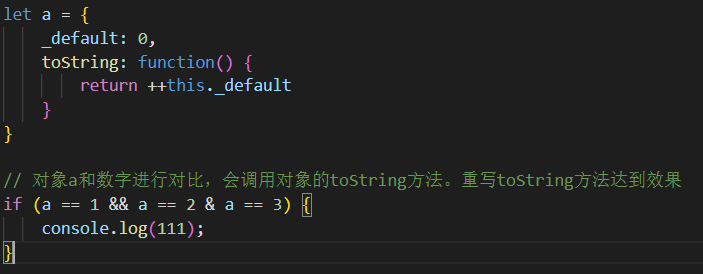
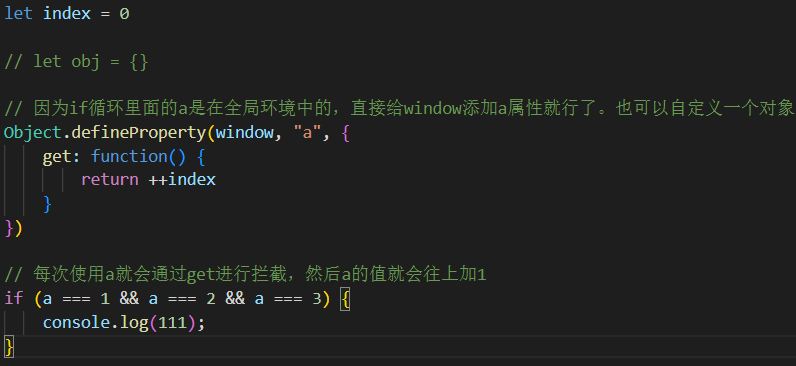
# 面试题目

### 第一题 Object.defineProperty()

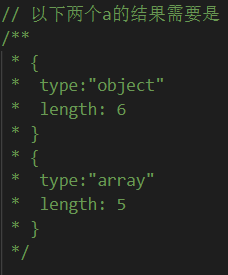
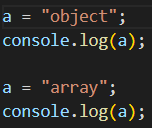
==



===



#### 思维扩散1



#### 思维扩散2



#### 思维扩散3



### 第二题



结果：30

解析：{}进行运算，会隐式转换为”[object Obejct]”



结果：0

解析：数组重写了toString方法，[1，2，3].toString() 🡺 “1,2,3”

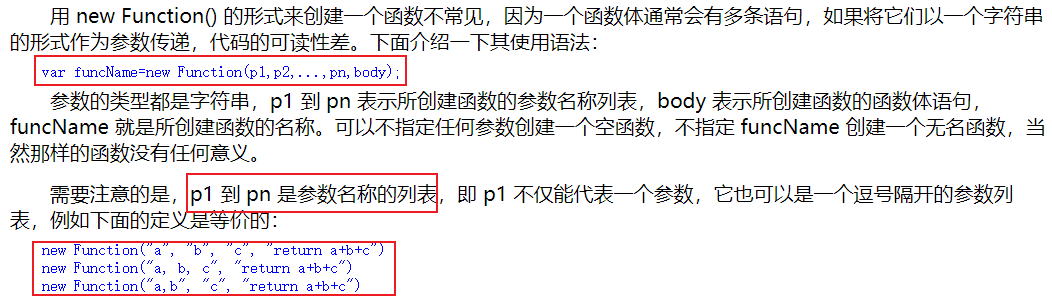
[] = “”



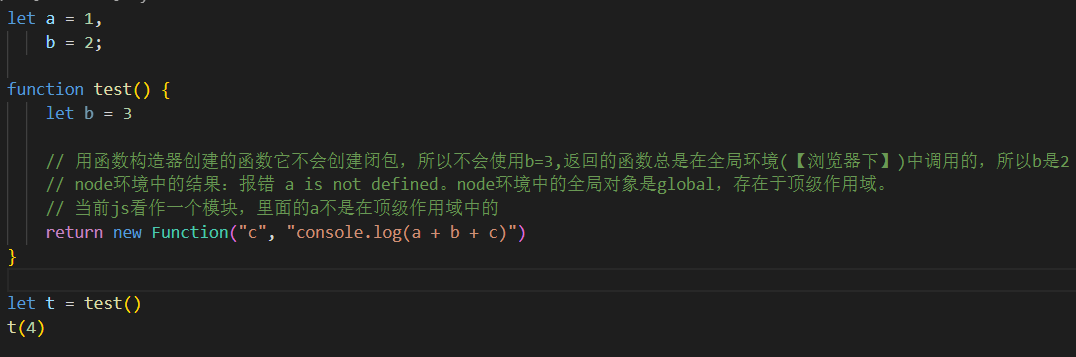
结果：0

解析：代码相当于

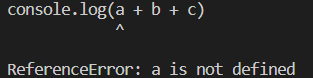
### 第三题：Function()相关



**通过new Function()声明的函数只能访问自己的作用域和全局的作用域**



node环境下的结果：



浏览器下的结果：

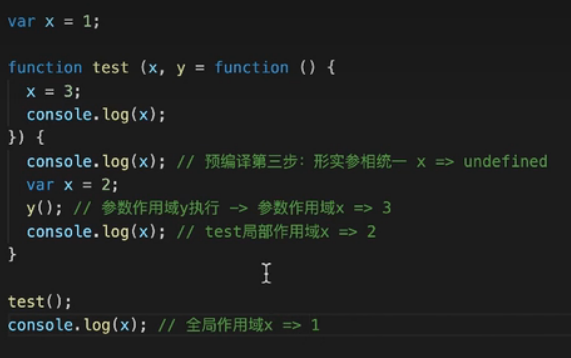


下面这两种方式调用Function是一样的



### 第四题(重点)：

[【全网首发:已完结】一道引发思考的『作用域间的纠葛』笔试题【知识储备】\_哔哩哔哩\_bilibili](https://www.bilibili.com/video/BV1Wq4y167UZ?spm_id_from=333.999.0.0&vd_source=a67e813436aadc4608992466bdaea6bc)

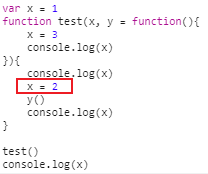


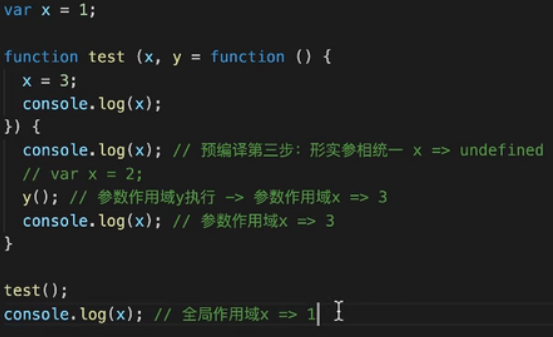
形参位置是有独立空间保存变量的.和函数体内部声明的变量，所在的作用域不同

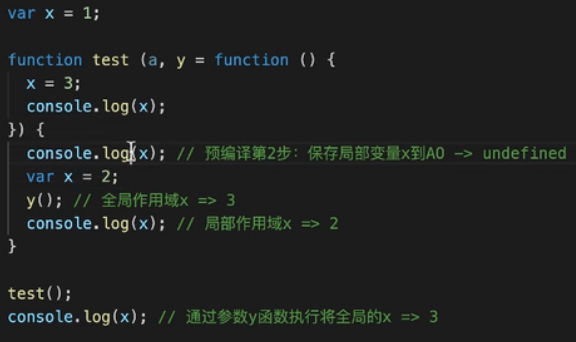
y的函数里面的x就是指向形参x

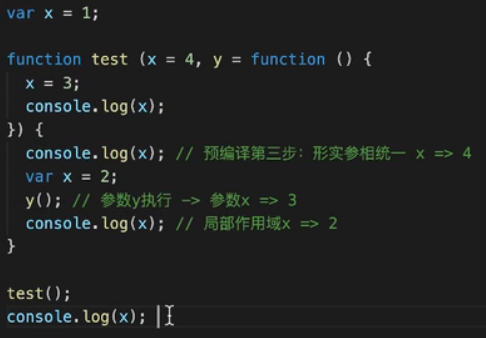
如果函数体里面重新let 了和形参名x一样的变量，那么这个变量x是当前函数体内的局部变量，和形参没有关系。函数体内部获取x变量，是从当前作用域里面获取。

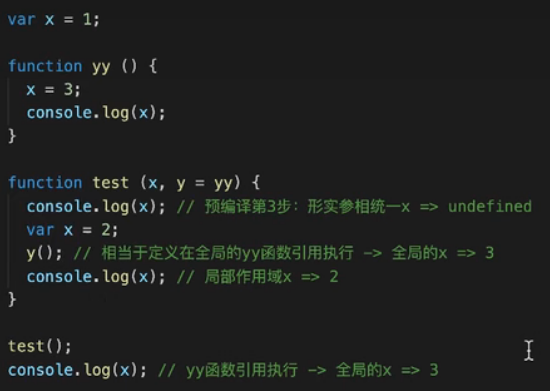
如果没有重新let 那么x就是形参x

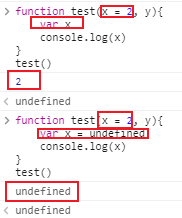
 





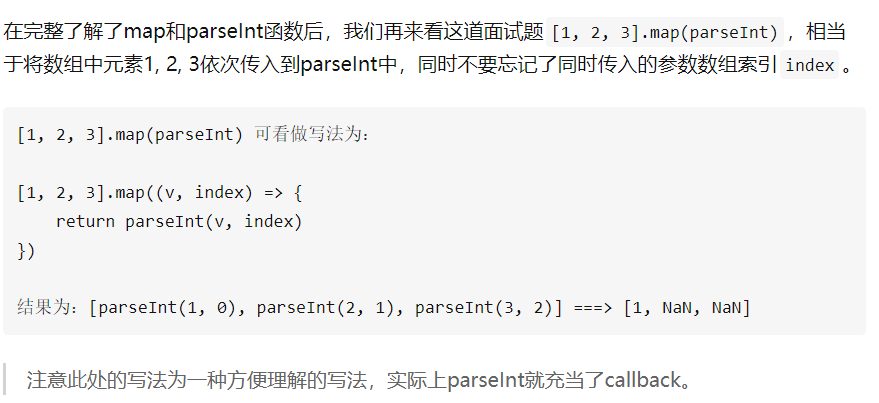




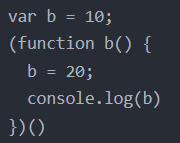


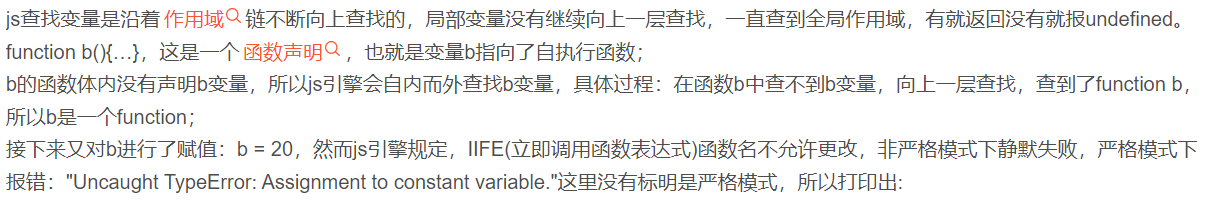
### 第五题

[1, 2, 3].map(parseInt)结果为？

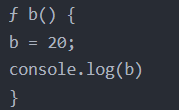


### 第六题

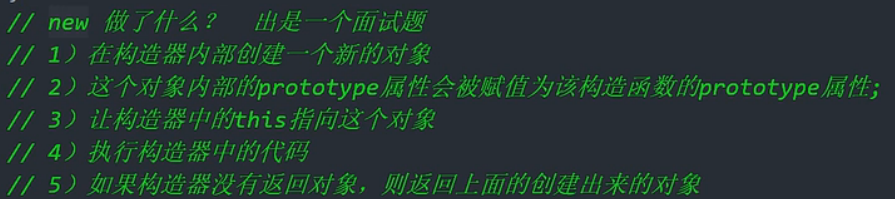


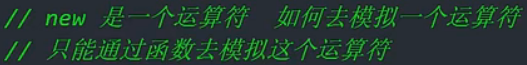


结果是立即执行函数



### new





闭包的概念

一个定义在函数内部的函数，可以读取到其他函数内部变量的函数，本质上，闭包就是一个把函数内部和外部连接起来的桥梁。

简单来说就是指有权访问另外一个函数作用域中的变量的函数（闭包就是能够读取其他函数内部变量的函数）

闭包的作用

函数内部定义的变量属于局部变量，当局部变量所在的函数被调用的时候，就开始执行，当调用执行一旦结束，局部变量就会被释放，当我们需要函数内部变量时，他已经被释放了，读取不到了，这个时候我们就可以用闭包，延长局部变量的执行时间，当函数执行完毕以后，局部变量不可以被内存释放，然后让外部可以访问到这个变量。

