#root',
      components:{
        'loginDialog':child //loginDialog:child 加不加引号都行
      }
    })

```

类声明组件 Vue.extend({})

```

let name= Vue.extend({ //类声明组件
  template:`<span>abc</span>`
})
//挂成全局组件
Vue.component('name',name)
let vm = new Vue({el:'#root',
  // 挂局部上
  components:{ cmp}
})
//导出写法用 Vue.component('vue-head',Vue.header)或
components:{'vue-head':Vue.header } <vue-head :aa="a"></vue-head>
(function(vue){
  var template = `<div>{{aa}}</div>`
  //head 组件名
  var head = vue.extend({
    template:template,
    props:['aa'],
  })\
  vue.header = head; //vue 上放 header
})(Vue)

```

## 动态组件

作用：1、改变挂载的组件,用 is 属性来切换组件<component :is="组件名"></component> 可以用 v-for

```
<component :is="组件名"> </component>
```

7、Props 两种使用方法:

1、子组件接收父组件传的值 props:[ 'msg' ]

2、参数约束: 具体约束类型, 是否必传, 范围操作

default:默认值如:leixing 不传时 required:false

```
<my-button :leixing="a"></my-button>
props: {
  leixing: {
    type:String,      //a 类型必须为 String
    required:true,    //为 true 必须传:leixing='a'
    validator(arg) {  //接收 leixing 值的范围判断
      //这里 arg 是 default 输出子组件 default, 不是就 false 报错
      if(arg==='default'){
        return true;
      }else {
        return false;
      }
    }
  }
},
```

9、:class 的几种写法

(1) 字符串 : :class ="abb"

(2) 数组 : :class ="['box','aa','bb']"

(3) json : :class ="{ box:true, aaa:false, bbb:true }"

(4) 组合 : :class ="['btn','bnt-dark',{active:index==i}]"

(5) 三目 : :class ="i==value?'active':'"

(6) 多个动态 class :class ="` \${'left'+key} \${key<signinDays?'active':''} `"  
signinDays:[0,1,2,3,4,5,6] key 是索引值 left0 active

9、V-model

父组件 v-model="a" 等同于 : value="a" @input="fn"

子组件是@input ="\$emit('input',\$event,target.value)"

## 10、sync

(1). sync 作用

v-model 是实现一个属性双向绑定的, 那多个时就用. sync

. sync 是 vue 中用于实现简单的“双向绑定”

vue 的 prop 是单向下行绑定: 父级的 prop 的更新会向下流动到子组件中, 但是反过来不行。可是有些情况, 我们需要对 prop 进行“双向绑定”。这个时候, 就可以用. sync 来解决

说明:

v-model 是@input 写死的一个标签上就一个, 实现一个 tab, @update:xx 要多少有多少, 只要 xx 变了就有。 . sync 一个标签能有多个实现多个 tab

.sync 的原理和 v-model 差不多，只是在 v-model 的基础上扩展了要多少有多少。

(2). sync 语法：

即然什么名都行，那就往语法上靠

:xx.sync = :xx="value"+@update:xx ="赋值"

这里 update:a/update:b 就是事件

```
父:a.sync=cur1 :b.sync=cur2
等同于 :a="a" :b="b" @update:a="val=>a=val" @update:b="val=>b=val"
子: this.$emit('update:a',this.a+1); this.$emit('update:b',this.b+1);
```

(3). sync 用法用例：

<text-document:title.sync="doc.title"></text-document> 当子组件需要更新 title 的值时，它需要显式地触发一个更新事件：this.\$emit('update:title', newValue)

这样 title 的属性在子组件内部更新，父组件也能感知的到，实现了“双向绑定”。

例. 在子组件点击每次都+1，实现双向绑定

```
<div id="root" v-cloak>a={{cur1}} b={{cur2}}
<cmp1 :a="cur1" :b="cur2" @update:a = "fn1" @update:b = "fn2"></cmp1>
<cmp1 :a="cur1" :b="cur2" @update:a="val=>cur1=val" @update:b="val=>cur2=val" ></cmp1>
>
同于<cmp1 :a.sync="cur1" :b.sync="cur2" ></cmp1>
</div>

