Js基本知识相关模块

一.原型和原型链

原型：

①所有引用类型都有一个\_\_proto\_\_(隐式原型)属性，属性值是一个普通的对象

②所有函数都有一个prototype(原型)属性，属性值是一个普通的对象

③所有引用类型的\_\_proto\_\_属性指向它构造函数的prototype

原型链：

当访问一个对象的属性时，会先在这个对象本身属性上查找，如果没有找到，则会去它的\_proto\_隐式的原型上查找，也就是它构造函数的prototype，如果还没找到就会在构造函数的\_proto\_中查找，直到\_proto\_属性的终点，这样一层一层向上查找就会形成一个链式结构，我们称之为原型链。

二.js实现继承的方式

1. 原型链继承

function Animal (name) {

// 属性

this.name = name || 'Animal';

// 实例方法

this.sleep = function(){

console.log(this.name + '正在睡觉！');

}

}

// 原型方法

Animal.prototype.eat = function(food) {

console.log(this.name + '正在吃：' + food);

};

function Cat(){

}

Cat.prototype = new Animal();

Cat.prototype.name = 'cat';

var cat = new Cat();

console.log(cat.name);

console.log(cat.eat('fish'));

console.log(cat.sleep());

console.log(cat instanceof Animal); //true

console.log(cat instanceof Cat); //true

特点：

1.非常纯粹的继承关系，实例是子类的实例，也是父类的实例

2.父类新增原型方法/原型属性，子类都能访问到

3.简单，易于实现

缺点：

1.要想为子类新增属性和方法，必须要在new Animal()这样的语句之后执行，不能放到构造器中

2.无法实现多继承

3.来自原型对象的所有属性被所有实例共享

4.创建子类实例时，无法向父类构造函数传参

2. 构造继承

function Cat(name){

Animal.call(this);

this.name = name || 'Tom';

}

// Test Code

var cat = new Cat();

console.log(cat.name);

console.log(cat.sleep());

console.log(cat instanceof Animal); // false

console.log(cat instanceof Cat); // true

特点：

1.解决了1中，子类实例共享父类引用属性的问题

2.创建子类实例时，可以向父类传递参数

3.可以实现多继承（call多个父类对象）

缺点：

1.实例并不是父类的实例，只是子类的实例

2.只能继承父类的实例属性和方法，不能继承原型属性/方法

3.无法实现函数复用，每个子类都有父类实例函数的副本，影响性能

3.组合继承

function Cat(name){

Animal.call(this);

this.name = name || 'Tom';

}

Cat.prototype = new Animal();

var cat = new Cat();

console.log(cat.name);

console.log(cat.sleep());

console.log(cat instanceof Animal); // true

console.log(cat instanceof Cat); // true

特点：

1.弥补了方式2的缺陷，可以继承实例属性/方法，也可以继承原型属性/方法

2.既是子类的实例，也是父类的实例

3.不存在引用属性共享问题

4.可传参

5.函数可复用

缺点：

1.调用了两次父类构造函数，生成了两份实例（子类实例将子类原型上的那份屏蔽了）

4.拷贝(寄生)继承

function Cat(name){

var animal = new Animal();

for(var p in animal){

Cat.prototype[p] = animal[p];

}

this.name = name || 'Tom';

}

var cat = new Cat();

console.log(cat.name);

console.log(cat.sleep());

console.log(cat instanceof Animal); // false

console.log(cat instanceof Cat); // true

特点：

1.支持多继承

缺点：

1.效率较低，内存占用高（因为要拷贝父类的属性）

2.无法获取父类不可枚举的方法（不可枚举方法，不能使用for in 访问到）

5.实例继承

function Cat(name){

var instance = new Animal();

instance.name = name || 'Tom';

return instance;

}

var cat = new Cat();

console.log(cat.name);

console.log(cat.sleep())

console.log(cat instanceof Animal); // true

console.log(cat instanceof Cat); // false

特点：

1.不限制调用方式，不管是new 子类()还是子类(),返回的对象具有相同的效果

缺点：

1.实例是父类的实例，不是子类的实例

2.不支持多继承

6.寄生组合继承

function Cat(name){

Animal.call(this);

this.name = name || 'Tom';

}

(function(){

// 创建一个没有实例方法的类

var Super = function(){};

Super.prototype = Animal.prototype;

//将实例作为子类的原型

Cat.prototype = new Super();

})();

var cat = new Cat();

console.log(cat.name);

console.log(cat.sleep());

console.log(cat instanceof Animal); // true

console.log(cat instanceof Cat); //true

特点：

1.堪称完美

缺点：

1.实现较为复杂

三.var，let和const对比

1.使用var声明的变量，其作用域为该语句所在的函数内，且存在变量提升的现象

2.使用let声明的变量，其作用域为该语句所在的代码块内，不存在变量提升

3.let不允许在作用域内，重复申明同一个变量

4.const用于定义常量需要给初始值，并且被赋值后不能被修改，但如果是对象则可以修改里面的属性

四.es6 es7新增规范

1.let

2.const

3.模版字符串 ` `

4.解构赋值 let arr=[1,3,2]; let arr1 = [6,7,8,…arr] console.log(arr1) 结果为:[6, 7, 8, 1, 3, 2]

let obj = {id:999, name:’张三’}; let obj2 = {…obj1, phone:110} console.log(obj2)

5.箭头函数() => {}

6.Promise: 异步编程，可解决回调地址

7.类：class使得面向对象编程变得更加简单和异域理解

8.模块化：export暴露(导出)，import引用(导入)

9.函数参数默认值: function(height, color){ let height = height || 88, color = color || blue }

10.第七种数据类型Symbol var s1 = Symbol();var s2 = Symbol();s1不等于s2,Symbol函数会生成一个唯一的值,也就是每次都不相等，至于它等于多少，并不重要