闭包的用途

闭包可以用在许多地方。它的最大用处有两个，一个是可以读取函数内部的变量，另一个就是让这些变量的值始终保持在内存中。

闭包主要特点：

1、函数套函数，闭包一定有嵌套函数

2、外层函数一定有局部变量，且内层函数一定操作了外层函数的这个变量

3、内层函数会使用return返回外部（如果不返回这个内层函数，你就没办法使用这个闭包，返回内层函数的最终的目的就是让外部可以访问到这个闭包）

注意

外层函数被多次调用都会创建新的作用域，内层函数操作的外层函数的局部变量之间是不会影响的

闭包的好处

可以重复使用变量，并且不会造成变量污染

全局变量可以重复使用，但是容易造成变量污染。局部变量仅在局部作用域内有效，不可以重复使用，不会造成变量污染。

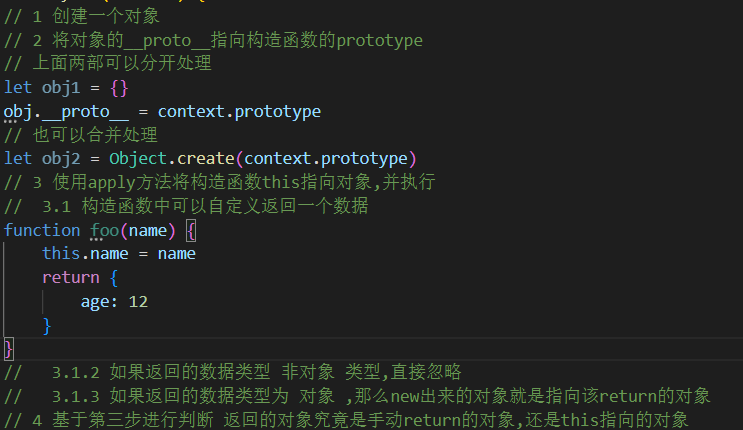
闭包的坏处

1.闭包会使得函数中的变量都被保存在内存中，内存消耗很大，所以不能滥用闭包，可能会导致内存泄露

解决这个问题的办法就是在不使用这些变量时，及时把不需要的局部变量全部删除

2.闭包会在父函数外部，改变父函数内部变量的值。

如果你把父函数当作对象使用，把闭包当作它的公用方法（Public），把内部变量当作它的私有属性（private）时，要小心不要随便改变父函数内部变量的值。



#### 实现代码

function myNew(context) {

    // 1 创建一个对象

    // 2 将对象的\_\_proto\_\_指向构造函数的prototype

    // 3 使用apply方法将构造函数this指向对象,并执行

    //  3.1 构造函数中可以自定义返回一个数据

    //   3.1.2 如果返回的数据类型 非对象 类型,直接忽略

    //   3.1.3 如果返回的数据类型为 对象 ,那么new出来的对象就是指向该return的对象

    // 4 基于第三步进行判断 返回的对象究竟是手动return的对象,还是this指向的对象

    const obj = new Object

    obj.\_\_proto\_\_ = context.prototype

    let res = context.apply(obj, [...arguments].slice(1))

    return typeof res === "object" ? res : obj

}

function Foo(name, age) {

    this.name = name

    this.age = age

    this.fn = function() {

        console.log(this.name);

    }

    // return {

    //     gender: "王五"

    // }

}

let s = myNew(Foo, "zs", 12)

console.log(myNew(Foo, "zs", 12));

s.fn()

### 额外附加题

#### JSON.stringify()深拷贝

##### 弊端

如果 obj 的属性有时间对象，则 JSON.stringify 后再 JSON.parse 的结果，时间将转为字符串的形式，而不是对象的形式

如果 obj 的属性有 RegExp (正则表达式的缩写)、Error 对象，则序列化的结果将只得到空对象；

如果 obj 的属性有函数（function），undefined，Symbol， 则序列化的结果会把 function 或 undefined 丢失；

如果obj 的属性有 NaN、Infinity 和 -Infinity，则序列化的结果会变成 null

JSON.stringify() 只能序列化对象的可枚举的自有属性，例如如果 obj 中的对象是有构造函数生成的， 则使用 JSON.parse(JSON.stringify(obj)) 深拷贝后，会丢弃对象的 constructor；

如果对象中存在循环引用的情况也无法正确实现深拷贝，并且会导致死循环，最后抛出异常

-----------------------------------

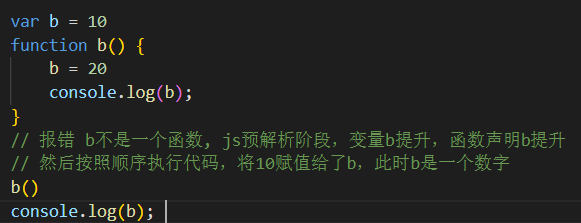
##### 其他常用场景

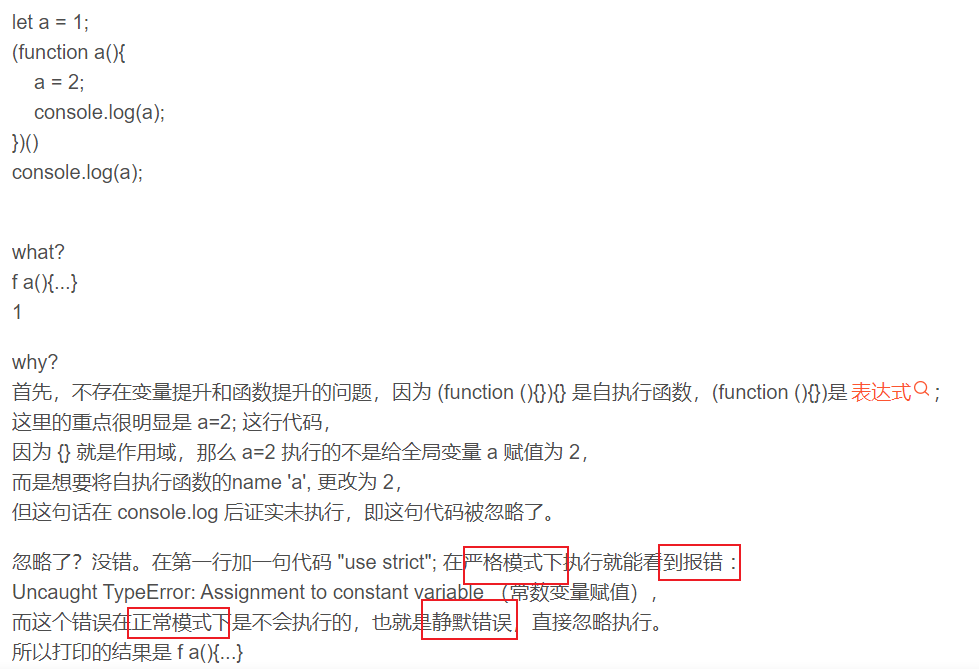
判断数组是否包含某对象： 将数组和要比对的对象转换成字符串后，比对字符串包含关系。（对象顺序很关键）

判断对象是否相等（对象顺序很关键）

让 localStorage / sessionStorage 可以存储对象

### 其他题目1





## 自定义的forEach解决异步问题

let arr = [

  () => console.log("start"),

  () => sleep(1000),

  () => console.log(1),

  () => sleep(2000),

  () => console.log("end"),

];

function sleep(delay) {

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

    setTimeout(() => {

      resolve();

    }, delay);

  });

}

Array.prototype.myForEach = async function (fn) {

  let arr = this,

    arrLen = arr.length,

    \_self = arguments[1];

  for (let i = 0; i < arrLen; i++) {

    await fn.apply(\_self, [arr[i], i, arr]);

  }

};

async function fn(arr) {

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

  //   await arr[i]();

  // }

  arr.myForEach(async (item) => {

    await item();

  });

}

fn(arr);

# css3新增的特性

* [一、选择器](https://blog.csdn.net/weixin_44337386/article/details/124740203#_7)
* [二、新样式](https://blog.csdn.net/weixin_44337386/article/details/124740203#_10)
  + [1、边框](https://blog.csdn.net/weixin_44337386/article/details/124740203#1_11)
  + [2、背景](https://blog.csdn.net/weixin_44337386/article/details/124740203#2_16)
  + [3、文字](https://blog.csdn.net/weixin_44337386/article/details/124740203#3_22)
  + [4、颜色](https://blog.csdn.net/weixin_44337386/article/details/124740203#4_27)
* [三、transition过渡（一般和:hover搭配使用）](https://blog.csdn.net/weixin_44337386/article/details/124740203#transitionhover_31)
* [四、transform转换](https://blog.csdn.net/weixin_44337386/article/details/124740203#transform_56)
* [五、animation动画](https://blog.csdn.net/weixin_44337386/article/details/124740203#animation_110)
* [六、颜色渐变](https://blog.csdn.net/weixin_44337386/article/details/124740203#_120)
* [七、弹性布局+网络布局](https://blog.csdn.net/weixin_44337386/article/details/124740203#_132)

# 清除浮动

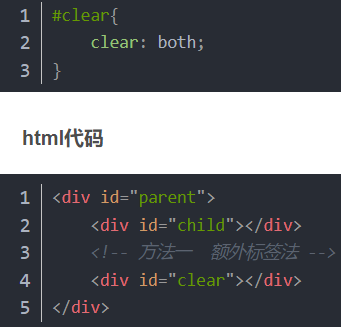
## 方法一：额外标签法

  给谁清除浮动，就在其后额外添加一个空白标签 ，给其设置clear：both。

  优点：通俗易懂，书写方便。

  缺点：添加许多无意义的标签，结构化比较差。

  clear：both：本质就是闭合浮动， 就是让父盒子闭合出口和入口，不让子盒子出来 。



## **方法二：父元素添加overflow:hidden**

通过触发BFC方式，实现清除浮动

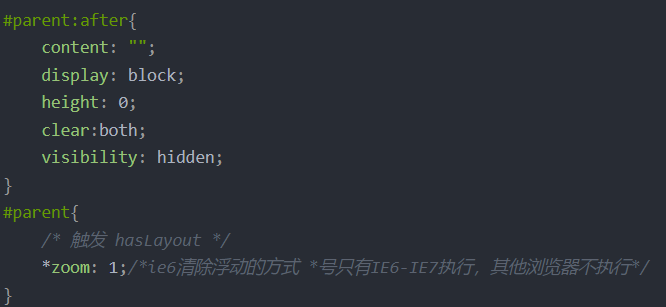
  优点：代码简洁

  缺点：内容增多的时候容易造成不会自动换行导致内容被隐藏掉，无法显示要溢出的元素。

## **方法三：使用after伪元素清除浮动**

  优点：符合闭合浮动思想，结构语义化正确。

  缺点：ie6-7不支持伪元素：after，使用zoom:1触发hasLayout



## **方法四：使用before和after双伪元素清除浮动**

## **方法五：为父元素设置高度**

# window对象和document对象的区别

1、document对象：代表给定浏bai览器窗口中的 HTML 文档。  
2、window对象：表示浏览器中打开的窗口。

window

1.window对象表示浏览器中打开的窗口。

2.window对象可以省略，如: alert() 等价于 window.alert()

document

1.document对象是window对象的一部分。如：

document.body 等价于 window.document.body

2.浏览器的html文档成为document对象。

**DOM 和 BOM 的区别（必会）**

1、BOM

1.1) BOM 是 Browser Object Model 的缩写，即浏览器对象模型。

1.2） BOM 没有相关标准。

1.3） BOM 的最根本对象是 window

2、DOM

2.1） DOM 是 Document Object Model 的缩写，即文档对象模型。

2.2） DOM 是 W3C 的标准。

2.3） DOM 最根本对象是 document（实际上是 window.document）

# 事件三要素（必会）

1、事件源、就是你点的那个 div，触发的对象

2、事件类型、表示动作，比如点击、滑过等

3、事件处理函数（事件处理程序）、表示结果，比如点开关跳转到另一个页面

# 使用 CSS 怎么让 Chrome 支持小于 12px 的文字比如 10px？

针对谷歌浏览器内核，加 webkit 前缀，用 transform:scale()这个属性进行缩放！

<style>

p span{font-size:10px;-webkit-transform:scale(0.8);display:block;}

</style>

<p>

<span>豪豪豪 10px</span>

</p>

# 拖拽释放drag

[(3条消息) 拖拽和释放API\_wochao96的博客-CSDN博客\_拖拽释放api](https://blog.csdn.net/wochao96/article/details/117672712)

[(3条消息) H5中拖拽和释放（drag和drop）\_a浮云的博客-CSDN博客\_h5拖拽释放](https://blog.csdn.net/you_very_brave/article/details/117049342?spm=1001.2101.3001.6650.1&utm_medium=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-1-117049342-blog-117672712.pc_relevant_multi_platform_whitelistv3&depth_1-utm_source=distribute.pc_relevant.none-task-blog-2%7Edefault%7ECTRLIST%7ERate-1-117049342-blog-117672712.pc_relevant_multi_platform_whitelistv3&utm_relevant_index=2)

# 项目问题

1. 为什么需要用vuex和cookie同时管理token？

2. token过期如何处理的？

3. 路由守卫如何做token拦截的？

4. 左侧菜单如何根据路由表动态生成的？

4.1.通过router.option.routes获取到路由信息（默认情况下，后续做了动态路由则通过动态路由表获取）

4.2.sidebar组件遍历获取到的路由表，通过hidde判断是否需要渲染到左侧导航栏中。

4.3.标题和图标则通过路由信息中的meta属性设置。

5. 代理跨域如何操作，原理是什么？

6. 环境变量配置的作用是什么？

7. 路由为何拆模块？如何拆模块？拆完如何用？

因为复杂中台项目的页面众多，不可能把所有的业务都`集中在一个文件上`进行管理和维护， (\*\*可维护性角度\*\*)

\*\*并且还有最重要的，前端的页面中主要分为两部分\*\* (我们的项目很可能带权限设计的!)

1. 一部分是所有人都可以访问的，(`静态路由` - 一直都有的路由) 比如: 登录页, 首页, 404

2. 一部分是只有有权限的人才可以访问的 (`动态路由` - 动态新增或移除的路由) 比如: 薪资管理, 人员管理

\*\*拆分多个模块便于更好的权限的控制 和 维护\*\*

# ES6新特性

[(1条消息) ES6新特性有哪些？\_芮芮白的博客-CSDN博客\_es6新特性](https://blog.csdn.net/m0_48298027/article/details/111242626)

[面试官：ES6中对象新增了哪些扩展? · Issue #36 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/36)

## **新的原始类型和变量声明**

**symbol**

**let和const**

**解构赋值**

**Map和Set**

**新的对象和方法**

**属性简写**

拓展运算符(...)