<script src="vue2.js"></script>
<script type="text/javascript">
var cmp1 = Vue.extend ({
  props:['a','b'],
  template:`<div><input type="button" @click="fn" value="+1 按钮" /></div>`,
  methods:{
    fn(){
      this.$emit('update:a',this.a+1);
      this.$emit('update:b',this.b+1);
    }
  }
})
let vm = new Vue({
  el:'#root',
  data:{
    cur1:0,cur2:0
  },
  components:{ cmp1},
  methods:{//其它俩种写示不加 fn1/fn2
    fn1(val){ this.cur1= val; },
    fn2(val){this.cur2= val; }
  }
})</script>
```



## 小结

一个组件需要提供多个双向绑定的属性时使用，只能选用一个属性来提供 `v-model` 功能，但如果还有其他属性也要提供双向绑定，就需要 `.sync`

## 11、slot 插槽应用扩展：占位符

**作用：**vue 中，经常需要向一个组件传递内容。为了解决这个问题，官方引入了插槽(slot)的概念。不用插槽内容会丢失

`<cmp1>`直接这么写接收不到`</cmp1>`

## 插槽分类

匿名插槽：它不需要设置 `name` 属性，也叫它单个插槽或者默认插槽。与具名插槽相对，。（它隐藏的 `name` 属性为 `default`。）`<slot>`默认`</slot>`

具名插槽：当一个组件中需要定义多个插槽时，就需要用到具名插槽。有 `name`

需要在 `slot` 标签中添加 `name` 属性，属性值任意写；

在引用组件时，通过添加 `slot = “属性值”` 来关联对应的插槽。

作用域插槽

## 一个 slot 区分，

当组件渲染的时候，`<slot></slot>` 将会被替换为 `“Your Profile”`。

```
<cmp1> Your Profile</cmp1> <slot>!!!</slot>
```

父组件不提供任何插槽内容时 默认显示 Submit, 不论 slot 有没有 name, 只要 cmp 组件是空就取默认值

```
<cmp1></cmp1> <slot>Submit</slot>
```

显示默认值情况：

```
<cmp1>aaa</cmp1> //找不找 name 为 title 的就默认
<cmp1></cmp1>
<cmp1 #title></cmp1>
<slot name="title">默认值</slot>
```

## slot 写法，

语法：a:12

父:slot=" 名 “ 子: <slot name="名"></slot>

```
父级:<cmp1>12</cmp1> //a:12 结果: 12 默认组件传值就用<slot>不加
name
子级写法:<slot></slot>
父级: <cmp1><div slot="名字">abc</div></cmp1>
子级写法:<slot name="名字"></slot> //结果: abc
父级: <cmp1><template v-slot:名字>abc1111</template></cmp1> 同
于 <cmp1><template #名字>abc</template></cmp1>
<cmp1 #user><template>abc</template></cmp1>
子级写法:<slot name="名字"></slot> //结果: abc1111
父级: <cmp1 #title>abc</cmp1> //同于: <cmp1 v-slot:title>abc</cmp1>
子级写法:<slot name="title"></slot> //结果: abc
父 <cmp1 title="用户标题"></cmp1> //结果: 子组件用户标题
子 <slot name="title">{{title}}</slot> props:['title'], //直接写
{{title}} props:['title'], 可不写 slot 这里已实践
```

```
父 <cmp1>abc</cmp1>           //父没定义插槽取 slot 默认值，结果：默认值
子 <slot name="title">默认值</slot>

父 <cmp1 #title>123</cmp1>       //父定义 slot 取组件值，结果：123
子 <slot name="title">默认值</slot>
```

slot 插槽作用域

什么：可以给插槽 template 里传参，也就从里边往外边传参数子组件 slot——>父组件 template

```
父: <template slot-scope="scope">{{scope}}</template> //同
子 <div slot-scope="scope">{{scope}}</div>
子: <slot a="12" :b="55"></slot>           //加:是变量，不加是 String
结果:{ "a": "12", "b": 55 }
```

## 12、Router

(1) 配路由在 router/index.js