12.字符串和数组新增了那些方法

字符串：1、includes 2、startswith 3、endsWith 等；数组1、Array.of //将一组数值转化为数组 2、Array.from 等

13.对象属性简写let name = ‘lilei’, let age = 18, let city = ‘cancan’; const student = {name: name, age: age, city: city} 可以简写成 const student = {name, age, city}

14. es7新增了async/await等

五.es6提供的枚举有for…in， Object. keys()，JSON. stringify()，Object. assign()

1.for…in 的遍历用法和Object. keys()一样，for(let item in obj){ } 可以遍历数组，字符串，对象，建议只用来遍历对象，而且在遍历时也会遍历到原型的方法和属性，如果想要避免的可以通过hasOwnPropetty属性判断。

2.Object. keys({name：'gwh', age: 28})打印出来的是数组[name, age]，如果是Object. keys("adc")打印出来的是下标数组[“0”,”1”,”2”]，同理如果是数组打印出来的也是Object. keys(['a', 'b', ‘c'])打印出来的是["0","1","2"]。

3.for…of的用法：它主要用于遍历数组的元素值。for(let item of [6, 5, 4, 3]){ console. log(item) }打印出来的分别是6,5,4,3。

4.break可以停止for...in和for...of循环 。

六.事件循环机制（Event Loop）

1，同步和异步，所有的线程都有同步队列和异步队列，立即执行的任务队列都是属于同步任务，如一个简单的函数；axios，ajax，settimeout，promise().then等这些就是异步任务。

2，任务队列-事件循环，同步任务立即执行，进入到主线程当中，异步任务会放在异步任务队列当中，等待同步执行完成后，返回来将异步任务放到主线程中执行，反复的这样循环，这就是事件循环。

3，宏观任务和微观任务；先执行微观任务，再执行宏观任务。微观任务主要有：promise，MutaionObserver，process. nextTick。宏观任务主要有：setTimeout、setInterval、I/O、UI交互事件、script整体代码等

七.函数的闭包以及解决方法

函数闭包

闭包指的是：能够访问另一个函数作用域的变量的函数

function outer() {

var a = ‘变量1’;

var inner = function() {

console.log(a)

};

return inner //inner就是一个闭包函数，因为它能访问到outer函数的作用域

}

1. 函数嵌套函数

2. 面向对象编程

3. 函数内部可以引用函数外部的参数和变量

4. 参数和变量不会被垃圾回收机制回收

解决方法

1，立即执行函数

2，使用let 声明变量

八.this的指向问题

普通函数function中this的指向：

1.如果函数没有被上一级对象调用，那么this指向的是window

2.如果这个函数被上一级的对象调用，那么this指向的是上一级的对象

3.如果这个函数中包涵多个对象，尽管这个函数是被最外层调用的，那么this的指向就是上一级对象。

4. 箭头函数里面没有 **this** ，所以要向上层作用域查找

总结：this永远指向的是最后调用它的对象，也就是看它执行时是谁调用

箭头函数和普通函数的区别

1.箭头函数是匿名函数，不能作为构造函数，不能使用new

2.箭头函数不绑定arguments，取而代之的是用rest参数…解决

3.箭头函数不绑定this，会捕获上下文的this值，作为自己的this值，也就是定义时所在对象

4.箭头函数通过call()或apply()方法调用一个函数时，只传入一个参数，对this指向并没有影响

5.箭头函数没有原型属性prototype

6.箭头函数不能当作generator函数，不能使用yield关键字

7.箭头函数不支持重命名函数参数，普通函数的函数支持重命名参数

九.深拷贝的几种方法

1.JSON.parse(JSON.stringify(arr));

2.函数方式，具体的是递归的思想，如下

function deepclone(obj){

let newObjClone = Array.isArray(obj) ? [] : {};

if(obj && typeof obj === ‘object’){

for(var key in obj){

if(obj.hasOwnProperty(key)){

if(obj[key] && typeof obj[key] === ‘object’){

obj[key] = deepclone(obj[key]);

}else{

newObjClone[key] = obj[key]

}

}

}

};

return newObjClone

}

3.Object.assign()

十.解决回调地狱的方法，在node里实现同步方法

一，拆解function

二，通过new Promise().then的链式调用 注意他其中的一些简写方法

三，通过es7的异步函数async/await

四，通过Generator的函数暂停执行的效果

在node里实现同步方法

一，引用nodejs的async库

二，通过函数的回调方法

三，通过promise. then方法

十一.类型判断

1、typeof

typeof '1'                 // 'String'

typeof 1                  // 'Number'

typeof false            // 'Boolean'

typeof undefined    // 'undefined'

typeof Symbol()      // 'Symbol'

typeof null               // 'Object' 类型标签存储在单元的低位。二进制的前三位为0的话，系统将其判断为Object类型

typeof {}                  //'Object'

typeof []                   // 'Object'

typeof Function       // 'function'

2、instance(判断已知对象类型的方法)，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

[] instanceof Array // true

({}) instanceof Object // true

(()=>{}) instanceof Function // true

Array, Object, Function的大小写不能错

3、Object.prototype.toString.call()

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

Object.prototype.toString.call('snow')       // '[object String]'  
Object.prototype.toString.call(1)               // '[object Number]'  
Object.prototype.toString.call(true)           // '[object Boolean]'  
Object.prototype.toString.call(Symbol())   // '[object Symbol]'  
Object.prototype.toString.call(null)            // '[object Null]'  
Object.prototype.toString.call(undefined)  // '[object Undefined]'

Object.prototype.toString.call({})               // '[object Object]'  
Object.prototype.toString.call([1,2])          // '[object Array]'   
Object.prototype.toString.call(()=>{})        // '[object Function]'