Object,assign()

Object.is()

**字符串新方法**

includes()

startsWith()/endsWith()，判断字符串是否以参数字符串开头或结尾

padStart()/padEnd()，用参数字符串按给定长度从前面或后面补全字符串，返回新字符串

**数组的新方法**

from()

of()

find()和findIndex()

includes()

**函数**

**参数默认值**

**箭头函数**

**class（类）**

**模块导入和导出**

**Promise**

<https://www.liaoxuefeng.com/wiki/1022910821149312/1023024413276544>

**Generator**

<https://www.liaoxuefeng.com/wiki/1022910821149312/1023024381818112>





# 变量提升与函数提升的区别？

**变量提升**

简单说就是在 JavaScript 代码执行前引擎会先进行预编译，预编译期间会将变量声明与函

数声明提升至其对应作用域的最顶端，函数内声明的变量只会提升至该函数作用域最顶层

当函数内部定义的一个变量与外部相同时，那么函数体内的这个变量就会被上升到最顶端

举例来说：

console.log(a); // undefined

var a = 3;

// 预编译后的代码结构可以看做如下运行顺序

var a; // 将变量 a 的声明提升至最顶端，赋值逻辑不提升。

console.log(a); // undefined

a = 3; // 代码执行到原位置即执行原赋值逻辑

**函数提升**

函数提升只会提升函数声明式写法，函数表达式的写法不存在函数提升

函数提升的优先级大于变量提升的优先级，即函数提升在变量提升之上

函数提升会优先于变量提升,而且不会被同名的变量覆盖但是,如果这个同名变量已经赋值了,那函数变量就会被覆 盖

# 箭头函数的作用(不是特点)

 箭头函数使表达更加简洁，隐式返回值

# js中继承的情况有哪几种

## 第一种：类继承

class Father {

  constructor(name) {

    this.name = name;

  }

}

class Child extends Father {

  constructor(age) {

    super("张三");

    this.age = age;

  }

}

const c = new Child(12);

console.log(c);



## 第二种：原型链继承

特点：1、实例可继承的属性有：

实例的构造函数的属性，父类构造函数属性，父类原型的属性。（新实例不会继承父类实例的属性！

缺点：1、新实例无法向父类构造函数传参。

2、继承单一。

3、所有新实例都会共享父类实例的属性。（原型上的属性是共享的，一个实例修改了原型属 性，另一个实例的原 型属性也会被修改！）

function Father() {

  this.name = "zhangsan";

}

function Child() {

  this.age = 12;

}

Child.prototype = new Father();

Child.prototype.constructor = Child;

const c = new Child();

console.log(c.constructor);

## 第三种：借用构造函数继承

特点：1、只继承了父类构造函数的属性，没有继承父类原型的属性。

2、解决了原型链继承缺点1、2、3。

3、可以继承多个构造函数属性（call多个）。

4、在子实例中可向父实例传参。

缺点：1、只能继承父类构造函数的属性。

2、无法实现构造函数的复用。（每次用每次都要重新调用）

3、每个新实例都有父类构造函数的副本，臃肿。

function Father() {

  this.name = "zhangsan";

}

function Child() {

  Father.call(this);

  this.age = 12;

}

const c = new Child();

console.log(c.name);

## 第四种：组合式继承(原型链继承+借用构造函数继承)

缺点：父构造函数new了两次，性能不是很好

优点：1、可以继承父类原型上的属性，可以传参，可复用。

2、每个新实例引入的构造函数属性是私有的

function Father() {

  this.name = "zhangsan";

  this.eat = function () {

    console.log("吃东西");

  };

}

// 父构造函数的原型方法

Father.prototype.run = function () {

  console.log("跑步");

};

function Child() {

  // 通过借用构造函数继承的方式，继承父构造函数的实例属性

  Father.call(this);

  this.age = 12;

}

Child.prototype = new Father();

Child.prototype.constructor = Child;

const c = new Child();

console.log(c.name); // 父构造函数的name值

c.eat(); // 父构造函数实例方法

c.run(); // 父构造函数原型上的方法

## 第五种：寄生组合继承

重点：

就是给原型式继承外面套了个壳子。

优点：

没有创建自定义类型，因为只是套了个壳子返回对象（这个），这个函数顺理成章就成 了创建的新对象。

缺点：

没用到原型，无法复用。

function Father() {

  this.name = "zhangsan";

  this.eat = function () {

    console.log("吃东西");

  };

}

// 父构造函数的原型方法

Father.prototype.run = function () {

  console.log("跑步");

};

function Child() {

  // 通过借用构造函数继承的方式，继承父构造函数的实例属性

  Father.call(this);

  this.age = 12;

}

// 改动点，就是在组合式继承基础上进行了一些改动,看下方红字部分

// Child.prototype = new Father();

Child.prototype = Object.create(Father.prototype);

Child.prototype.constructor = Child;

const c = new Child();

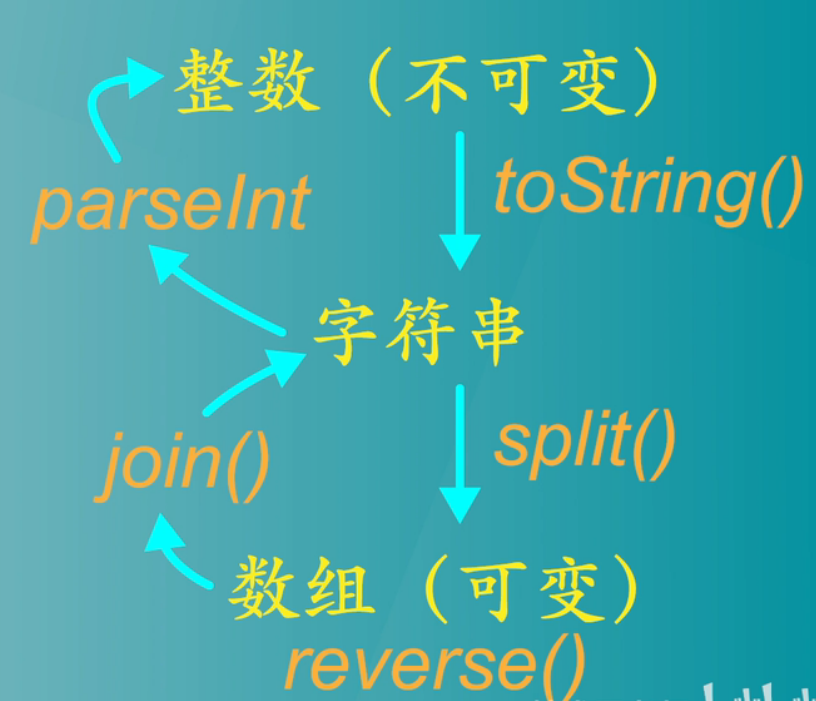
console.log(c.name); // 父构造函数的name值

c.eat(); // 父构造函数实例方法

c.run(); // 父构造函数原型上的方法

console.log(c.constructor);

# 整数反转



# js获取数组最后一位元素的几种方法



# 怎么判断两个对象相等

1. 判断两个对象相等,我们要判断他们两个对象的引用地址是否一致

通过Object.is()同值相等

通过===判断

2.当需求是比较两个对象内容是否一致时就没用了

想要比较两个对象内容是否一致，思路是要遍历对象的所有键名和键值是否都一致：

(1)判断两个对象是否指向一内存

(2)使用Object.getOwnPropertyNames获取对象所有键名数组

(3)判断两个对象的键名数组是否相等

(4)遍历键名,判断键值是否都相等

# get和post请求的区别

1.get请求一般是去取获取数据（其实也可以提交，但常见的是获取数据）；

post请求一般是去提交数据。

2.get因为参数会放在url中，所以隐私性，安全性较差，请求的数据长度是有限制的，

不同的浏览器和服务器不同，一般限制在 2~8K 之间，更加常见的是 1k 以内；

post请求是没有的长度限制，请求数据是放在body中；

3.get请求刷新服务器或者回退没有影响，post请求回退时会重新提交数据请求。

4.get请求可以被缓存，post请求不会被缓存。

5.get请求会被保存在浏览器历史记录当中，post不会。get请求可以被收藏为书签，因为参数就是url中，但post不能。它的参数不在url中。

6.get请求只能进行url编码（appliacation-x-www-form-urlencoded）,post请求支持多种（multipart/form-data等）。

## 深入理解

1…GET 和 POST都是http请求方式， 底层都是 TCP/IP协议；通常GET 产生一个 TCP 数据包；POST 产生两个 TCP 数据包（但firefox是发送一个数据包），

2.对于 GET 方式的请求，浏览器会把 http header 和 data 一并发送出去，服务器响应 200

（返回数据）表示成功；

而对于 POST，浏览器先发送 header，服务器响应 100， 浏览器再继续发送 data，服

务器响应 200 （返回数据）。

# Cache-Control

* 在 Response Headers 中。
* 控制强制缓存的逻辑。
* 例如 Cache-Control: max-age=3153600（单位是秒）

Cache-Control 有哪些值：

* max-age：缓存最大过期时间。
* no-cache：可以在客户端存储资源，每次都必须去服务端做新鲜度校验，来决定从服务端获取新的资源（200）还是使用客户端缓存（304）。
* no-store：永远都不要在客户端存储资源，永远都去原始服务器去获取资源

# 在地址栏里输入一个地址回车会发生哪些事情

1、***解析URL***：首先会对 URL 进行解析，分析所需要使用的传输协议和请求的资源的路径。如果输入的 URL 中的协议或者主机名不合法，将会把地址栏中输入的内容传递给搜索引擎。如果没有问题，浏览器会检查 URL 中是否出现了非法字符，如果存在非法字符，则对非法字符进行转义后再进行下一过程。

2、***缓存判断：***浏览器会判断所请求的资源是否在缓存里，如果请求的资源在缓存里并且没有失效，那么就直接使用，否则向服务器发起新的请求。

 if-Modified-Since  Last-Modified 资源的最后修改时间。

if-none-match Etag 资源的唯一标识（一个字符串，类似于人类的指纹）

3、***DNS解析***： 下一步首先需要获取的是输入的 URL 中的域名的 IP 地址，首先会判断本地是否有该域名的 IP 地址的缓存，如果有则使用，如果没有则向本地 DNS 服务器发起请求。本地 DNS 服务器也会先检查是否存在缓存，如果没有就会先向根域名服务器发起请求，获得负责的顶级域名服务器的地址后，再向顶级域名服务器请求，然后获得负责的权威域名服务器的地址后，再向权威域名服务器发起请求，最终获得域名的 IP 地址后，本地 DNS 服务器再将这个 IP 地址返回给请求的用户。用户向本地 DNS 服务器发起请求属于递归请求，本地 DNS 服务器向各级域名服务器发起请求属于迭代请求。

4、***获取MAC地址：*** 当浏览器得到 IP 地址后，数据传输还需要知道目的主机 MAC 地址，因为应用层下发数据给传输层，TCP 协议会指定源端口号和目的端口号，然后下发给网络层。网络层会将本机地址作为源地址，获取的 IP 地址作为目的地址。然后将下发给数据链路层，数据链路层的发送需要加入通信双方的 MAC 地址，本机的 MAC 地址作为源 MAC 地址，目的 MAC 地址需要分情况处理。通过将 IP 地址与本机的子网掩码相与，可以判断是否与请求主机在同一个子网里，如果在同一个子网里，可以使用 APR 协议获取到目的主机的 MAC 地址，如果不在一个子网里，那么请求应该转发给网关，由它代为转发，此时同样可以通过 ARP 协议来获取网关的 MAC 地址，此时目的主机的 MAC 地址应该为网关的地址。

5、***TCP三次握手***： 下面是 TCP 建立连接的三次握手的过程，首先客户端向服务器发送一个 SYN 连接请求报文段和一个随机序号，服务端接收到请求后向客户端发送一个 SYN ACK报文段，确认连接请求，并且也向客户端发送一个随机序号。客户端接收服务器的确认应答后，进入连接建立的状态，同时向服务器也发送一个ACK 确认报文段，服务器端接收到确认后，也进入连接建立状态，此时双方的连接就建立起来了。

6、***HTTPS握手***： 如果使用的是 HTTPS 协议，在通信前还存在 TLS 的一个四次握手的过程。首先由客户端向服务器端发送使用的协议的版本号、一个随机数和可以使用的加密方法。服务器端收到后，确认加密的方法，也向客户端发送一个随机数和自己的数字证书。客户端收到后，首先检查数字证书是否有效，如果有效，则再生成一个随机数，并使用证书中的公钥对随机数加密，然后发送给服务器端，并且还会提供一个前面所有内容的 hash 值供服务器端检验。服务器端接收后，使用自己的私钥对数据解密，同时向客户端发送一个前面所有内容的 hash 值供客户端检验。这个时候双方都有了三个随机数，按照之前所约定的加密方法，使用这三个随机数生成一把秘钥，以后双方通信前，就使用这个秘钥对数据进行加密后再传输。

7、***返回数据***： 当页面请求发送到服务器端后，服务器端会返回一个 html 文件作为响应，浏览器接收到响应后，开始对 html 文件进行解析，开始页面的渲染过程。

8、***页面渲染***： 浏览器首先会根据 html 文件构建 DOM 树，根据解析到的 css 文件构建 CSSOM 树，如果遇到 script 标签，则判端是否含有 defer 或者 async 属性，要不然 script 的加载和执行会造成页面的渲染的阻塞。当 DOM 树和 CSSOM 树建立好后，根据它们来构建渲染树。渲染树构建好后，会根据渲染树来进行布局。布局完成后，最后使用浏览器的 UI 接口对页面进行绘制。这个时候整个页面就显示出来了。

9、***TCP四次挥手***： 最后一步是 TCP 断开连接的四次挥手过程。若客户端认为数据发送完成，则它需要向服务端发送连接释放请求。服务端收到连接释放请求后，会告诉应用层要释放 TCP 链接。然后会发送 ACK 包，并进入 CLOSE\_WAIT 状态，此时表明客户端到服务端的连接已经释放，不再接收客户端发的数据了。但是因为 TCP 连接是双向的，所以服务端仍旧可以发送数据给客户端。服务端如果此时还有没发完的数据会继续发送，完毕后会向客户端发送连接释放请求，然后服务端便进入 LAST-ACK 状态。客户端收到释放请求后，向服务端发送确认应答，此时客户端进入 TIME-WAIT 状态。该状态会持续 2MSL（最大报文生存时间，指报文段在网络中生存的时间，超时会被抛弃） 时间，若该时间段内没有服务端的重发请求的话，就进入 CLOSED 状态。当服务端收到确认应答后，也便进入 CLOSED 状态。

# 说一下HTTP和HTTPS协议的区别?

1、HTTPS协议需要CA证书,费用较高;而HTTP协议不需要

2、HTTP协议是超文本传输协议,信息是明文传输的,HTTPS则是具有安全性的SSL加密传输协议;

3、使用不同的连接方式,端口也不同,HTTP协议端口是80,HTTPS协议端口是443; 4、HTTP协议连接很简单,是无状态的;HTTPS协议是具有SSL和HTTP协议构建的可进行加密传输、身份认证的网络协议,比HTTP更加安全

# 为什么token要存在vuex中还要存本地一份

用户登录后，访问其他页面需要携带token，vuex是储存在内存里面的，而内存的特点就是快，将token存在vuex中可以提高获取token速度。因为localStorage的读取是一次磁盘读取，读取速度远低于vuex的内存读取，为了避免重复读取localStorage影响性能，需要将localStorage的数据放到vuex里维护。

由于vuex是储存在内存里面的，所以刷新页面就会消失，所以要存本地一份，刷新后token从本地获取。

每次更新token也是修改vuex中的token，然后再覆盖到localstorage中

* 本地存储
  + 获取麻烦
  + 数据不是响应式
* Vuex 容器（推荐）
  + 获取方便
  + 响应式的
* 登录成功，将 Token 存储到 Vuex 容器中
  + 获取方便
  + 响应式
* 为了持久化，还需要把 Token 放到本地存储
  + 持久化

# undefined和null的区别

**相同点**

第一：Undefined和Null两种数据类型都只有一个字面值，分别是undefined和null.