```
import Vue from 'vue'
import Router from 'vue-router'
Vue.use(Router);

import index from '@components/index';
import cmp1 from '@components/cmp1';
let router = new Router({
  mode:'hash', //默认有#
  routes:[
    {path:'/', component:index},
    {path:'/cmp1', component:cmp1},
  ]
})
export default router;
```

最外层 index.js 把配置的路由引进来

```
import router from './router';
```

APP.vue

```
<button @click="fn1">首页</button> 函数里内容 this.$router.push('/')
<button @click="fn2">新闻</button> 函数里内容 this.$router.push('/cmp1')
<router-view/>
```

同于

```
<a href="#/">首页</a>
<a href="#/news">新闻</a>
<router-view/>
```

同于

```
<router-link to="/">首页</router-link>
<router-link to="/cmp1">新闻</router-link>
<router-view/>
```

(2)<router-link></router-link>精确匹配选中样式.router-link-exact-active {}

(3)路由三种模式（默认 hash）

访问首页 / 与 /cmp1

1、history——地址变页面不刷新

history 下必须加上如下，否则在 <http://localhost:3002/cmp1> 下 f5 刷新会真像服务器请求，服务器没有 get 就是 404 报错，加上 true 可正常

```
devServer:{
  historyApiFallback:true,
}
```

没#号 例: <http://localhost:3002/>      <http://localhost:3002/cmp1>

2、hash——页面不刷新

有#号 例: <http://localhost:3002/#/>      <http://localhost:3002/#/cmp1>

——像描点，默认模式如: <http://localhost:8080/#/news>

3、abstract——不存在地址栏、不改地址也不改 hash，一般受于后台给前端

不存在有没有#号，只要是当前链接就能切换，如：保留输入的地址切换地址不在变化 如 <http://localhost:3002>，一直这个下切换

(4) 处理 404 访问不存在页面

{path: '\*' } path 中\*通吃，通配 找不到的页面都走\*的组件，代替 404，router/index.js

```
let router = new Router({
  routes:[
    {path:'/',component:index}, {path:'/cmp1',component:cmp1},
    {path: '*',component: notfound} //找不到 cmp3 显示 notfound 东西
  ]
})
```

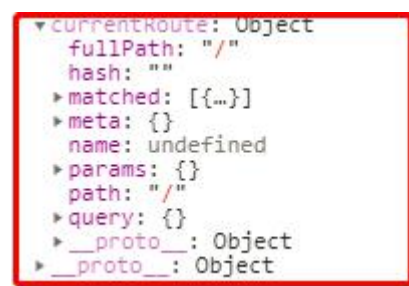
App.vue

这里的找不到页面指路由配置中没引入 cmp3 没加: {path:'/cmp3',component:cmp3}

App.vue 中用了就会空白就表示 404 如<router-link to="/cmp3">新闻</router-link>

等同关系:

```
this.$router.currentRoute == this.$route //内是路由相关参数
$router == new Router({})    this.$router.push('/')
```



```
▼ currentRoute: Object
  fullPath: "/"
  hash: ""
  matched: [...]
  meta: {}
  name: undefined
  params: {}
  path: "/"
  query: {}
  __proto__: Object
```

#### (4) 路由传参—params 与 query

##### 1、Params 方式

如: <http://localhost:3001/news/1323/yjui> //对应写法:id/:name

获参 params 二者同:

```
this.$route.params == s.$router.currentRoute.params
```

```
▶ {id: "1323", name: "223"} "this.$route.params"
```

```
▶ {id: "1323", name: "223"} "this.$router.currentRoute.params"
```

传参: {path: '/news/:id/:name', component: news}, //index.js

传参: <router-link to="/news/1323/yjui">新闻</router-link> //news.vue

取参与赋值

```
let {id, name} = this.$route.params/this.$router.currentRoute.params
```

```
console.log(id, name) //1323 yjui
```

##### 2、query 方式—a=12&b=5

url 格式: <http://localhost:3001/news/1323/yjui?a=12&b=5> //这种写法都是字符串

格式:

```
:to="{path:'', query:{}}" 等同于 this.$router.push({path:'', query:{}})
```

网址都为: <http://localhost:3002/#/cmp1/3123/yjui?a=33&b=55>

```
<router-link :to="{path: '/news/3123/news', query: {a:33, b:55}}">新闻</router-link>
```

等价于

```
this.$router.push({path: '/news/3123/news', query: {a:33, b:55}}) // $router->new Router()
```

取值与获参 query:

```
this.$router.currentRoute.query == this.$route.query
```

#### (5) 组件相同参数不同获取当前参数问题\_用 watch

App.vue

改成这样结果也一样:

```
<router-link :to="{path: '/news/1323/news'}">首页</router-link>
```

```
<router-link :to="{path: '/news/123/hot'}">hot</router-link>
```

```
<template>
```

```
<div id="app">
```

```
<router-link to="/news/1323/news">首页</router-link>
```

```
<router-link to="/news/123/hot">hot</router-link>
```

```
<router-view />
```

```
</div>
```

```
</template>
```

```
<script>
```

```
export default {
```

```
  created() {
```

```
    let { id, name } = this.$router.currentRoute.params;
```

```
    console.log(id, name, this.$route.params);
```

```
  },
```

```
};</script>
```

这种形式只能取到第一次初始加载的参数不能实时触发改变:

```
created() {
  let {id,name} = this.$router.currentRoute.params
  console.log([id,name])
}
```

Router/index.js

```
import Vue from 'vue'
import Router from 'vue-router'
Vue.use(Router);
import cmp1 from '@/components/cmp1';
export default new Router({
  mode: 'history',
  routes: [
    { path: '/news/:id/:name', component: cmp1 },
  ]
});
```

总结组件相同参数不同实时监听的三种写法:

(1) 直接把 created 改成 `updated() {}`

(2) 身上某个东西变了能得到通知 watch, 这里的数据是路由参数数据为\$route 进行监听

随着 <http://localhost:3002/news/1323/news> 与 <http://localhost:3002/news/123/hot> 的切换它显示:

```
123 hot ▶ {id: "123", name: "hot"}
1323 news ▶ {id: "1323", name: "news"}
```

```
updated() {
  let {id,name} = this.$router.currentRoute.params
  console.log([id,name])
},//或
created() {
  let updateData=()=>{
    let {id,name} = this.$router.currentRoute.params
    console.log([id,name])// ["11", "news"] ["22", "hot"]
  }
  updateData();
  this.$watch('$route', updateData)
},//或
export default {
  watch: {
    $route() {
      let { id, name } = this.$router.currentRoute.params;
      console.log(id, name, this.$route.params);
    },
  },
};
```

附加：组件什么时候会更新/重新渲染（updated）：总结就是 data 变了，自己的或是父级

1、data 变了

2、props 变了会更新也就是父级参数变了

3、强制更新——几乎用不上

## (6) Router-path 两种模式与嵌套

(1) 绝对路径——path 写法有绝对路径/。推荐绝对路径 默认 mode:hash

访问时 path 加在/#/后面

访问：一层：<http://localhost:3002/#/index> 二层：<http://localhost:3002/#/aaa>

分别结果：App CMP1 AppCMP1CMP2

```
{path: '/index', component: cmp1, children: [
  {path: '/aaa', component: cmp2} //绝对路径直接找
]}
```

(2) 相对路径

访问：二层：<http://localhost:3002/#/index/aaa>

```
{path: '/index', component: cmp1, children: [
  {path: 'aaa', component: cmp2} //相对路径逐级找
]}
```

嵌套主要用到：

(1) path 相对或绝对路径访问与 children

(2) <router-view/>

```
App.vue:<div id="app">App<router-view/></div>
cmp1: <div>CMP1<router-view/></div>
```

## (7) 命名路由

作用：路由配置是 vue 使用的基础，采用传统方式麻烦且不清晰，而命名路由无论 path 多长多繁琐，都能直接通过 name 就匹配到了，十分方便，所以，强烈推荐使用命名路由的方式

访问：<http://localhost:3001/#/index/12?a=1&b=2>

Router/index.js