4、根据对象的constructor判断：constructor

var c= [1,2,3];

var d = new Date();

var e = function(){alert(111);};

console(c.constructor === Array)//true

console(d.constructor === Date)//true

console(e.constructor === Function)//true

十二.深度优先遍历和广度优先遍历

//广度优先遍历

const data = [

{

name: 'a',

children: [

{ name: 'b', children: [{ name: 'e' }] },

{ name: 'c', children: [{ name: 'f' }] },

{ name: 'd', children: [{ name: 'g' }] },

],

},

{

name: 'a2',

children: [

{ name: 'b2', children: [{ name: 'e2' }] },

{ name: 'c2', children: [{ name: 'f2' }] },

{ name: 'd2', children: [{ name: 'g2' }] },

],

}

];

function widthTravel(data){

let result = [];

let quene = data;

while(quene.length > 0) {

let sortItem = quene.shift();

result.push(sortItem.name);

sortItem.children && quene.push(...sortItem.children);

};

console.log(result.join(','));

};

widthTravel(data);

//深度优先遍历

const data = [

{

name: 'a',

children: [

{ name: 'b', children: [{ name: 'e' }] },

{ name: 'c', children: [{ name: 'f' }] },

{ name: 'd', children: [{ name: 'g' }] },

],

},

{

name: 'a2',

children: [

{ name: 'b2', children: [{ name: 'e2' }] },

{ name: 'c2', children: [{ name: 'f2' }] },

{ name: 'd2', children: [{ name: 'g2' }] },

],

}

];

function deeptravel(data){

let result = [];

data.forEach((item)=> {

let map = (data) => {

result.push(data.name);

data.children && data.children.forEach(child => { map(child) });

};

map(item);

});

return result.join(',');

};

deeptravel(data);

十三.函数柯里化

在一个函数中，首先填充几个参数，然后再返回一个新的函数的技术，称为函数的柯里化。通常可用于在不侵入函数的前提下，为函数 预置通用参数，供多次重复调用。

const add = function add(x) {

return function (y) {

return x + y

}

}

const add1 = add(1)

add1(2) === 3

//柯里化封装的函数

function curry(fn, curArgument){

return function(){

let args = [...arguments];

if(curArgument !== undefined){

args = args.concat(curArgument);

// args.push(...curArgument)

};

if(fn.length > args.length){

return curry(fn, args)

};

// return fn.apply(this, args);

return fn.apply(null, args);

// return fn(...args)

}

};

function sum(a,b,c){

console.log(a+b+c);

};

const arrfun = curry(sum);

arrfun(1,2,3);

arrfun(1,2)(3);

arrfun(1)(2)(3);

十四.js的new过程

当我们new一个构造函数，得到的实例继承了构造器的构造属性(this.name这些)以及原型上的属性。主要有三步：

1. 创建一个空对象，将它的引用赋给 this，继承函数的原型。

2. 通过 this 将属性和方法添加至这个对象

3. 最后返回 this 指向的新对象，也就是实例（如果没有手动返回其他的对象）

十五.es高阶函数

主要是filter(返回值为布尔值)，reduce, map其使用如下

let nums = [16,38,49,88,291,32,231,123,101,22]；

//新的写法: filter / map / reduce

let new\_nums = nums.filter(function (n) {

return n < 100

})

// map 映射函数

let new\_nums1 = new\_nums.map(function (n) {

return n \* 3

})

// reduce(func(上一次结果 初始值为默认值，数组的n),默认值=0) 对数组中所有的内容进行汇总。

let new\_nums2 = new\_nums1.reduce(function (prev,n){

return prev + n

},0)

//直接可以写成

let new\_writeNums = nums.filter(function (n){return n < 100}).map

(function(n){return n \* 3}).reduce(function(prev,n){return prev + n},0)

十六.Promise和async/await以及generator

1.promise

(1)

①Promise是异步编程的一种解决方案,它也主要是用来解决回调地狱的问题。

②promise是一个对象，代表一个异步操作，有三种状态，进行中(pending)，成功(fulfilled)，失败(rejected)。只有异步操作的结果的可以决定当前是哪种状态，promise一旦新建执行，就没有办法中途停止。

③promise允许函数作为参数传入， resolve为成功的回调， reject为失败的回调，当条件满足时，res/rej会将成功或失败的结果暴露出来。

④通过.then拿到暴露出来的结果，.then是一个全新的promise函数，.then允许两个函数作为参数，一个函数拿到成功的暴露结果，一个拿失败暴露的结果。

⑤通过catch获取一个指定发生错误时的回调函数。error为错误信息。

⑥一般来说，不要在then方法里面定义 reject状态的回调函数（即then的第二个参数），总是使用catch方法。  
⑦通过return Promise.reject('break without exception.’)来跳出Promise的链式调用。如下面的链式调用

let needBreak = true;

let p = new Promise((resolve, reject) => { resolve(‘step1') });

p.then(data => {

console.log(data);

return 'step2';

}).then(data => {

console.log(data);

if (needBreak) {

//跳出链式调用

return Promise.reject('break without exception.');

}

return 'step3';

}).then(data => {

console.log(data);

return 'step4';

}).catch(reason => {

console.log(reason);

}).finally(() => {

console.log('finished.');

}); 输出结果为//step1 ，step2，break without exception，finished

//——————————————————————————分割线

const pro=new Promise((resolve,reject)=>{

if(!true){ resolve('123')}else{reject("4576")}}).then((res)=>{

console.log(res)},(rej)=>{ console.log(rej)}).catch((error)=>{

console.log(error)

})

(2)

Promise.all方法用于将多个Promise实例，包装成一个新的Promise实例。只有当作为参数所有的promise函数运行完毕，才会执行.then回调。

如果作为参数的 Promise 实例，自己定义了catch方法，那么它一旦被出错，并不会触发Promise.all()的catch方法。

const promises = [2, 3, 5, 7, 11, 13].map(function (id) {

return getJSON('/post/' + id + ".json");

});

Promise.all(promises).then(function (posts) {

// ...