第二：Undefined和Null类型在转换为Boolean类型的值时，都会转换为false.

第三：在需要将两者转换成对象的时候，都会抛出一个TypeError的异常。

第四：Undefined类型派生自Null类型，所以在非严格相等的比较下，两者是相等的。

**不同点**

第一：null是JavaScript的关键字，而undefined是JavaScript的一个全局变量，也就是挂载在window对象上的一个变量，并不是关键字。

第二：在使用typeof运算符进行检测时，Undefined类型的值会返回undefined.而Null类型的值返回为object

第三：在进行数值类型的转换时，undefined会转换为NaN,无法参与计算，而null会转换为0,可以参与计算。

第四：建议：无论在什么情况下都没有必要将一个变量显示的赋值为undefined。如果需要定义某个变量来保存将来要使用的对象，应该将其初始化为null.

# 判断整数的不同方法

/\* 1. 添加一个是数字的判断 \*/  
function isInteger(obj) {  
 return typeof obj === 'number' && obj%1 === 0  
}  
  
/\* 2. 使用Math.round、Math.ceil、Math.floor判断 整数取整后还是等于自己。利用这个特性来判断是否是整数\*/  
function isInteger(obj) {  
 return Math.floor(obj) === obj  
}  
  
/\* 3. 通过parseInt判断 某些场景不准确 \*/  
function isInteger(obj) {  
 return parseInt(obj, 10) === obj  
}  
  
/\* 4. 通过位运算符\*/  
function isInteger(obj) {  
 return (obj | 0) === obj  
}  
  
/\* 5.ES6提供了Number.isInteger \*/

# dom节点平滑滚动到可视区域，顶部，底部

 document.body.scrollIntoView({ behavior: "smooth", block: "start" }) // 顶部  
 document.body.scrollIntoView({ behavior: "smooth", block: "end" }) // 底部  
 document.body.scrollIntoView({ behavior: "smooth"}) // 可视区域

# 各种数组克隆方法]

const clone = (arr) => arr.slice(0);  
const clone = (arr) => [...arr];  
const clone = (arr) => Array.from(arr);  
const clone = (arr) => arr.map((x) => x);  
const clone = (arr) => JSON.parse(JSON.stringify(arr));  
const clone = (arr) => arr.concat([]);  
const clone = (arr) => structuredClone(arr);

# findIndex()与indexOf()的区别

findIndex比[indexOf](https://so.csdn.net/so/search?q=indexOf&spm=1001.2101.3001.7020" \t "_blank)更强大一些，可以通过回调函数查找对象数组，indexOf只能查找数组中指定的值，不过indexOf可以指定开始查找位置的索引。

# last-of-type和last-child的区别

只能找当前父元素下的直接子元素，不包括孙子元素

## last-child

先找到父元素，找到所有的子元素，找到最后一个，判断是不是li，是就是选中，不是就是无效选择器

## last-of-type

先找到父元素，找到所有的类型为li的元素，选中最后一个

# commonjs和ES6模块的差异

[Module 的加载实现 - ES6 教程 - 网道 (wangdoc.com)](https://wangdoc.com/es6/module-loader.html#es6-%E6%A8%A1%E5%9D%97%E4%B8%8E-commonjs-%E6%A8%A1%E5%9D%97%E7%9A%84%E5%B7%AE%E5%BC%82)

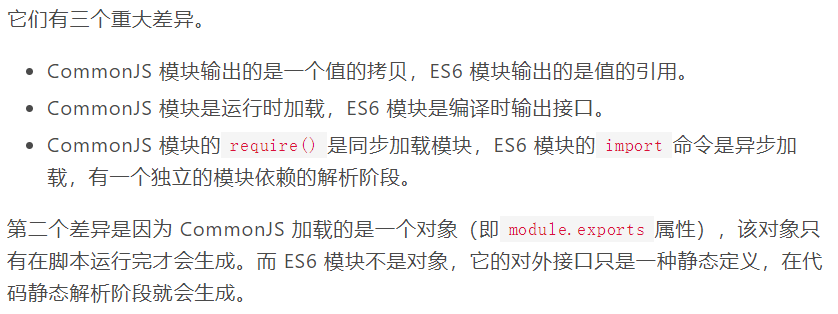
## **CommonJS**

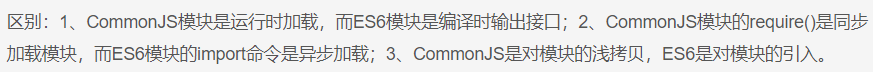
* 对于基本数据类型，属于复制。即会被模块缓存。同时，在另一个模块可以对该模块输出的变量重新赋值。
* 对于复杂数据类型，属于浅拷贝。由于两个模块引用的对象指向同一个内存空间，因此对该模块的值做修改时会影响另一个模块。
* 当使用require命令加载某个模块时，就会运行整个模块的代码。
* 当使用require命令加载同一个模块时，不会再执行该模块，而是取到缓存之中的值。也就是说，CommonJS模块无论加载多少次，都只会在第一次加载时运行一次，以后再加载，就返回第一次运行的结果，除非手动清除系统缓存。
* 循环加载时，属于加载时执行。即脚本代码在require的时候，就会全部执行。一旦出现某个模块被"循环加载"，就只输出已经执行的部分，还未执行的部分不会输出。

## **ES6模块**

* ES6模块中的值属于【动态只读引用】。
* 对于只读来说，即不允许修改引入变量的值，import的变量是只读的，不论是基本数据类型还是复杂数据类型。当模块遇到import命令时，就会生成一个只读引用。等到脚本真正执行时，再根据这个只读引用，到被加载的那个模块里面去取值。
* 对于动态来说，原始值发生变化，import加载的值也会发生变化。不论是基本数据类型还是复杂数据类型。
* 循环加载时，ES6模块是动态引用。只要两个模块之间存在某个引用，代码就能够执行。

## 区别





1.CommonJS 模块是运行时加载，ES6 模块是编译时输出接口。

2.CommonJS 模块的require()是同步加载模块，ES6 模块的import命令是异步加载，有一个独立的模块依赖的解析阶段。

3.CommonJS是对模块的浅拷贝，ES6 Module是对模块的引入，即ES6 Module只存只读，不能改变其值，具体点就是指针指向不能变，类似const 。

4.import的接口是read-only（只读状态），不能修改其变量值。 即不能修改其变量的指针指向，但可以改变变量内部指针指向。可以对commonJS对重新赋值（改变指针指向），但是对ES6 Module赋值会编译报错。

## 共同点

1.CommonJS和ES6 Module都可以对引⼊的对象进⾏赋值，即对对象内部属性的值进行改变。

# export 和export default的区别

首先，它们是在组件复用的过程中使用。

import引用其他组件的值，而export则是暴露自己的成员变量供其他组件调用接收。

在一个组件中，***可以同时使用export和export default。***

1、export default 向外暴露的成员，可以使用任意变量来接收

2、在一个模块中，export default 只允许向外暴露一次

3、在一个模块中，可以同时使用export default 和export 向外暴露成员

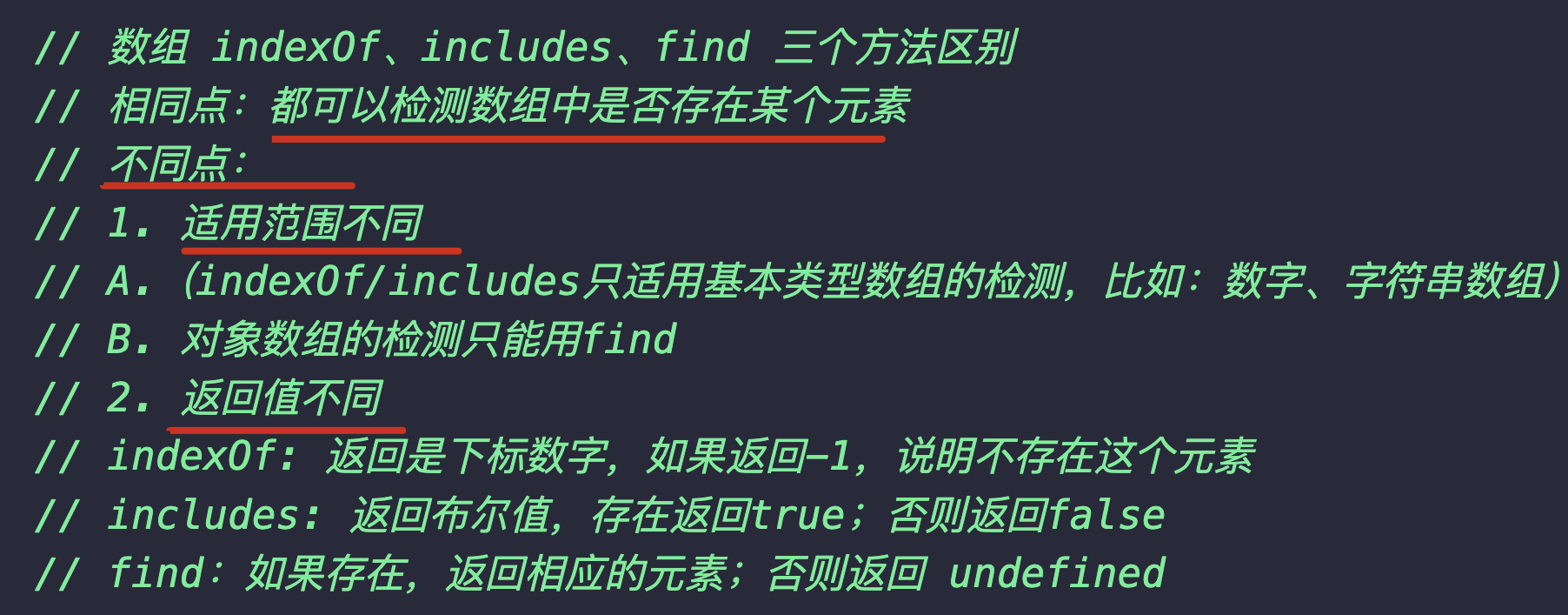
4、使用export向外暴露的成员，只能使用{  }的形式来接收，这种形式，叫做【按需导出】

5、export可以向外暴露多个成员，同时，如果某些成员，在import导入时，不需要，可以不在{ }中定义

6、使用export导出的成员，必须严格按照导出时候的名称，来使用{ }按需接收

7、使用export导出的成员，如果想换个变量名称接收，需要使用as 。

# 数组indexOf,includes,find方法的区别



# 为什么0.1+0.2!=0.3

* 进制转换 ：js 在做数字计算的时候，0.1 和 0.2 都会被转成二进制后无限循环 ，但是 js 采用的 IEEE 754 二进制浮点运算，最大可以存储 53 位有效数字，于是大于 53 位后面的会全部截掉，将导致精度丢失。

# 闭包

## 闭包的概念

一个定义在函数内部的函数，可以读取到其他函数内部变量的函数，本质上，闭包就是一个把函数内部和外部连接起来的桥梁。

简单来说就是指有权访问另外一个函数作用域中的变量的函数（闭包就是能够读取其他函数内部变量的函数）

## 闭包的作用

函数内部定义的变量属于局部变量，当局部变量所在的函数被调用的时候，就开始执行，当调用执行一旦结束，局部变量就会被释放，当我们需要函数内部变量时，他已经被释放了，读取不到了，这个时候我们就可以用闭包，延长局部变量的执行时间，当函数执行完毕以后，局部变量不可以被内存释放，然后让外部可以访问到这个变量。

## 闭包的用途

闭包可以用在许多地方。它的最大用处有两个，一个是可以读取函数内部的变量，另一个就是让这些变量的值始终保持在内存中。

## 闭包主要特点：

1、函数套函数，闭包一定有嵌套函数

2、外层函数一定有局部变量，且内层函数一定操作了外层函数的这个变量

3、内层函数会使用return返回外部（如果不返回这个内层函数，你就没办法使用这个闭包，返回内层函数的最终的目的就是让外部可以访问到这个闭包）

注意

外层函数被多次调用都会创建新的作用域，内层函数操作的外层函数的局部变量之间是不会影响的

## 闭包的好处

可以重复使用变量，并且不会造成变量污染

全局变量可以重复使用，但是容易造成变量污染。局部变量仅在局部作用域内有效，不可以重复使用，不会造成变量污染。

## 闭包的坏处

1.闭包会使得函数中的变量都被保存在内存中，内存消耗很大，所以不能滥用闭包，可能会导致内存泄露

解决这个问题的办法就是在不使用这些变量时，及时把不需要的局部变量全部删除

2.闭包会在父函数外部，改变父函数内部变量的值。

如果你把父函数当作对象使用，把闭包当作它的公用方法（Public），把内部变量当作它的私有属性（private）时，要小心不要随便改变父函数内部变量的值。

# 构造函数和类的区别

1、类的内部定义的所有方法，都是不可枚举的。

2、ES6 的 class 类必须用 new 命令操作，而 ES5 的构造函数不用 new 也可以执行。

3、ES6 的 class 类不存在变量提升，必须先定义 class 之后才能实例化，不像 ES5 中可以将

构造函数写在实例化之后。

4、ES5 的继承，实质是先创造子类的实例对象 this，然后再将父类的方法添加到 this 上面。

ES6 的继承机制完全不同，实质是先将父类实例对象的属性和方法，加到 this 上面（所以

必须先调用 super 方法），然后再用子类的构造函数修改 this。

Class在语法上更贴合面向对象的写法

Class实现继承更加易读、易理解

 class本质还是function.