```
{path: '/index/:id', name: 'index', component: cmp1},
```

App.vue

```
<router-link :to="{path: '/index/12', query: {a:1, b:2}}">首页</router-link>
```

等同于

```
<router-link :to="{name: 'index', params: {id:12}, query: {a:1, b:2}}">首页</router-link>
```

注：path 中路径要与 to 中 path 一致如：

正确写法

```
routes:[{path: '/index/:id'}]一致于: to="{path: '/index/12'}"
```

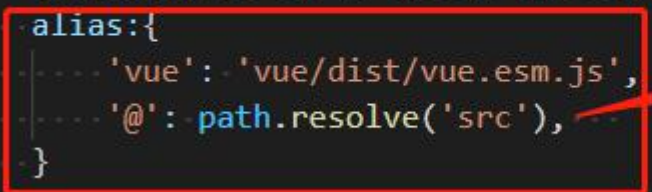
错误写法

```
routes:[{path: '/index/:id'}]不能: to="{path: '/12'}"
```

```

resolve:{
  extensions:['.js','.jsx','.vue'],
  alias:{
    'vue': 'vue/dist/vue.esm.js',
    '@': path.resolve('src'),
  }
},

```

 @指向src

#### (8)Vue-Router 路由钩子函数(导航守卫)

守卫作用：过来新的地址要先经过守卫同意了才能进去 也就是 next()

分类：全局和局部

路由钩子函数有三种：

- 1: 全局钩子： beforeEach、afterEach
- 2: 单个路由里面的钩子： beforeEnter、beforeLeave
- 3: 局部组件路由： beforeRouteEnter、beforeRouteUpdate、beforeRouteLeave

首页可以控制导航跳转，beforeEach，afterEach 等，一般用于页面 title 的修改。一些需要登录才能调整页面的重定向功能。

**beforeEach** 主要有 3 个参数 to, from, next:

**to:** route 即将进入的目标路由对象，

**from:** route 当前导航正要离开的路由

**next:** function 一定要调用该方法 resolve 这个钩子。执行效果依赖 next 方法的调用参数。可以控制网页的跳转。

语法：全局钩子函数： 加在 router/index.js 上

//进入路由前触发

全局前置守卫【先函数在跳页】： router.beforeEach((to, from, next)=>{})

//进入路由后触发

全局后置钩子【先跳页在执行】： router.afterEach((to, from)=>{})

全局形式如：

//to 去哪

//from 目前在哪

//next 如果同意了进入链接页就调用 next() 作用证明，加 next() 可跳到对应页，不加随便点哪 url 都不变没反应

```
var router = new Router({});
```

```
router.beforeEach((to, from, next)=>{}); //router.afterEach((to, from)=>{})
```

```
export default router;
```

局部钩子函数：放组件中

//加 next() 后才跳页。在渲染组件的对应路由被 confirm 前调用。不能访问组件 this

进入这个路由改 Url: beforeRouteEnter(to, from, next) {}

//加 next() 后才跳页。当前路由即同样的 Foo 组件，组件被复用时调用。能访问组件 this。需要路由参数用 watch 获取路由地址变化的东西，还可以用 beforeRouteUpdate 也改变内容与链接跳转

当前路由改变改 Url: beforeRouteUpdate(to, from, next) {}

//加 next() 后才跳页。离开该组件的对应路由时调用。能访问组件 this

离开该组件路由改 Url : beforeRouteLeave(to, from, next) {}

### 全局钩子函数与局部区别

全局写法 `router.xxx` 挂在 `router` 实例上；全局写在 `router/index.js` 中。除了 `afterEach` 都有 `next()`、局部不需挂 `router`，局部写在组件中路由

Ref

作用：ref 除了可以获取本页面的 dom 元素，还可以拿到子组件中的 data 和去调用子组件中的方法

父子组件事件监听写法两种

方法一：\$emit 写法

子：