}).catch(function(reason){

// ...

});

(3)

Promise.race()法用于将多个Promise实例，包装成一个新的Promise实例。只有当作为参数所有的promise函数有一个运行完毕，就会执行.then回调。

const promises = [2, 3, 5, 7, 11, 13].map(function (id) {

return getJSON('/post/' + id + ".json");

});

Promise.race(promises).then(function (posts) {

// ...

}).catch(function(reason){

// ...

});

2.asynic/await

①async是一个修饰符，async定义的函数会默认的返回一个Promise对象的reslove的值，因此对async函数可以直接进行then操作，返回的值即为then方法传入分参数，

async function fun() { return 1 };

fun().then(x => { console. log(x) });

要写return，要不then里面获取不到x的值，return相当于调用reslove(1)

②.await是一个修饰符，必须放在async函数的域里面。作用就是获取Promise函数中reslove或者是reject的值，如果await后面并不是一个promise的返回值，则会按同步程序返回值处理，为undefine

function doubleAfter2seconds(num) {

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

setTimeout(() => {

resolve(2 \* num)

}, 2000);

} )

};

async function testResult() {

let first = await doubleAfter2seconds(30);

console.log(first);

let second = await doubleAfter2seconds(50);

console.log(second);

let third = await doubleAfter2seconds(30);

console.log(first + second + third);

}

testResult()

3.generator函数以及yield

在Javascript中，一个函数一旦开始执行，就会运行到最后或遇到return时结束，运行期间不会有其它代码能够打断它，也不能从外部再传入值到函数体内;

而Generator函数（生成器）的出现使得打破函数的完整运行成为了可能，其语法行为与传统函数完全不同;

Generator函数是ES6提供的一种异步编程解决方案，有几个显著的特征：

①function关键字与函数名之间有一个星号 "\*" （推荐紧挨着function关键字

②函数体内使用 yield 表达式，定义不同的内部状态 （可以有多个yield）

③直接调用 Generator函数并不会执行，也不会返回运行结果，而是返回一个遍历器对象（Iterator Object）

④依次调用遍历器对象的next方法，遍历 Generator函数内部的每一个状态

function\* gen() {

yield 'hello'

yield 'world'

return 'ending'

}

let it = gen()

it.next() // {value: "hello", done: false}

it.next() // {value: "world", done: false}

it.next() // {value: "ending", done: true}

it.next() // {value: undefined, done: true}

每次调用Iterator对象的next方法时，内部的指针就会从函数的头部或上一次停下来的地方开始执行，直到遇到下一个 yield 表达式或return语句暂停。换句话说，Generator 函数是分段执行的，yield表达式是暂停执行的标记，而 next方法可以恢复执行。

yield 表达式

①yield 表达式只能用在 Generator 函数里面，用在其它地方都会报错

②yield 表达式如果用在另一个表达式中，必须放在圆括号里面

function\* demo() {

console.log('Hello' + yield); // SyntaxError

console.log('Hello' + yield 123); // SyntaxError

console.log('Hello' + (yield)); // OK

console.log('Hello' + (yield 123)); // OK

}

③yield 表达式用作参数或放在赋值表达式的右边，可以不加括号

function\* demo() {

foo(yield 'a', yield 'b'); // OK

let input = yield; // OK

}

④yield 表达式和return语句的区别

相似：都能返回紧跟在语句后面的那个表达式的值

区别：每次遇到 yield，函数就暂停执行，下一次再从该位置继续向后执行；而 return 语句不具备记忆位置的功能; 一个函数只能执行一次 return 语句，而在 Generator 函数中可以有任意多个 yield

具体还有更多的情况请百度

十七.事件代理和事件委托

“事件代理”即是把原本需要绑定在子元素的响应事件（click、keydown……）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。比如在ul上代理所有li的click事件。

十八.数组的去重

总共有十多种，这里列举7种

1. 利用es6的set去重

function unique1(arr) {

return Array.from(new Set(arr)); // 利用Array.from将Set结构转换成数组

}

2. 利用双for以及splice去重

function unique2(arr) {

for(var i = 0; i < arr.length ; i++) {  
 for(var j = i + 1; j < arr.length; j++){

if(arr[i] == arr[j]) {  
 arr.splice(j, 1);

j—;

}  
 }  
 };

return arr

}

3. 利用indexof去重

function unique3(arr) {

let newArr = [];

for(var i = 0; i < arr.length; i++) {  
 if(arr.indexOf(arr[i]) == -1){  
 newArr.push(arr[i].);  
 }  
 };

return newArr;

}

4. 利用sort()方法

function unique4(arr) {

var newArry = arr.sort(), arrys = [newArry[0]];

for(var i = 1; i < newArry.length; i++) {  
 if(arr[i-1] !== arr[i]){  
 arrys.push(arr[i].);  
 }  
 };

return arrys;

}

5. 利用filter()方法

function unique5(arr) {

return arr.filter((item, index) => {  
 return arr.indexOf(item, 0) === index;  
 })

}

6. 利用includes方法

function unique6(arr) {

let array = [];

for(var i = 1; i < arr.length; i++) {  
 if(!array.includes(arr[i])){  
 array.push(arr[i].);  
 }  
 };

return array;

}

7. 利用对象的属性不能相同的特点进行去重

function unique7(arr) {

let newObj = {}, array = [];

for(var i = 1; i < arr.length; i++) {  
 if (!newObj[arr[i]]) {

newObj[arr[i]] = 1

array.push(arr[i])

} };

return array;