 类的所有方法都定义在类的prototype属性上

 类创建的实例,里面也有\_\_proto\_\_ 指向类的prototype原型对象

 所以ES6的类它的绝大部分功能，ES5都可以做到，新的class写法只是让对象原型的写法更加清晰、更像面向对象编程的语法而已。

使用的时候，直接对类使用new命令，跟构造函数的用法完全一致，并且类必须用new调用，否则会报错

# 事件委托

事件委托也称之为事件代理（Event Delegation）。是JavaScript中常用绑定事件的常用技巧。顾名思义，“事件代理”即是把原本需要绑定在子元素的响应事件委托给父元素，让父元素担当事件监听的职务。事件代理的原理是DOM元素的事件冒泡。

# 事件捕获和事件冒泡

捕获类型事件只能使用addEventListener注册，传统的注册方法只能是事件冒泡

## 什么是事件冒泡和事件捕获

**事件流**：

页面触发一个事件时，会按照一定的顺序来响应事件，事件的响应过程为事件流

事件流可分为冒泡型事件流、捕获型事件流。

（1）事件冒泡：微软公司提出的，事件由子元素传递到父元素的过程叫做冒泡（false）。

查找事件（事件响应）的顺序：

文本节点–>元素节点—>body—>html—>document（例如点击事件）<向上响应>

（2）捕获事件：网景公司提出的，事件由父元素传递到子元素的过程叫做事件捕获。（ture）

查找事件（事件响应）的顺序：

document–>html–>body–>元素节点–>文本节点 <向下响应>

## 应用

在使用"**事件监听**"的方式绑定事件时，可以设置事件的响应方式。

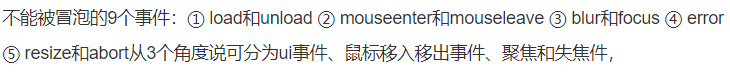
DOM对象.addEventListener(事件，事件处理程序，事件冒泡方式)

**事件冒泡方式：**

（1）默认为false，表示**冒泡阶段**完成事件处理

（2）true为**捕获阶段**完成事件处理

## 不能冒泡的事件



# 状态码响应

## 400

访问网页显示400，其含义是**你访问的页面域名不存在或者请求错误。**

一般是因为我们输入的语法格式有错误，服务器无法理解用户的请求，不知道要表达的是什么。这个时候我们需要认真检查下语义、请求参数是否有误，不然再怎么刷新都没有用。

## 401

**表示服务器收到未经身份验证的请求。**由于您网站上特定网页或目标的凭据无效

## 403

访问网页显示403，就是今天小泽遇到的情况，这是一种常见的错误提示，**表示资源不可用。**服务器理解了客户的请求，但拒绝处理它，通常是由于服务器上文件或目录的权限设置导致的WEB访问错误。

## 405

出现405错误代码表示资源被禁止，对于请求所标识的资源，不允许使用请求行中所指定的方法。**有可能是文件目录权限不够导致的。**

# css工程化概念

让我们更好的去开发和维护，整理我们的css代码，让浏览器读取我们的css代码达到一个高效的解析速度，为我们未来的next语法去做准备，目前市面上实现的就是预处理器，打包工具

# drag拖拽的api

元素在拖放过程中触发的事件

先列下拖拽过程中可能触发的事件，如下：

dragstart：事件主体是被拖放元素，在开始拖放被拖放元素时触发。

darg：事件主体是被拖放元素，在正在拖放被拖放元素时触发。

dragenter：事件主体是目标元素，在被拖放元素进入某元素时触发。

dragover：事件主体是目标元素，在被拖放在某元素内移动时触发。

dragleave：事件主体是目标元素，在被拖放元素移出目标元素是触发。

drop：事件主体是目标元素，在目标元素完全接受被拖放元素时触发。

dragend：事件主体是被拖放元素，在整个拖放操作结束时触发。

其中事件主体是拖放元素的是，dragstart（开始拖动） 、darg（正在拖放） 、dragend（拖放结束），其他4个事件主体都是目标元素，进入、移动、离开、完全进入四个状态。

# transition属性

是一个简写属性，可用于设置四个过渡属性：

transition-property过渡效果的CSS属性的名称（height、width、opacity等）；

transition-duration完成过渡效果需要时间；

transition-timing-function规定速度效果的速度曲线（linear：线性过渡。ease：平滑过渡。ease-in：逐渐加速。ease-out：逐渐减速。ease-in-out：先加速后减速）；

transition-delay过渡效果何时开始（延迟时间）。

# hash和history的区别

1.hash 就是指 url 后面的 # 号以及后面的字符，history没有带#，外观上比hash 模式好看些  
2.原理的区别（原理）

hash:

可以使用 hashchange 事件来监听 hash 值的变化，从而对页面进行跳转（渲染）

history:

HTML5 提供了 History API 来实现 URL 的变化

pushState 和 repalceState 两个 API 来操作实现 URL 的变化 ；

我们可以使用 popstate 事件来监听 url 的变化，从而对页面进行跳转（渲染）；

3. hash 能兼容到IE8， history 只能兼容到 IE10；

connect-history-api-fallback

# 为什么不使用Object.defineProperty对数组做响应式

能监听到数组下标的变化，监听不到长度的变化，然后重写数组的7个方法，能监听到长度的变化，导致监听不到数组的下标的变化

defineProperty的劫持，是需要便利所有的属性；其实defineProperty是有能力检测数组下标变化的；只是vue2是放弃了这个方案，最大的问题就是考虑到数组的长度可能会特别大；性能问题太大；

# **insertAdjacentHTML()**

将文本解析为 **element** 元素，并将结果节点插入到DOM树中的指定位置。它不会重新解析它正在使用的元素，因此它不会破坏元素内的现有元素。这避免了额外的[序列化](https://so.csdn.net/so/search?q=%E5%BA%8F%E5%88%97%E5%8C%96&spm=1001.2101.3001.7020)步骤，使其比直接使用innerHTML操作更快。

element.insertAdjacentHTML(position,text)

position顾名思义，就是想要插入的位置，一共有4个固定的值

'beforebegin'：元素element自己的前面。

'afterbegin'：插入到元素element里面的第一个子节点之前（也就是总是会插入到最前面，例如我插入5个节点，顺序是1、2、3、4、5，那么我就需要以5、4、3、2、1的顺序插入，有一种栈结构先进后出的感觉）。

'beforeend'：插入元素element里面的最后一个子节点之后（这个比较容易理解，就是插入到最后一个节点后，例如我插入5个节点，顺序是1、2、3、4、5，那就正常的1、2、3、4、5就好啦，但是注意是在已有节点的后面哦）。

'afterend'：元素element自己的后面。

# babel的原理

babel的转移过程分为三个阶段，这三个步骤分别是：

1、解析parse:将代码解析生成抽象语法树（AST）,即词法分析和语法分析的过程。

2、转换Transform:对于AST进行变换的一些列的操作，babel接收得到的AST并通过babel-traverse对其进行遍历，在此过程中进行添加，更新以及移除等操作。

3、生成Generate:将变换后的AST再转换为JS代码，使用到的模块是babel-generator。

# Vue的特点





# computed 中的属性名和 data 中的属性名可以相同吗？

答案是：不可以同名

因为不管是 computed 属性名还是 data 数据名还是 props 数据名 都会被挂载在 vm 实例上，因此这三个都不能同名。

# vue.use()的原理

Vue.use方法的第一个参数只能接收对象与函数形式的参数，对象参数需要有install方法，当传入参数中是对象，且有install方法时，会自动执行其中的install方法，**并且install方法中的this依然指向该对象。**

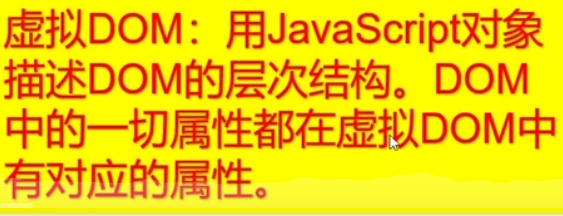
传入该对象的install方法和函数中的第一个参数，必然是Vue对象本身

Vue.use会自动阻止多次注册相同的插件，届时即使多次调用也只会注册一次该插件

## 插件的作用：

插件通常用来为Vue添加全局功能

# 虚拟dom



* 用 JavaScript 对象模拟真实 DOM 树，对真实 DOM 进行抽象；
* diff 算法 — 比较两棵虚拟 DOM 树的差异；
* pach 算法 — 将两个虚拟 DOM 对象的差异应用到真正的 DOM 树。

我们先根据真实DOM生成一颗virtual DOM，当virtual DOM某个节点的数据改变后会生成一个新的Vnode，然后Vnode和oldVnode作对比，发现有不一样的地方就直接修改在真实的DOM上，然后使oldVnode的值为Vnode。

diff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。

**Diff算法是一种对比算法**。对比两者是旧虚拟DOM和新虚拟DOM，对比出是哪个虚拟节点更改了，找出这个虚拟节点，并只更新这个虚拟节点所对应的真实节点，而不用更新其他数据没发生改变的节点，实现精准地更新真实DOM，进而提高效率。

新旧虚拟DOM对比的时候，Diff算法比较只会在同层级进行, 不会跨层级比较。 所以Diff算法是:深度优先算法。 时间复杂度:O(n)

当数据改变时，会触发setter，并且通过Dep.notify去通知所有订阅者Watcher，订阅者们就会调用patch方法，给真实DOM打补丁，更新相应的视图。

# Vue 中如何实现一个虚拟 DOM？说说你的思路（高薪常问）

首先要构建一个 VNode 的类，DOM 元素上的所有属性在 VNode 类实例化出

来的对象上都存在对应的属性。

例如 tag 表示一个元素节点的名称，text 表示一个文本节点的文本，chlidren 表示子节点等。

将 VNode 类实例化出来的对象进行分类，例如注释节点、文本节点、元素节点、组件节点、函数式节点、克隆节点。

然后通过编译将模板转成渲染函数 render，执行渲染函数 render，在其中创建不同类型的 VNode 类，最后整合就可以得到一个虚拟 DOM（vnode）。

最后通过 patch 将 vnode 和 oldVnode 进行比较后，生成真实 DOM。

# key的作用

1、key的作用主要是为了更高效的对比虚拟DOM中的某个节点是否是相同节点。

key 的作用主要是为了高效的更新虚拟 DOM

2、Vue在patch过程中判断两个节点是否是相同节点key是一个必要条件，渲染一组列表时，key往往是唯一标识，所以如果不定义key的话，Vue只能认为比较的两个节点是同一个，哪怕它们实际上不是，这导致了频繁更新元素，使得整个patch过程比较低效，影响性能。

3、实际使用中在渲染一组列表时key必须设置，而且必须是唯一标识，应该避免使用数组索引作为key，这可能导致一些隐蔽的bug；Vue中在使用相同标签元素过渡切换时，也会使用key属性，其目的也是为了让Vue可以区分它们，否则Vue只会替换其内部属性而不会触发过渡效果。

4、从源码中可以知道，Vue判断两个节点是否相同时主要判断两者的key和元素类型等，因此如果不设置，它的值就是是undefined，则可能永远认为这是两个相同节点，只能去做更新操作，这造成了大量的DOM更新操作，明显是不可取的。

# patch

patch过程是一个递归过程，遵循深度优先、同层比较的策略；以vue3的patch为例

* 首先判断两个节点是否为相同同类节点，不同则删除重新创建
* 如果双方都是文本则更新文本内容
* 如果双方都是元素节点则递归更新子元素，同时更新元素属性
* 更新子节点时又分了几种情况：
  + 新的子节点是文本，老的子节点是数组则清空，并设置文本；
  + 新的子节点是文本，老的子节点是文本则直接更新文本；
  + 新的子节点是数组，老的子节点是文本则清空文本，并创建新子节点数组中的子元素；
  + 新的子节点是数组，老的子节点也是数组，那么比较两组子节点，更新细节blabla

vue3中引入的更新策略：编译期优化patchFlags、block等

## Vue 怎么用 vm.$set() 解决对象新增属性不能响应的问题 ？

 如果目标是数组，直接使用数组的 splice 方法触发相应式；

 如果目标是对象，会先判读属性是否存在、对象是否是响应式，最终如果要对属性进行响应式处理，则是通过调用 defineReactive 方法进行响应式处理（ defineReactive 方法就是 Vue 在初始化对象时，给对象属性采用 Object.defineProperty 动态添加 getter 和 setter 的功能所调用的方法）

# scoped 的实现原理

Vue 中的 scoped 属性的效果主要通过 PostCSS 转译实现，

如下是转译前的Vue

代码： <template><div>Vue.js scoped</div></template>

<style scoped>.scoped {font-size:14px;}</style>

浏 览 器 渲 染 后 的 代 码

<div data-v-fed36922>Vue.js scoped</div>.scoped[data-v-fed36922]{font-size:14px;}

即：PostCSS 给所有 dom 添加了一个唯一不重复的动态属性，然后，给 CSS 选择器额外添加一个对应的属性选择器来选择该组件中 dom，这种做法使得样式私有化

# vuex与全局对象的区别

1 Vuex 的状态存储是响应式的。当 Vue 组件从 store 中读取状态的时候，若 store 中的状态发生变化，那么相应的组件也会相应地得到高效更新。

2 你不能直接改变 store 中的状态。改变 store 中的状态的唯一途径就是显式地提交 (commit) mutation。这样使得我们可以方便地跟踪每一个状态的变化，从而让我们能够实现一些工具帮助我们更好地了解我们的应用

[全局变量](https://so.csdn.net/so/search?q=%E5%85%A8%E5%B1%80%E5%8F%98%E9%87%8F&spm=1001.2101.3001.7020)可以任意修改

3 全局变量多了会造成命名污染，vuex不会，同时解决了父组件与孙组件，以及兄弟组件之间通信的问题。

# 说说nextTick的使用和原理

在【下次 DOM 更新】 【循环结束之后】执行延迟回调。

在修改数据之后立即使用这个方法，获取更新后的 DOM。

1. nextTick是等待下一次 DOM 更新刷新的工具方法。
2. Vue有个异步更新策略，意思是如果数据变化，Vue不会立刻更新DOM，而是开启一个队列，把组件更新函数保存在队列中，在同一事件循环中发生的所有数据变更会异步的批量更新。这一策略导致我们对数据的修改不会立刻体现在DOM上，此时如果想要获取更新后的DOM状态，就需要使用nextTick。
3. 开发时，有两个场景我们会用到nextTick：