```
<button @click="$emit('add', 12, 5)">add</button>
```

父：

```
<cmpl @add="fn"/> methods: {fn(a,b){console.log(a, b);//12 5},},
```

方法二：ref 写法\$on 接收，ref 就是给这个组件起个名字

子：

```
<button @click="$emit('add', 12, 5)">add</button>
```

父：

\$on 用在 mounted 生命周期为加载完，created 是 dom 未挂载不可

```
<cmpl ref="cmpl" />
mounted() {this.$refs.cmpl.$on('add', function(a,b) {console.log(a,b);//12 5})}
```

这里证明 箭头函数可以合保证 this 不变

\$once 注意如父组件数据实现 `x++`, `data() {return x=0;}` 要指定 this

//这里 this 组件是 cmpl, 这里点击是还没返应 this 不是 App, 惹祸的 `function() {}`

实现 `x++`: `this.$refs.cmpl.$once('add', function(a,b) {this.x++})`

//这里 this 组件是 App, this 指针是 App 的

可实现 `x++`: `this.$refs.cmpl.$once('add', (a,b)=>{this.x++})`

总结：

父组件中用@add 接收和在 js 中用\$on 效果是一样的

有两种 props 是不一样的，如下：

给类传参用 `propsData` 【new Blue】

组件接收参数用 `props`

### 15、聊聊你对 Vue.js 的 template 编译的理解？

答：简而言之，就是先转化成 AST 树，再得到的 `render()` 函数进行服务器端渲染返回 VNode（Vue 的虚拟 DOM 节点）

详情步骤：

首先，通过编译器把 template 编译成 AST 语法树（abstract syntax tree 即 源代码的抽象语法结构的树状表现形式），`compile` 是 `createCompiler` 的返回值，`createCompiler` 是用以创建编译器的。另外 `compile` 还负责合并 `option`。

然后，AST 会经过 `generate`（将 AST 语法树转化成 `render function` 字符串的过程）得到 `render` 函数，`render` 的返回值是 VNode，VNode 是 Vue 的虚拟 DOM 节点，里面有（标签名、子节点、文本等等）



## HTML 与 CSS

- 1、英文单词不发生词内换行
- 2、文字超出 ...
- 3、隐藏元素的方法有哪些
- 4、谈谈你对浏览器兼容性问题的理解
- 5、简述 JavaScript 中创建自定义对象的方式
- 6、简要描述 JavaScript 中定义函数的几种方式
- 7、split join
- 8、快速排序
- 9、json 和 jsonp 的区别

## 1、英文单词不发生词内换行

```
word-break: break-word;
```

## 2、文字超出 ...

```
word-break: break-word;
overflow: hidden;
text-overflow: ellipsis;
```

两行:

```
overflow: hidden;
text-overflow: ellipsis;
display: -webkit-box;
-webkit-line-clamp: 2;
-webkit-box-orient: vertical;
```

## 3、隐藏元素的方法有哪些

**Opacity:0, Visibility:hidden Display:none**

```
position: absolute; top: -9999px; left: -9999px;
```

## 4、谈谈你对浏览器兼容性问题的理解

```
background-color: #f1ee18; /*所有识别*/
.background-color: #00deff\9; /*IE6、7、8 识别*/
+background-color: #a200ff; /*IE6、7 识别*/
_background-color: #1e0bd1; /*IE6 识别*/
```

属性分为固有属性 property 和自定义属性 attribute

## 5、简述 JavaScript 中创建自定义对象的方式

参考答案: `if(e.stopPropagation){e.stopPropagation();}else{e.cancelBubble = true;}`

自定义对象（user-defined object）指由用户创建的对象，兼容性问题需要由编写者注意。创建自定义对象的方式有：

1、对象直接量: `var person = {name: "rose"};`

2、创建 Object 实例: `var person = new Object(); person.name = "rose";`

3、构造函数

```
function Person(name) {
  this.name = name;
}
var p = new Person('rose'); //{name: 'rose'}
```

4、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\_\_