}

数组中相同对象元素的去重，例如

let objArray = [{name: ‘gwh’, age: 18}, {name: ‘xlj’, age: 16}, {age:18, name: ‘gwh’}];

function unique(arrays) {  
 let newobj = {};  
 arrays.forEach((item) => {  
 newobj[JSON.stringify(item, [‘name’, ‘age’])] = item;  
 });

let newList = Object.keys(newobj).map((keys) => {  
 return JSON.parse(keys)  
 });

return newList;  
}

十九.数组的方法

数组的方法有：

shift(删除第一个并返回), unshift(添加到第一个), pop(), push(), map(), filter(), sort(), splice(), join(), reduce(), find(), foreach(), reserve()

二十. 防抖和截流

防抖函数:

function debounch(fun, dealy) {  
 let timer = null;

return function(..args){  
 if(timer) { clearTimeout(timer) };  
 timer = setTimeout(() => {  
 fun.apply(this, args)  
 }, dealy)  
 }  
 }

截流函数:

function throttle(func, dealy) {  
 let prev = Data.now();

return function(){

let arg = arguments;  
 var now = Data.now();  
 if(now - prev >= dealy) {  
 func.apply(this, arg);

prev = Data.now();  
 };

}  
 }

二十一. 前端的性能优化

1. 减少 HTTP请求数

(1). 从设计实现层面简化页面

(2). 合理设置 HTTP缓存

(3). 资源合并与压缩

(4). CSS Sprites

(5). 图片懒加载

2. 将外部脚本置底

3. javascript脚本懒加载

4. 将 CSS放在 HEAD中

5. 减少不必要的 HTTP跳转

6. 避免重复的资源请求

7. 减少重绘回流

8. 使用cdn

二十二. 数据类型

数据类型可以分为两类：一、原始类型（基本数据类型）；二、对象类型（即引用数据类型）。

基本数据类型：String、Number、Null、Undefined、Boolean 、Symbol

对象类型（即引用数据类型）：对象类型Object；比如：Object、array、function、data等。

Null:表示声明对象为赋值，Null类型的值只有一个就是null，null这个值专门用来表示这个为空的对象。使用typeof检查null时返回object。

二十三. 数据结构

（一）、数组：数组是可以再内存中连续存储多个元素的结构，在内存中的分配也是连续的，数组中的元素通过数组下标进行访问，数组下标从0开始。

优点：

1、按照索引查询元素速度快

2、按照索引遍历数组方便

缺点：

1、数组的大小固定后就无法扩容了

2、数组只能存储一种类型的数据

3、添加，删除的操作慢，因为要移动其他的元素。

（二）、栈：栈是一种特殊的线性表，仅能在线性表的一端操作，栈顶允许操作，栈底不允许操作。栈的特点是：先进后出，或者说是后进先出，从栈顶放入元素的操作叫入栈，取出元素叫出栈。

（三）、队列：队列与栈一样，也是一种线性表，不同的是，队列可以在一端添加元素，在另一端取出元素，也就是：先进先出。使用场景：因为队列先进先出的特点，在多线程阻塞队列管理中非常适用。

（四）、链表：链表是物理存储单元上非连续的、非顺序的存储结构，数据元素的逻辑顺序是通过链表的指针地址实现，每个元素包含两个结点，一个是存储元素的数据域 (内存空间)，另一个是指向下一个结点地址的指针域。根据指针的指向，链表能形成不同的结构，例如单链表，双向链表，循环链表等。

链表的优点：

链表是很常用的一种数据结构，不需要初始化容量，可以任意加减元素；

添加或者删除元素时只需要改变前后两个元素结点的指针域指向地址即可，所以添加，删除很快；

缺点：

因为含有大量的指针域，占用空间较大；

查找元素需要遍历链表来查找，非常耗时。

适用场景：

数据量较小，需要频繁增加，删除操作的场景

（五）、树：树是一种数据结构，它是由n（n>=1）个有限节点组成一个具有层次关系的集合。把它叫做 “树” 是因为它看起来像一棵倒挂的树，也就是说它是根朝上，而叶朝下的。它具有以下的特点：

1. 每个节点有零个或多个子节点；

2. 没有父节点的节点称为根节点；

3. 每一个非根节点有且只有一个父节点；

4. 除了根节点外，每个子节点可以分为多个不相交的子树；

（六）、散列表：散列表，也叫哈希表，是根据关键码和值 (key和value) 直接进行访问的数据结构，通过key和value来映射到集合中的一个位置，这样就可以很快找到集合中的对应元素。

（七）、堆：堆是一种比较特殊的数据结构，可以被看做一棵树的数组对象，具有以下的性质：

1. 堆中某个节点的值总是不大于或不小于其父节点的值；

2. 堆总是一棵完全二叉树。

将根节点最大的堆叫做最大堆或大根堆，根节点最小的堆叫做最小堆或小根堆。（八）、图：图是由结点的有穷集合V和边的集合E组成。其中，为了与树形结构加以区别，在图结构中常常将结点称为顶点，边是顶点的有序偶对，若两个顶点之间存在一条边，就表示这两个顶点具有相邻关系。

按照顶点指向的方向可分为无向图和有向图。

图是一种比较复杂的数据结构。这里不做过多的介绍。

二十四. 手写Promise

function Promise(excutor) {

let self = this

self.status = 'pending'

self.value = null

self.reason = null

self.onFulfilledCallbacks = []

self.onRejectedCallbacks = []

function resolve(value) {

if (self.status === 'pending') {

self.value = value

self.status = 'fulfilled'

self.onFulfilledCallbacks.forEach(item => item())

}

}

function reject(reason) {

if (self.status === 'pending') {

self.reason = reason

self.status = 'rejected'

self.onRejectedCallbacks.forEach(item => item())

}

}

try {

excutor(resolve, reject)

} catch (err) {

reject(err)