* created中想要获取DOM时；
* 响应式数据变化后获取DOM更新后的状态，比如希望获取列表更新后的高度。

1. nextTick签名如下：function nextTick(callback?: () => void): Promise<void>

所以我们只需要在传入的回调函数中访问最新DOM状态即可，或者我们可以await nextTick()方法返回的Promise之后做这件事。

1. 在Vue内部，nextTick之所以能够让我们看到DOM更新后的结果，是因为我们传入的callback会被添加到队列刷新函数(flushSchedulerQueue)的后面，这样等队列内部的更新函数都执行完毕，所有DOM操作也就结束了，callback自然能够获取到最新的DOM值。

# MVVM

Model–View–ViewModel （MVVM） 是一个软件架构设计模式，由微软 WPF 和 Silverlight 的架构师 Ken Cooper 和 Ted Peters 开发，是一种简化用户界面的事件驱动编程方式。由 John Gossman（同样也是 WPF 和 Silverlight 的架构师）于2005年在他的博客上发表

MVVM 源自于经典的 Model–View–Controller（MVC）模式 ，MVVM 的出现促进了前端开发与后端业务逻辑的分离，极大地提高了前端开发效率，MVVM 的核心是 ViewModel 层，它就像是一个中转站（value converter），负责转换 Model 中的数据对象来让数据变得更容易管理和使用，该层向上与视图层进行双向数据绑定，向下与 Model 层通过接口请求进行数据交互，起呈上启下作用。如下图所示：

# 指定组件显示的内容的多种方式

new Vue({选项})

1 el 选项，通过一个选择器找到容器，容器内容就是组件内容

2 template 选项，<div>组件内容</div> 作为组件内容

3 render选项，它是一个函数，函数回默认传人createElement的函数（h），这个函数用来创建结构，在render函数返回渲染为组件内容。它的优先级更高。

render 是vue提供的一个渲染函数，优先级大于el,template等选项，用来提供组件结构

# 怎么在组件中监听路由参数的变化？

有两种方法可以监听路由参数的变化，但是只能用在包含<router-view />的组

件内。

**第一种**

watch: {

'$route'(to, from) {

// 在此处监听

},

},

**第二种**

beforeRouteUpdate (to, from, next) {

//这里监听

},

# 什么是 Vue 的计算属性（必会）

在模板中放入太多的逻辑会让模板过重且难以维护，在需要对数据进行复杂

处理，且可能多次使用的情况下，尽量采取计算属性的方式，好处：使得数据处

理结构清晰；

1、 依赖于数据，数据更新，处理结果自动更新；

2、 计算属性内部 this 指向 vm 实例；

3、 在 template 调用时，直接写计算属性名即可；

4、 常用的是 getter 方法，获取数据，也可以使用 set 方法改变数据；

5、 相较于 methods，不管依赖的数据变不变，methods 都会重新计算，但是依赖数据不变的时候

# 什么时候使用插槽

当子组件被复用时，需要在特定的区域展示不同的定制化的内容

# **什么是插槽？**

插槽（Slot）是Vue提出来的一个概念，正如名字一样，插槽用于决定将所携带的内容，插入到指定的某个位置，允许用户自定义组件的模板标签。从而使模板分块，具有模块化的特质和更大的重用性。插槽显不显示、怎样显示是由父组件来控制的，而插槽在哪里显示就由子组件来进行控制

# 组件和插件的区别

两者的区别主要表现在以下几个方面：

* 编写形式
* 注册形式
* 使用场景



组件 (Component) 是用来构成你的 App 的业务模块，它的目标是 App.vue

插件 (Plugin) 是用来增强你的技术栈的功能模块，它的目标是 Vue 本身

简单来说，插件就是指对Vue的功能的增强或补充

# export 和export default的区别

1.export与export default均可用于导出常量、函数、文件、模块等

2.在一个文件或模块中，export、import可以有多个，export default仅有一个

3.通过export方式导出，在导入时要加{ }，export default则不需要，并可以起任意名称

(1) 输出单个值，使用export default

(2) 输出多个值，使用export

(3) export default与普通的export不要同时使用

# 双向数据绑定口头描述

## 为什么要把dom元素当作一个订阅者?

vue要做的事情，就是把数据的变化通知到每个订阅者，每个订阅者其实就是DOM元素，那么Dom元素为什么要订阅数据的变化呢？因为数据变了dom元素要更新自己，但是dom元素并不知道数据什么时候发送变化，所以需要订阅数据变化的消息

当数据改变时，会触发setter，并且通过Dep.notify去通知所有订阅者Watcher，

调用订阅者的update方法,更新视图

# 服务端渲染SSR

## ****传统****服务端渲染SSR ****web开发****

网页内容在服务端渲染完成，⼀次性传输到浏览器

浏览器拿到的是全部的dom结构

## 新版的SSR

[手把手教你打造Vue SSR\_哔哩哔哩\_bilibili](https://www.bilibili.com/video/BV1dE411C7f5/?spm_id_from=333.337.search-card.all.click&vd_source=a67e813436aadc4608992466bdaea6bc)

SSR解决方案，后端渲染出完整的首屏的dom结构返回，前端拿到的内容包括首屏及完整spa结构，应用激活后依然按照spa方式运行

* Vue SSR是一个在SPA上进行改良的服务端渲染
* 通过Vue SSR渲染的页面，需要在客户端激活才能实现交互
* Vue SSR将包含两部分：服务端渲染的首屏，包含交互的SPA

SSR主要解决了以下两种问题：

* seo：搜索引擎优先爬取页面HTML结构，使用ssr时，服务端已经生成了和业务想关联的HTML，有利于seo
* 首屏呈现渲染：用户无需等待页面所有js加载完成就可以看到页面视图（压力来到了服务器，所以需要权衡哪些用服务端渲染，哪些交给客户端）

但是使用SSR同样存在以下的缺点：

* 复杂度：整个项目的复杂度
* 库的支持性，代码兼容
* 性能问题
  + 每个请求都是n个实例的创建，不然会污染，消耗会变得很大
  + 缓存 node serve 、 nginx判断当前用户有没有过期，如果没过期的话就缓存，用刚刚的结果。
  + 降级：监控cpu、内存占用过多，就spa，返回单个的壳
* 服务器负载变大，相对于前后端分离务器只需要提供静态资源来说，服务器负载更大，所以要慎重使用

所以在我们选择是否使用SSR前，我们需要慎重问问自己这些问题：

1. 需要SEO的页面是否只是少数几个，这些是否可以使用预渲染（Prerender SPA Plugin）实现
2. 首屏的请求响应逻辑是否复杂，数据返回是否大量且缓慢

Vue SSR是一个在SPA上进行改良的服务端渲染

通过Vue SSR渲染的页面，需要在客户端激活才能实现交互

Vue SSR将包含两部分：服务端渲染的首屏，包含交互的SPA

**（1）服务端渲染的优点：**

* 更好的 SEO： 因为 SPA 页面的内容是通过 Ajax 获取，而搜索引擎爬取工具并不会等待 Ajax 异步完成后再抓取页面内容，所以在 SPA 中是抓取不到页面通过 Ajax 获取到的内容；而 SSR 是直接由服务端返回已经渲染好的页面（数据已经包含在页面中），所以搜索引擎爬取工具可以抓取渲染好的页面；
* 更快的内容到达时间（首屏加载更快）： SPA 会等待所有 Vue 编译后的 js 文件都下载完成后，才开始进行页面的渲染，文件下载等需要一定的时间等，所以首屏渲染需要一定的时间；SSR 直接由服务端渲染好页面直接返回显示，无需等待下载 js 文件及再去渲染等，所以 SSR 有更快的内容到达时间；

**（2) 服务端渲染的缺点：**

* 更多的开发条件限制： 例如服务端渲染只支持 beforCreate 和 created 两个钩子函数，这会导致一些外部扩展库需要特殊处理，才能在服务端渲染应用程序中运行；并且与可以部署在任何静态文件服务器上的完全静态单页面应用程序 SPA 不同，服务端渲染应用程序，需要处于 Node.js server 运行环境；
* 更多的服务器负载：在 Node.js 中渲染完整的应用程序，显然会比仅仅提供静态文件的 server 更加大量占用CPU 资源 (CPU-intensive - CPU 密集)，因此如果你预料在高流量环境 ( high traffic ) 下使用，请准备相应的服务器负载，并明智地采用缓存策略。

### ****单页应用SPA****

单页应用优秀的用户体验，使其逐渐成为主流，页面内容由JS渲染出来，这种方式称为客户端渲染

浏览器拿到的仅有宿主元素#app(结构)，并没有内容

# 你是怎么处理vue项目中的错误的

主要的错误来源包括：

* 后端接口错误
* 代码中本身逻辑错误

## 后端接口错误

通过axios的interceptor实现网络请求的response先进行一层拦截

apiClient.interceptors.response.use(

response => {

return response;

},

error => {

if (error.response.status == 401) {

router.push({ name: "Login" });

} else {

message.error("出错了");

return Promise.reject(error);

}

}

);

## 代码逻辑问题

### 全局设置错误处理函数

设置全局错误处理函数

Vue.config.errorHandler = function (err, vm, info) {

// handle error

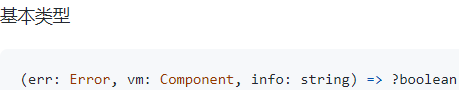
// `info` 是 Vue 特定的错误信息，比如错误所在的生命周期钩子

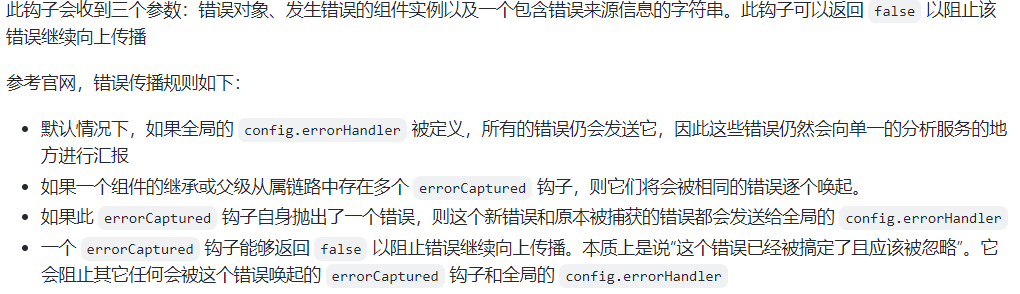
// 只在 2.2.0+ 可用

}

errorHandler指定组件的渲染和观察期间未捕获错误的处理函数。这个处理函数被调用时，可获取错误信息和 Vue 实例

### 生命周期钩子



例子









# Vue3.0 性能提升主要是通过哪几方面体现的

[面试官：Vue3.0 性能提升主要是通过哪几方面体现的？ · Issue #46 · febobo/web-interview (github.com)](https://github.com/febobo/web-interview/issues/46)

## diff算法优化

vue3在diff算法中相比vue2增加了静态标记

关于这个静态标记，其作用是为了会发生变化的地方添加一个flag标记，下次发生变化的时候直接找该地方进行比较，提高性能

## 静态提升

Vue3中对不参与更新的元素，会做静态提升，只会被创建一次，在渲染时直接复用

这样就免去了重复的创建节点，大型应用会受益于这个改动，免去了重复的创建操作，优化了运行时候的内存占用

## 事件监听缓存

默认情况下绑定事件行为会被视为动态绑定，所以每次都会去追踪它的变化

上述发现开启了缓存后，没有了静态标记。也就是说下次diff算法的时候直接使用

## SSR优化

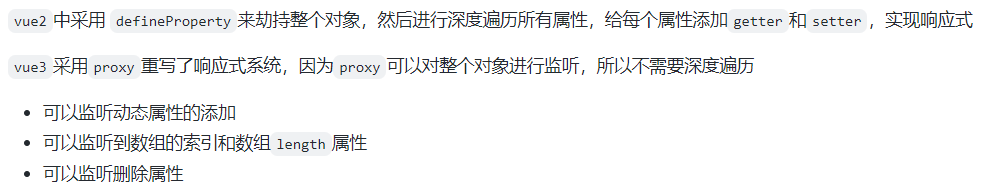
当静态内容大到一定量级时候，会用createStaticVNode方法在客户端去生成一个static node，这些静态node，会被直接innerHtml，就不需要创建对象，然后根据对象渲染

## 源码体积

相比Vue2，Vue3整体体积变小了，除了移出一些不常用的API，再重要的是Tree shanking

任何一个函数，如ref、reavtived、computed等，仅仅在用到的时候才打包，没用到的模块都被摇掉，打包的整体体积变小

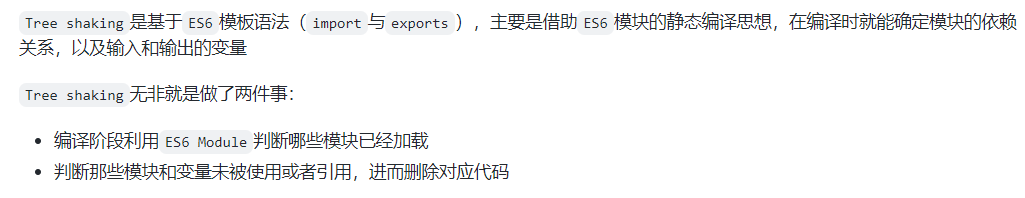
## 响应式系统

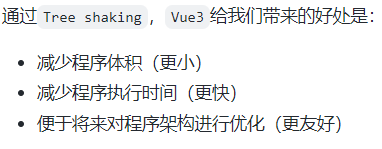


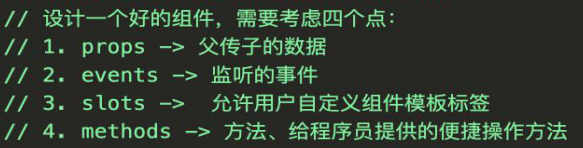
# Tree Sharking

Tree shaking 是一种通过清除多余代码方式来优化项目打包体积的技术，专业术语叫 Dead code elimination

简单来讲，就是在保持代码运行结果不变的前提下，去除无用的代码







# cookie和session的区别

## Cookie:

Cookie：其实就是客户端储存的，什么是客户端，就是浏览器存储，

能看的见的，在浏览器设置-历史纪录中能看见，能手动清除Cookie。

所以它一般都会被用在不重要的地方，因为它很容易被发现（cookie以明文储存信息），

而且储存量很小(单个cookie保存的数据不能超过4K)，

它的有效期也不咋地，一般你清除一下浏览器就没了，正常20分钟后，

cookie生命周期结束.