```
const person = { name: 11 };
const me = Object.create(person);
console.log(me); //{name: 11}
```

## 6、简要描述 JavaScript 中定义函数的几种方式

参考答案:

JavaScript 中，有三种定义函数的方式：

1、函数语句：即使用 `function` 关键字显式定义函数。如：`function f(x){return x+1;}`

2、函数定义表达式：也称为“函数直接量”。形如：`var f = function(x){return x+1;};`

3、使用 `Function()` 构造函数定义，形如：`Var f = new Function("x","return x+1;");`

## 7、split join

split 把字符串分割成数组，

```
var a = 'XHTML?css?JavaScript'
console.log(a.split("?")); //["XHTML", "css", "JavaScript"]
```

join 把数组转成字符串用 join() 里的字符隔开

```
var a = ["XHTML", "CSS", "JavaScript"]
console.log(a.join("#")); //XHTML#css#JavaScript
```

## 8、快速排序

定义：

1. 先从数列中取出一个数作为基准数。
2. 分区过程，将比这个数大的数全放到它的右边 right，小于或等于它的数全放到它的左边 left。
3. 再对左右区间重复第二步，直到各区间只有一个数。

先从后向前找，再从前向后找。交换位置

```
function quickSort(arr) {
    if (arr.length < 2) {
        return arr
    }

    var left = [],
        right = [],
        mid = arr.splice(Math.floor(arr.length / 2), 1);

    for (var i = 0; i < arr.length; i++) {
        if (arr[i] < mid) {
            left.push(arr[i]);
        } else {
            right.push(arr[i])
        }
    }

    return quickSort(left).concat(mid, quickSort(right))
}

document.getElementById('app').innerHTML = quickSort([6, 1, 2, 4, 3, 5])
console.log(quickSort([6, 1, 2, 4, 3, 5]))
```

## 冒泡排序

6, 1, 2, 4, 3, 5->124356

定义： 比较相邻的前后二个数据，如果前面数据大于后面的数据，就将两个数据交换。

这样对数组的第 0 个数据到 N-1 个数据进行一次遍历后，最大的一个数据就“沉”到数组第 N-1 个位置。

N=N-1，如果 N 不为 0 就重复前面二步，否则排序完成。

```
function bubbleSort(arr) {
    for (var i = 0; i < arr.length - 1; i++) {
        for (var j = 0; j < arr.length - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                var temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
    return arr;
}

document.getElementById('app').innerHTML = bubbleSort([6, 1, 2, 4, 3, 5])
// console.log(bubbleSort([6, 1, 2, 4, 3, 5]))
```

## 选择排序

每次改变的结果 612435->162435->126435->123465->123456

比如在一个长度为 N 的无序数组中，在第一趟遍历 N 个数据，找出其中最小的数值与第一个元素交换，第二趟遍历剩下的 N-1 个数据，找出其中最小的数值与第二个元素交换.....第 N-1 趟遍历剩下的 2 个数据，找出其中最小的数值与第 N-1 个元素交换，至此选择排序完成。

```
function selectSort(arr) {
    var min, temp;
    for (var i = 0; i < arr.length - 1; i++) {
        min = i;
        for (var j = i + 1; j < arr.length; j++) {
            if (arr[j] < arr[min]) {
                min = j;
            }
        }
        temp = arr[i];
        arr[i] = arr[min];
        arr[min] = temp;
    }
    return arr;
}

console.log(selectSort([6, 1, 2, 4, 3, 5]))
```

### 9、json 和 jsonp 的区别

JSON 是一种数据交换格式。易于解析和生成,“key/value”键值对的集合 {key:value}

JSONP 是非官方跨域数据交互协议

JSON 是描述信息的格式, JSONP 是信息传递双方约定的方法。

我们拿谍战片来打个比方, JSON 是地下党们用来书写和交换情报的“暗号”, 而 JSONP 则是把用暗号书写的情报传递给自己同志时使用的接头方式。

jsonp 原理简单的说, 就是 json 不支持跨域, 而 js 可以跨域, 因此在服务器端用客户端提供的 js 函数名将 json 数据封装起来, 再将函数提供给客户端调用, 从而获得 json 数据。