}

}

Promise.prototype.then = function (onFulfilled, onRejected) {

onFulfilled = typeof onFulfilled === 'function' ? onFulfilled : function (data) {resolve(data)}

onRejected = typeof onRejected === 'function' ? onRejected : function (err) {throw err}

let self = this

if (self.status === 'fulfilled') {

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

try {

let x = onFulfilled(self.value)

if (x instanceof Promise) {

x.then(resolve, reject)

} else {

resolve(x)

}

} catch (err) {

reject(err)

}

})

}

if (self.status === 'rejected') {

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

try {

let x = onRejected(self.reason)

if (x instanceof Promise) {

x.then(resolve, reject)

} else {

resolve(x)

}

} catch (err) {

reject(err)

}

})

}

if (self.status === 'pending') {

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

self.onFulfilledCallbacks.push(() => {

let x = onFulfilled(self.value)

if (x instanceof Promise) {

x.then(resolve, reject)

} else {

resolve(x)

}

})

self.onRejectedCallbacks.push(() => {

let x = onRejected(self.reason)

if (x instanceof Promise) {

x.then(resolve, reject)

} else {

resolve(x)

}

})

})

}

}

Promise.prototype.catch = function (fn) {

return this.then(null, fn)

}

二十五. 前端工程化的理解

从模块化、组件化、规范化、自动化四个方面去说

（一）模块化：模块化就是将一个大文件拆分成相互依赖的小文件，再进行统一的拼装和加载。

JS的模块化：在ES6已经在语言层面上规定了模块系统。

1. 用++Webpack + Babel++将所有模块打包成一个文件同步加载，也可以搭乘多个chunk异步加载；

2. 用++System+Babel++主要是分模块异步加载；

3. 用浏览器的<script type="module">加载

css的模块化：用SASS、LESS、Stylus等预处理器实现CSS的文件拆分

资源的模块化：主要就是Webpack的引入，资源模块化后，优点是：

依赖关系单一化。所有CSS和图片等资源的依赖关系统一走JS路线，无需额外处理CSS预处理器的依赖关系，也不需处理代码迁移时的图片合并、字体图片等路径问题。

资源处理集成化。现在可以用loader对各种资源做各种事情，比如复杂的vue-loader等等。

项目结构清晰化。使用Webpack后，你的项目结构总可以表示成这样的函数： dest = webpack(src, config)。

（二）组件化：从UI拆分下来的每个包含模板(HTML)+样式(CSS)+逻辑(JS)功能完备的结构单元，我们称之为组件。组件化≠模块化，模块化只是在文件层面上，对代码或资源的拆分；而组件化是在设计层面上，对UI的拆分。

（三）规范化：

1. 目录结构的合理制定：有助于提高项目的逻辑结构合理性；对应扩展和合作；方便资源的统一定位管理。

2. 编码规范包括：HTML规范，CSS规范，JS规范，图片规范，命名规范，前后端接口规范，职责分离，文档规范，组件管理，git分支管理，commit描述规范，commit描述规范

（四）自动化：前端工程化的很多脏活累活都应该交给自动化工具来完成。包括：1.图标合并 2.持续继承 3.自动化构建 4.自动化部署 5.自动化测试

二十六. 前端文件的批量上传

前端处理逻辑如下：

<div>

<form method="post"enctype="multipart/form-data" data-ajax="false">

//@\*multiple=“multiple”多选必备\*@

<input type="file" name="Files" id="Files" multiple="multiple" value="上传Files" />

<br /><br />

//@\*type=“button”以后用button\*@

<input type="button" value="提交Files" onclick="uplod\_Files()" />

</form>

</div>

<script>

function uplod\_Files() {

var formData = new FormData();

//就像cookie一样用,存入files[i];数组形式

var files = document.getElementById("Files").files;

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

{

formData.append("files\_"+i, files[i]);

};

$.ajax({

url: "/ToPdf/Get\_Files",//请求地址

dataType: "json",//数据格式

type: "POST",//请求方式

async: true,//是否异步请求

cache: false,//上传文件无需缓存

contentType: false,//必须

//用于对data参数进行序列化处理 这里必须false

processData: false,

data: formData,

success: function (data) {

}

})

}

</script>

二十七. ES6中的Map和Set

Map

Map对象保存键值对。任何值(对象或者原始值) 都可以作为一个键或一个值。构造函数Map可以接受一个数组作为参数。一个Object的键只能是字符串或者 Symbols，但一个Map 的键可以是任意值。初始化Map需要一个二维数组，或者直接初始化一个空Map。

Map具有以下方法：

1. set(key, val): 向Map中添加新元素

2. get(key): 通过键值查找特定的数值并返回

3. has(key): 判断Map对象中是否有Key所对应的值，有返回true,否则返回false

4. delete(key): 通过键值从Map中移除对应的数据

5. clear(): 将这个Map中的所有元素删除

6. size：返回Map对象中所包含的键值对个数

var m = new Map([['Michael', 95], ['Bob', 75], ['Tracy', 85]]);

console.log(m) // {"Michael" => 95, "Bob" => 75, "Tracy" => 85}

m.get('Michael'); // 95

var m = new Map(); // 空Map

m.set('Adam', 67); // 添加新的key-value

m.has('Adam'); // 是否存在key 'Adam': true

m.get('Adam'); // 67

m.delete('Adam'); // 删除key 'Adam'

m.get('Adam'); // undefined

遍历方法

1. keys()：返回键名的遍历器

2. values()：返回键值的遍历器

3. entries()：返回键值对的遍历器

4. forEach()：使用回调函数遍历每个成员

const map = new Map([['a', 1], ['b', 2]])

for (let key of map.keys()) {

console.log(key)

}

// "a"

// "b"

for (let value of map.values()) {

console.log(value)

}

// 1

// 2

for (let item of map.entries()) {

console.log(item)