## session:

Session:服务端，也是放在服务器上(session的默认失效时间为30分钟),

Session是另一种记录客户状态的机制，基于Cookie实现，

不同的是Cookie保存在客户端浏览器中，而Session保存在服务器上

客户端浏览器访问服务器的时候，

服务器把客户端信息以某种形式记录在服务器上，

这就是Session，

客户端浏览器再次访问时只需要从Session

中查找该客户的状态就可以了

## 它们之间的区别：

1、cookie数据存放在客户的浏览器上，session数据放在服务器上。

2、cookie不是很安全，别人可以从本地的cookie获取你的信息，

考虑到安全性应当使用session。

3、session会在一定时间内保存在服务器上。当访问增多，

会比较占用你服务器的性能，考虑到减轻服务器性能方面，应当使用cookie。

4、个站点单个cookie保存的数据不能超过4K，

很多浏览器都限制一最多保存20个cookie。

5、可以考虑将登陆信息等重要信息存放为session，

其他信息如果需要保留，可以放在cookie中。

6、cookie的生命周期是累计的，从创建时，

就开始计时，20分钟后，cookie生命周期结束，

7、关机会造成session生命周期的结束，

但是对cookie没有影响

8、访问范围：

session为一个用户浏览器独享

cookie为多个用户浏览器共享

# [css计算属性](https://www.cnblogs.com/kaicy/p/14706442.html)

比如说，当我一个元素需要位移，比如我需要居中，但是当父元素没有相对定位，它又不能定位居中，设置margin-left:50%也会出现本身宽度的一半的偏差，还得在偏回来，麻烦。

所以这里给大家介绍一个计算属性：

calc(百分比 - 像素) // 百分比减掉像素值  
// 需要居中的盒子  
.box{  
 width:500px;  
 height:400px;  
 margin-left:calc(50% - 250px);  
 margin-top:calc(50% - 200px);  
}  
友情提示：注意加空格百分比和像素和减号之间

# localStorage和sessionStorage对比

## 生命周期

localStorage:localStorage的生命周期是永久的，关闭页面或浏览器之后localStorage中的数据也不会消失。localStorage除非主动删除数据，否则数据永远不会消失。

sessionStorage的生命周期是在仅在当前会话下有效。sessionStorage引入了一个“浏览器窗口”的概念，sessionStorage是在同源的窗口中始终存在的数据。只要这个浏览器窗口没有关闭，即使刷新页面或者进入同源另一个页面，数据依然存在。但是sessionStorage在关闭了浏览器窗口后就会被销毁。同时独立的打开同一个窗口同一个页面，sessionStorage也是不一样的。

## 存储大小：

localStorage和sessionStorage的存储数据大小一般都是：5MB

## 存储位置：

localStorage和sessionStorage都保存在客户端，不与服务器进行交互通信。

## 存储内容类型：

localStorage和sessionStorage只能存储字符串类型，对于复杂的对象可以使用ECMAScript提供的JSON对象的stringify和parse来处理

## 获取方式：

localStorage：window.localStorage;；sessionStorage：window.sessionStorage;。

## 应用场景：

localStoragese：常用于长期登录（+判断用户是否已登录），适合长期保存在本地的数据。sessionStorage：敏感账号一次性登录；

# vuex actions配置选项



# vuex getters配置选项



# for和foreach和map

## 性能：

for > forEach > map

for 循环当然是最简单的，因为它没有任何额外的函数调用栈和上下文

forEach 其次，因为它其实比我们想象得要复杂一些。它不是普通的 for 循环的语法糖，还有诸多参数和上下文需要在执行的时候考虑进来，这里可能拖慢性能；

map 最慢，因为它的返回值是一个等长的全新的数组，数组创建和赋值产生的性能开销很大

## 为什么map和forEach性能没有for好，要在ES5时设计出这两个方法

* 在不确定长度或者计算长度有损性能时可以使用
* 作用域保存数据的问题

# ts的好处

1.ts是js的超级，给js增加了可选的静态类型和基于类的面向对象的编程

2.ts在编译时会有错误提示，js在运行时才能暴露错误

3.ts是强类型语言，强制定义各种数据的类型，可读性强

* ts是js的超集，存在类型的脚本语言
* 继承了js的所有编程类型，js代码可在ts环境很好的运行
* 为构建大型应用而生，但小程序同样适用
* 强大的类型系统，拥有静态类型检查能力
* 新增类型注解和类型推断
* 拥有丰富的class扩展功能
* 添加了系统级设计能力，设计模式由顶层由下进行设计

# http 1.1 和 http 2.0 的区别

[(4条消息) HTTP 2.0 与HTTP1.1的差别\_crazy的蓝色梦想的博客-CSDN博客\_http2.0和http1.1](https://blog.csdn.net/qq_41257129/article/details/103432915)

## (1) 多路复用

http 2.0 可以在一个 tcp/ip 连接上并发的发送多个 http 请求，不必等待上一个 http 请求返回请求报文。虽然 http 1.1 可以启动多个 tcp/ip 连接来达到并发，但是 tcp/ip 连接本身的建立也会产生额外的性能开销。

http1.1串联等待前一个

http2.0并发

## (2) 头部压缩

http 报文分为三个部分，状态行，头部，主体。http 1.1 会对主体进行压缩，但不会压缩头部，http 2.0 使用 hpack 算法对 header 进行压缩。

## (3) 服务 器推送

比方说，一个资源服务器上有 html，css，js，http 1.1 会在浏览器解析 html 的时候再次发送请求，请求 css 和 js 资源。但是 http 2.0 会在浏览器第一次请求 html 的时候，检查这个 html 引用的资源，如果被引用的资源刚好也在这台服务器上，那么服务器会自动地把被引用的资源，比如，css，js 推送给浏览器。

http1.1先请求html->请求css+请求js

http2.0请求html返回html+css+js

## http2.0的优点

### 多路复用（multiplexing)：

通过单一的TCP连接，可以无限制处理多个HTTP请求。所以请求的处理都在一条TCP连接上完成，因此TCP的处理效率得到提高。解决了（head of line blocking）的问题。

### 请求优先级（request priorityzation):

SPDY不仅可以无限制的并发处理请求，还可以给请求逐个分配优先级顺序。这样重要的请求就会优先响应。

### header压缩：

HTTP1.1的头部是冗余的，SPDY协议会采用合适的压缩算法压缩HTTP请求和响应的首部。这样，通信产生的数据包数量和发送的字节数就更少了。

### 推送功能（server push）：

SPDY协议支持服务器主动向客户端推送数据的功能。

到底推送了什么？比如客户端请求了一个index.html文件,服务器会将客户端所需要的资源伴随index.html文件一起发送给客户端，不必等待客户端请求。

## HTTP1.1的升级

### 默认长连接。

HTTP1.0也提供长连接，但是默认是短连接，要想长连接必须将头部的connection设置为keep-alive。而HTTP1.1默认开启connection：keep-alive，不用设置。

强制设置Host请求首部字段。Host字段是HTTP1.1唯一一个强制要求设置的字段。

### 管线化（pipelining）：

不用等待响应，就可以发送下一个请求。

### 缓存处理的扩展：

HTTP1.1增加的Cache-control可以控制缓存行为，并且支持断点续传。

### 带宽优化：

HTTP1.0中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了 ，并且不支持断点续传。HTTP1.1增加了range请求头部字段，允许客户端只请求服务器某个对象的一部分资源。返回206状态码表示请求一部分资源成功，否则返回200的状态码，并且返回对象的全部资源。

## HTTP1.1的瓶颈

HTTP1.1有很多优点，但是其标准会使其成为发展的瓶颈。

### 队头阻塞（Head of line blocking）：

虽然HTTP1.1支持管线化，允许一次发起多个请求，但是服务器会按照请求的顺序依次响应。一旦响应某个请求出现了阻塞，那么后面的请求即使已经处理完毕了，仍然需要等待阻塞的请求处理完毕。

### 多个TCP连接:

虽然http1.1支持管道化，但是很多浏览器根本不支持它。（因为它真的很鸡肋）。HTTP1.1的请求并发还是基于多个TCP连接，但建立tcp连接的本较高，比较耗时。

### 头部信息冗余：

每一次请求都会带上一些相同的首部。头部数据都是以纯文本的形式发送的，通常会给每个请求增加500~800字节的负荷。

请求只能从客户端开始。客户端不能接受除响之外的指令。

# axios原理

[(4条消息) axios原理及使用\_大大艺术家的博客-CSDN博客\_axios原理](https://blog.csdn.net/Joy_Huu/article/details/109523557)

[(4条消息) axios原理\_xx小黄人的博客-CSDN博客](https://blog.csdn.net/VegetableKCCCC/article/details/120127261)

## axios是什么？

axios是一个基于promise的http库，可以用在浏览器和node.js的环境中；本质上也是对原生xhr的封装，只不过它是promise的实现版本，符合最新的ES规范

## axios有什么特性

1.可以用在浏览器和node.js的环境中，从浏览器中创建XMLHttpRequests；基于node内置核心模块http实现axios，从中创建http请求，也就是说，axios可以在浏览器上和服务器上都可以发起请求

2.是一个基于Promise的HTTP库，支持promise的所有API

源码中返回的是一个promise对象

3.拦截请求和响应

4.可以转换请求数据和响应数据，并对响应的内容自动转换为json类型的数据

5.安全性更高，客户端支持防御XSRF

首先axios会判断当前环境是否为标准的浏览器环境，如果是标准的浏览器环境的话，就会继续执行。对于XSRF攻击来说，最基本的就是跨域了，所以我们对发出请求的域和当前的域做同源判断，如果是跨域的话，就必须有凭证config.withCredentials || isURLSameOrigin(fullPath)，凭证简单来说就是当前请求在跨域时是否可以带上cookie。

在满足上面条件时，即是同源或者跨域但是有凭证这两种情况，同源的我们自然不必再处理，而对于跨域的我们接下去看，当config中有xsrfCookieName时，将config.xsrfCookieName的内容使用cookies.read处理后返回给xsrfValue

## axios较其他网络请求库的优点

1.简单易用，api接近于jquery，比原生的fetch之类简单

2.浏览器兼容性好，都能兼容IE7，使用fetch的话就得自己处理兼容

3.通用性好，能在node和浏览器中使用，api一致

4.稳定大牌，vue官网文档中有推荐

## 基本原理

1.axios还是属于xhr，因此需要实现一个ajax或基于http；

2.需要结合promise对象对结果进行处理

## 缺点

1.不支持jsonp,需要自己封装

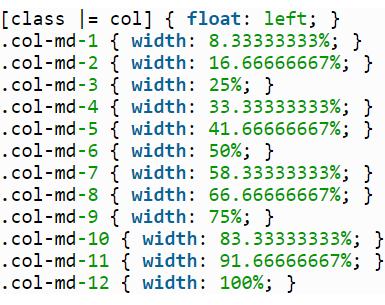
2.基于xhr实现，所以无法在service worker,web worker中使用

# 栅格布局的原理

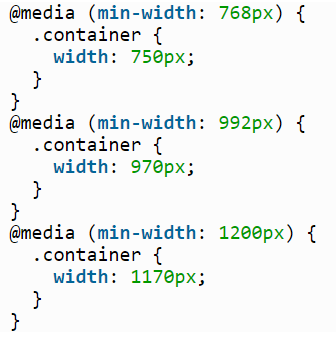
[浅谈CSS 栅格系统布局原理分析\_CSS教程\_CSS\_网页制作\_脚本之家 (jb51.net)](https://www.jb51.net/css/585500.html)

众所周知，现在前端有很多类似于bootstrap，foundation这样优秀的UI框架，它们都提供了自己的一套响应式布局方案，即栅格系统。用过的人都知道只要给页面的元素添加其栅格系统指定的类名，就能达到你想达到的响应式布局效果，简洁而优雅。

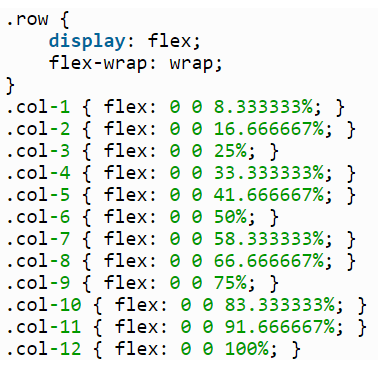
bootstrap@3.x版本为了兼容IE8，采用的是浮动方式来实现栅格系统



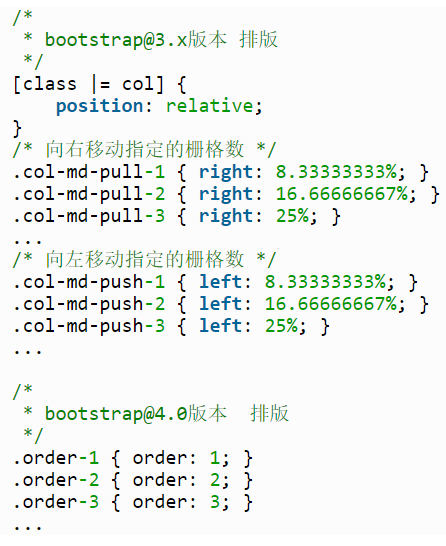
即每行的一个栅格都是用左浮动和百分比来进行排版，当窗口宽度改变，对应改变container容器的宽度，对应栅格宽度自然也跟着改变：



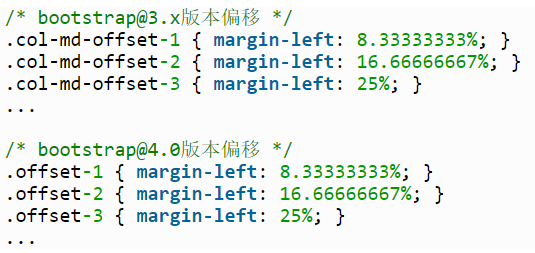
bootstrap@4.0版本放弃了对版本IE的支持，栅格系统采用的是最新的伸缩布局方式