}

// ["a", 1]

// ["b", 2]

// 或者

for (let [key, value] of map.entries()) {

console.log(key, value)

}

// "a" 1

// "b" 2

Set

Set也是一组key的集合，与Map类似。但是区别是Set不存储value，并且它的key不能重复。创建一个Set，需要提供一个Array作为输入，或者直接创建一个空Set.

var s1 = new Set(); // 空Set

var s = new Set([1, 2, 3, 3, '3']);

console.log(s); // Set {1, 2, 3, “3"}

Set实例对象的方法

1. add(value)：添加某个值，返回 Set 结构本身(可以链式调用)。

2. delete(value)：删除某个值，删除成功返回true，否则返回false。

3. has(value)：返回一个布尔值，表示该值是否为Set的成员。

4. clear()：清除所有成员，没有返回值。

const mySet = new Set(['a', 'a', 'b', 1, 2, 1])

console.log(mySet) // {'a', 'b', 1, 2}

myset.add('c').add({'a': 1})

console.log(mySet) // {'a', 'b', 1, 2, 'c', {a: 1}}

console.log(mySet.size) // 6

mySet.has(2) // true

遍历方法

keys()：返回键名的遍历器。

values()：返回键值的遍历器。

entries()：返回键值对的遍历器。

forEach()：使用回调函数遍历每个成员。

二十八. 二分查找的算法

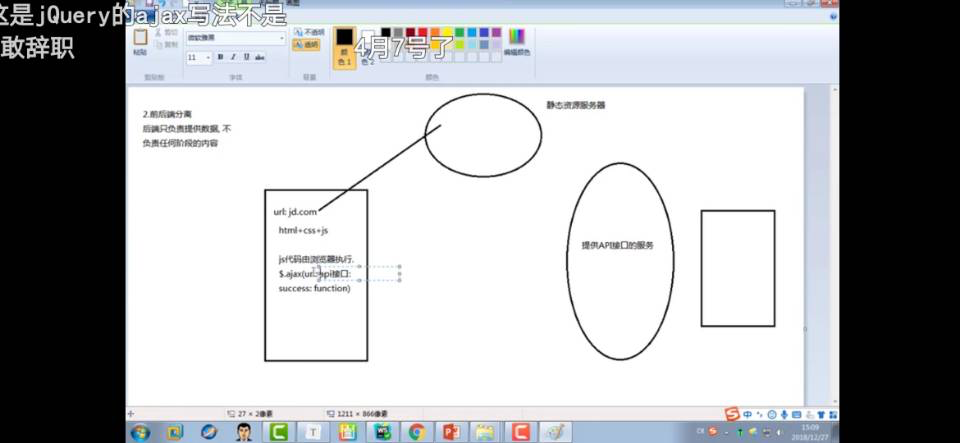
function second\_binary(arr, key) {  
 let low = 0, high = arr.length - 1;  
 while(low <= high) {  
 let mid = parseInt(low + high)/2;  
 if(arr[mid] == key) {  
 return mid  
 }else if(key > arr[mid] ){  
 low = mid + 1;  
 }else if(key < arr[mid] ){  
 high = mid - 1;  
 }else {

return -1;

}  
 }  
}

二十九. 前后端分离的看法

之前是通过jsp去写的 包括HTML CSS java代码都放在一起并且存储在服务器里，服务器根据客户端的url去返回所有的资源，也就是说整个页面在服务器已经全部渲染好了再给浏览器的。

ajax的出现促进前后端的分离，CSS HTML js代码放在一个服务器上 接口api放在另一个服务器上，到客户端url请求的时候，会先去静态服务器上获取HTML CSS js 文件返回浏览器直接渲染，然后在通过执行js代码去另一台接口api服务器去获取数据，这样便于分离，而且一台接口api服务器可以提供多种平台的开发使用，如下图

前后端通信以及网络

一.http的请求方式

get：向特定资源发出请求（请求指定页面信息，并返回实体主体）。

post: 向指定资源提交数据进行处理请求（提交表单、上传文件），又可能导致新的资源的建立或原有资源的修改。

put：向指定资源位置上上传其最新内容（从客户端向服务器传送的数据取代指定文档的内容）

delete: 请求服务器删除request-URL所标示的资源（请求服务器删除页面）

head: 与服务器索与get请求一致的相应，响应体不会返回，获取包含在小消息头中的原信息（与get请求类似，返回的响应中没有具体内容，用于获取报头）

options: 返回服务器针对特定资源所支持的HTML请求方法 或web服务器发送测试服务器功能（允许客户端查看服务器性能）

trace: 回显服务器收到的请求，用于测试和诊断

connect: HTTP/1.1协议中能够将连接改为管道方式的代理服务器

二.get和post的区别

（1） 在客户端，Get方式在通过URL提交数据（用来从服务器取东西），数据在URL中可以看到；POST方式，数据放置在HTML HEADER内提交。

（2）GET方式提交的数据最多只能有1024字节，而POST则没有此限制。

（3）安全性问题。正如在（1）中提到，使用 Get的时候，参数会显示在地址栏上，而 Post不会。所以，如果这些数据是中文数据而且是非敏感数据，那么使用 get；如果用户输入的数据不是中文字符而且包含敏感数据，那么还是使用 post为好。