栅格系统可以决定每一个栅格的排版顺序，两种栅格系统实现方式不同，自然他们得排版方式也不同：



可以看到，@3.0版本采用的是相对定位进行左右移动栅格来进行排版，@4.0版本就很简洁，只用采用flex布局特有的order属性来进行排版。当然，这两个版本也有相同的排版方式，就是offset偏移：



# 浅谈vue几个缺点

1.不利于SEO

SEO是Search Engine Optimization的缩写，意思是“搜索引擎优化”，搜索引擎的基础爬虫的原理就是抓取你的url，然后获取你的html源代码并解析。而vue里面的大部分数据是存在于js代码里面的，也就是说，如果一个单页应用，html在服务器端还没有渲染出所有数据，在浏览器才渲染出所有数据，而搜索引擎请求到的html是没有渲染数据的。 这样就很不利于内容被搜索引擎搜索到。

2.首页白屏

主要原因还是因为他是SPA的特性，会在首页的时候将所有的js、css数据全部加载，当项目过于庞大时，这个白屏时间就会比较明显。

详细介绍及解决方案

3.不支持IE8及以下的浏览器

vue的底层原理使用了Object.defineProperty，这是es5的特性，但是IE8及以下几乎不支持es5，并且Vue需要Promise的支持，IE8同样不支持Promsie。

# 微任务和宏任务

微任务和宏任务是异步任务的两个种类。  
宏任务:当前调用栈中执行的代码成为宏任务。(主代码块，定时器等等)。  
微任务:当前(此次事件循环中)宏任务执行完，在下一个宏任务开始之前需要执行的任务,可以理解为回调事件。(promise.then,proness.nextTick等等)。  
宏任务中的事件放在callback queue中，由事件触发线程维护;微任务的事件放在微任务队列中，由js引擎线程维护。在挂起任务时，JS引擎会将所有任务按照类别分到这两个队伍中，首先在macrotask的队列中取出第一个任务，执行完毕后取出microtask队列中的所有任务顺序执行;之后再取macrotask任务，周而复始，直至两个队列的任务都取完

# Promise的常见应用场景有哪些

[Promise的常见应用场景有哪些 - web开发 - 亿速云 (yisu.com)](https://www.yisu.com/zixun/145491.html)

## **Promise.all**

### **应用场景1：多个请求结果合并在一起**

具体描述：一个页面，有多个请求，我们需求所有的请求都返回数据后再一起处理渲染

思考：如果并发请求的话，每个请求的loading状态要单独设置，多个的话可能多个loading 重合，页面显示的内容 根据请求返回数据的快慢 有所差异，具体表现在渲染的过程，为提升用户体验，我们可以采用 所有请求返回数据后，再一起渲染，此时我们关闭请求的单独loading设置，通过Promise.all 汇总请求结果，从开始到结束，我们只设置一个 loading 即可。

### **应用场景2：合并请求结果并处理错误**

描述：我们需求单独处理一个请求的数据渲染和错误处理逻辑，有多个请求，我们就需要在多个地方写

思考：我们能否把多个请求合并在一起，哪怕有的请求失败了，也返回给我们，我们只需要在一个地方处理这些数据和错误的逻辑即可。

### **应用场景3：验证多个请求结果是否都是满足条件**

描述：在一个微信小程序项目中，做一个表单的输入内容安全验证，调用的是云函数写的方法，表单有多7个字段需要验证，都是调用的一个 内容安全校验接口，全部验证通过则 可以 进行正常的提交

## **Promise.race**

### **应用场景1：图片请求超时**

### **应用场景2：请求超时提示**

描述：有些时候，我们前一秒刷着新闻，下一秒进入电梯后，手机页面上就会提示你 “网络不佳”

## **Promise.prototype.then**

### **应用场景1：下个请求依赖上个请求的结果**

描述：类似微信小程序的登录，首先需要 执行微信小程序的 登录 wx.login 返回了code，然后调用后端写的登录接口，传入 code ，然后返回 token ，然后每次的请求都必须携带 token，即下一次的请求依赖上一次请求返回的数据

### **应用场景2：中间件功能使用**

描述：接口返回的数据量比较大，在一个then 里面处理 显得臃肿，多个渲染数据分别给个then，让其各司其职

# 请说说你对标签语义化的理解？

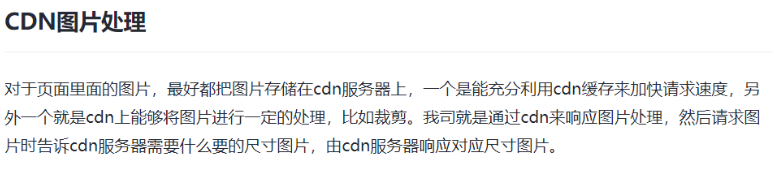
1 一个语义元素能够清楚的描述其意义给浏览器和开发者，即使在去掉或丢失样式的时候，也能够让页面呈现出清晰的结构；

2 有利于SEO优化，让页面和搜索引擎建立良好的沟通，爬虫依赖于标签来确定上下文和各个关键词的权重，有助于爬虫抓取更多的有效信息；

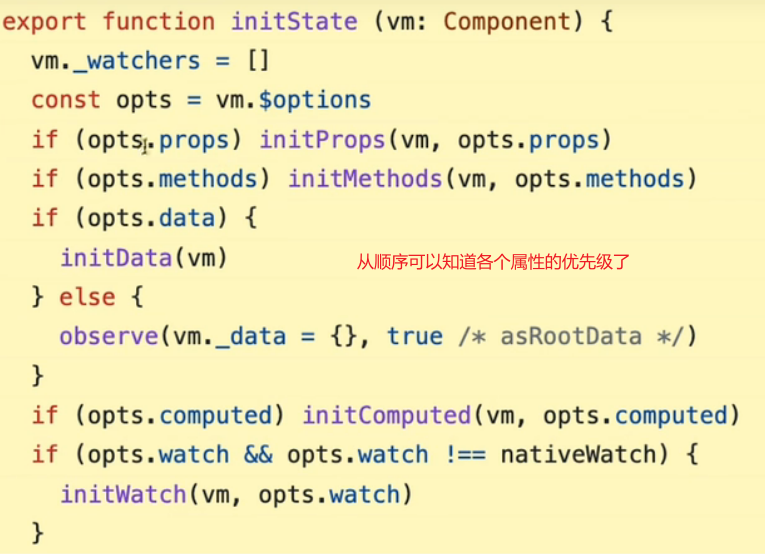
3 方便其他设备解析（如屏幕阅读器、盲人阅读器、移动设备等），并以具有意义的方式来渲染网页；

4 便于团队开发和维护，语义化更具有可读性，遵循W3C标准的团队都遵循语义化标准，可以减少差异化；

# CDN图片处理



# options api的优先级问题



# 组件封装流程

## 1 封装Message组件的流程

1导入Message组件和所需的render，createVNode函数

2 创建一个方法

2.1 createVNode：根据组件创建虚拟节点，并传入组件所需要的自定义属性

2.2 createElement准备一个DOM容器

2.3 render：把虚拟节点渲染到容器中

2.4 开启定时器，到时间后移出DOM容器内容

// 实现使用函数调用xtx-message组件的逻辑

import { createVNode, render } from 'vue'

import XtxMessage from './XTXMessage/xtx-message.vue'

// 准备dom容器 在函数的外面定义这个容器,只在初始化时创建一次,如果定义在函数中,函数每次被调用都要创建一次

const div = document.createElement('div')

div.setAttribute('class', 'xtx-message-container')

document.body.appendChild(div)

// 定时器标识

let timer = null

export default ({ type, text }) => {

  // 实现：根据xtx-message.vue渲染消息提示

  // 1. 导入组件

  // 2. 根据组件创建虚拟节点 createVNode是vue自带的方法

  const vnode = createVNode(XtxMessage, { type, text })

  // 3. 准备一个DOM容器

  // 4. 把虚拟节点渲染DOM容器中

  render(vnode, div)

  // 5. 开启定时，移出DOM容器内容

  clearTimeout(timer)

  timer = setTimeout(() => {

    render(null, div)

  }, 3000)

}

## 2 封装comfirm的信息

大致思路和Message组件是一样的

但是comfirm需要点击确定和取消按钮，而不是过了几秒后会自动关闭

所以需要使用promise封装

import { createVNode, render } from 'vue'

import XtxConfirm from './XTXComfirm/xtx-confirm'

// 3.准备div容器

const div = document.createElement('div')

div.setAttribute('class', 'xtx-confirm-container')

document.body.appendChild(div)

// 1 该函数渲染XtxConfirm组件，标题和文本

//    函数的返回值是promise对象

export default ({ title, text }) => {

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

    const submitCallback = () => {

      render(null, div) // 销毁组件

      resolve()

    }

    const cancelCallback = () => {

      render(null, div) // 销毁组件

      reject(new Error('点击取消'))

    }

    // 2.根据组件创建虚拟节点

    //    点击确认按钮，触发resolve同时销毁组件

    //    点击取消按钮，触发reject同时销毁组件

    const vnode = createVNode(XtxConfirm, { title, text, submitCallback, cancelCallback })

    // 4.将虚拟节点添加到DOM容器中

    render(vnode, div)

  })

}

<!-- 下面逻辑后面再看，估计会有点混乱，所以这里进行说明

    渲染出数据结构

    1 在src\components\library\XTXTabs文件夹下新建了两个组件，一个是【tab选项卡容器】组件，一个是【选项卡内容组件】,【选项卡导航】由xtxtabs中通过jsx构建

    2 在src\views\member\order\index.vue导入了xtx-tabs和xtx-tabs-panel组件

    3 此时xtx-tabs里面传入了默认插槽xtx-tabs-panel，通过this.$slots.default()可以获取到默认插槽的数组

    4 遍历默认插槽数组，可以获取到默认插槽中组件的自定义属性props，然后通过该属性循环遍历构建出【选项卡导航】

    5 因为xtx-tabs-panel组件中的模板就是选项卡的内容了，所以直接将通过this.$slots.default()获取的【默认插槽】，加入到return的数据结构中即可

    6 问题来了 上述情况只能在xtx-tabs-panel是静态创建的情况下才能生效，如果通过v-for的循环遍历出xtx-tabs-panel组件，是不可行的

      所以需要同时考虑到静态和动态创建的情况

      6.1 静态创建和动态创建的数据结构区别在于，静态创建的有type属性，可以通过type.name来区分

          动态创建的插槽模板都是在children属性下

      6.2 通过this.$slots.default()获取插槽模板数组

      6.3 创建一个空数组dynamicPanels

      6.4 通过forEach遍历数组元素，通过type.name判断，

            如果是静态直接将元素推入空数组

            如果是动态的，就遍历元素的children属性，将属性中的元素推入空数组

      6.4 遍历dynamicPanels获取nav选项卡导航数据结构

      6.5 dynamicPanels和选项卡导航直接在return中返回

-->

<script>

import { provide } from 'vue'

import { useVModel } from '@vueuse/core'

export default {

  name: 'XtxTabs',

  props: {

    modelValue: {

      type: [String, Number],

      default: ''

    }

  },

  setup (props, { emit }) {

    // useVModel用法查看：src\components\library\XTXCheckbox\xtx-checkbox.vue组件

    const activeName = useVModel(props, 'modelValue', emit)

    provide('activeName', activeName)

    // 点击选项卡导航

    function switchTab (name, index) {

      // 修改选中项的标识

      activeName.value = name

      // 触发xtx-tabs 的自定义事件

      emit('tabs-check', { name, index })

    }

    return {

      activeName,

      switchTab

    }

  },

  render () {

    // 获取到默认插槽，是一个数组，数组中每个对象都有对应的属性，其中props便是我们在src\views\member\order\index.vue组件中传入的自定义属性

    // 同时该插槽也是选项卡内容,直接添加到return返回的数据结构中，免去了自己通过jsx定义选项卡内容的过程

    console.log(this.$slots.default())

    // const tabPanel = this.$slots.default() // 也就是src\components\library\XTXTabs\xtx-tabs-panel.vue组件的template内容

    const dynamicPanels = [] // 需要考虑静态和动态的情况，所以需要定义一个数组来收集不同情况下的插槽信息

    this.$slots.default().forEach(slot => {

      if (slot.type.name === 'XtxTabsPanel') { // 静态创建的插槽，添加到数组

        dynamicPanels.push(slot)

      } else {

        // 通过v-for动态创建的插槽,添加到数组

        slot.children.forEach(item => {

          dynamicPanels.push(item)

        })

      }

    })

    // 导航选项

    const nav =

    <nav>

      {

        dynamicPanels.map((item, i) => {

          // 这里绑定事件需要使用回调函数，不然还没有点击，函数就被调用了

          return <a onClick={() => this.switchTab(item.props.name, i)} class={{ active: item.props.name === this.modelValue }}>{item.props.label}</a>

        })

      }

    </nav>

    // 返回的数据结构

    return <div class="xtx-tabs">{[nav, dynamicPanels]}</div>

  }

}

</script>

<style lang='less' scoped>

.xtx-tabs {

  background: #fff;

  > nav {

    height: 60px;

    line-height: 60px;

    display: flex;

    border-bottom: 1px solid #f5f5f5;

    > a {

      width: 110px;

      border-right: 1px solid #f5f5f5;

      text-align: center;

      font-size: 16px;

      cursor: pointer;

      &.active {

        border-top: 2px solid @xtxColor;

        height: 60px;

        background: #fff;

        line-height: 56px;

      }

    }

  }

}

</style>

# 状态码 200 请求成功，但实际结果与预期不符，怎么定位是前端问题还是后端？

[状态码 200 请求成功，但实际结果与预期不符，怎么定位是前端问题还是后端？-有了 (zhipin.com)](https://youle.zhipin.com/questions/3dee1491e118d88btnVy3N-_FlA~.html)

1 查看接口请求参数（流程或文档等不规范可能会导致有部分必传参数由于文档中没有标记或标记了没有传也可以请求成功接口 200，但会导致与预期结果不符）；如是上述情况，则加上参数进行验证；查看是否与预期结果相符；

相符：前端问题，优化文档规范

反之后端问题

2查看接口响应参数，请求参数合法并且是按照接口文档中所规定的规则去传的，但响应参数不符合接口文档约定（后端问题）