(4） 安全的和幂等的，GET请求一般不应产生副作用；POST表示可能改变服务器上的资源的请求。

三.请求返回状态码：

200 OK 当您的操作将在响应正文中返回数据时，出现此结果。

204 No Content 当您的操作成功,但不在响应正文中返回数据时,出现此结果。

304 Not Modified（重定向） 当测试实体自上次检索以来是否被修改时，出现此结果。

403 Forbidden 客户端未能获得授权。这通常是在401之后输入了不正确的用户名或密码。

401 Unauthorized 客户端无权访问该资源。这通常会使得浏览器要求用户输入用户名和密码，以登录到服务器。

413 Payload Too Large（客户端错误） 当请求长度过长时，出现此结果。

400 BadRequest（客户端错误） 当参数无效时，出现此结果。

404 Not Found（客户端错误） 当资源不存在时，出现此结果。

405 Method Not Allowed（客户端错误）由于方法和资源组合不正确而出现此错误。 例如，您不能对一个实体集合使用 DELETE 或 PATCH。

412 Precondition Failed 客户端错误

501 Not Implemented（服务器错误)当未实施某个请求的操作时，出现此结果

503 Service Unavailable (服务器错误)当Web API服务不可用时，出现此结果。

四.localstorage和sessionstorage以及cookie

1. cookie：它主要用于保存登陆信息，比如登陆某个网站市场可以看到’记住密码’，这就是通过在cookie中存入一段辨别用户身份的数据来实现的。

2. sessionStorage：会话，是可以将一部分数据在当前会话中保存下来，刷新页面数据依旧存在。但是页面关闭后，sessionStorage中的数据就会被清空。

3. localStorage：localStorage生命周期是永久，这意味着除非用户显示在浏览器提供的UI上清除localStorage信息，否则这些信息将永远存在。存放数据大小为一般为5MB,而且它仅在客户端（即浏览器）中保存，不参与和服务器的通信。

三者之间的区别

五.浏览器输入URL到页面展示的全过程

1. 输入url

(1). 先从浏览器缓存里找IP,因为浏览器会缓存DNS记录一段时间，如果有直接从缓存中把网页展示出来。如没找到，进入到第二步。

2. DNS(域名)解析

(1). 请求一旦发起，浏览器首先要做的事情就是解析这个域名，一般来说，浏览器会首先查看本地硬盘的 hosts 文件，看看其中有没有和这个域名对应的规则，如果有的话就直接使用 hosts 文件里面的 ip 地址。

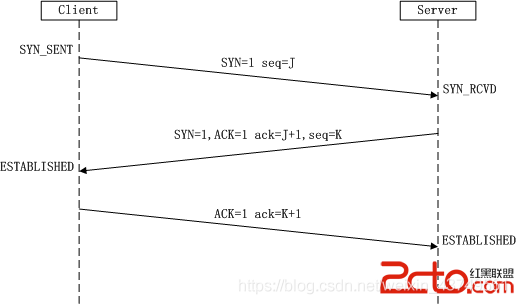
(2). 如果在本地的 hosts 文件没有能够找到对应的 ip 地址，浏览器会发出一个 DNS请求到本地DNS服务器 。本地DNS服务器一般都是你的网络接入服务器商提供，比如中国电信，中国移动。

(3). 到达本地服务器后，本地DNS服务器会首先查询它的缓存记录，如果缓存中有此条记录，就可以直接返回结果。如果没有，本地DNS服务器还要向DNS根服务器进行查询。

(4). 根DNS服务器没有记录具体的域名和IP地址的对应关系，而是告诉本地DNS服务器，你可以到域服务器上去继续查询，并给出域服务器的地址。这种过程是迭代的过程。(例如:.com的顶级域名服务器)。

(5). 本地DNS服务器继续向域服务器发出请求，在这个例子中，请求的对象是.com域服务器。.com域服务器收到请求之后，也不会直接返回域名和IP地址的对应关系，而是告诉本地DNS服务器，你的域名的解析服务器的地址。

(6). 最后，本地DNS服务器向域名的解析服务器发出请求，这时就能收到一个域名和IP地址对应关系，本地DNS服务器不仅要把IP地址返回给用户电脑，还要把这个对应关系保存在缓存中，以备下次别的用户查询时，可以直接返回结果，加快网络访问。



3. 建立TCP连接

浏览器通过DNS获取到web服务器真的IP地址后，便向web服务器发起tcp连接请求，通过TCP三次握手建立好连接后，浏览器便可以将HTTP请求数据通过发送给服务器了。

(1) .第一次握手：主机A发往主机B，主机A的初始序号是X，设置SYN位，未设置ACK位，等待服务端B确认。

(2) .第二次握手：主机B发往主机A，主机B的初始序号是Y，确认号(ACK)是X+1，X+1确认号暗示己经收到主机A发往主机B的同步序号。设置SYN位和ACK位。

(3) .第三次握手：主机A发往主机B，主机A的序号是X+1，确认号是Y+1，Y+1确认号暗示已经收到主机B发往主机A的同步序号。设置ACK位，未设置SYN位

4.浏览器向 web 服务器发送一个 HTTP 请求

HTTP请求是一个基于TCP协议之上的应用层协议——超文本传输协议。一个http事务由一条(从客户端发往服务器的)请求命令和一个(从服务器发回客户端的)响应结果组成。

5.服务器处理请求

服务器是如何处理我们的请求的呢，后端从在固定的端口接收到TCP报文开始，它会对TCP连接进行处理，对HTTP协议进行解析，并按照报文格式进一步封装成HTTP Request对象，供上层使用。

一些大一点的网站会将你的请求到反向代理服务器中，因为当网站访问量非常大，网站越来越慢，一台服务器已经不够用了。于是将同一个应用部署在多台服务器上，将大量用户的请求分配给多台机器处理。此时，客户端不是直接通过HTTP协议访问某网站应用服务器，而是先请求到Nginx，Nginx再请求应用服务器，然后将结果返回给客户端，这里Nginx的作用是反向代理服务器。同时也带来了一个好处，其中一台服务器万一挂了，只要还有其他服务器正常运行，就不会影响用户使用。

6.服务器返回一个 HTTP 响应

服务器收到了我们的请求，也处理我们的请求，到这一步，它会把它的处理结果返回，也就是返回一个HTTP响应。

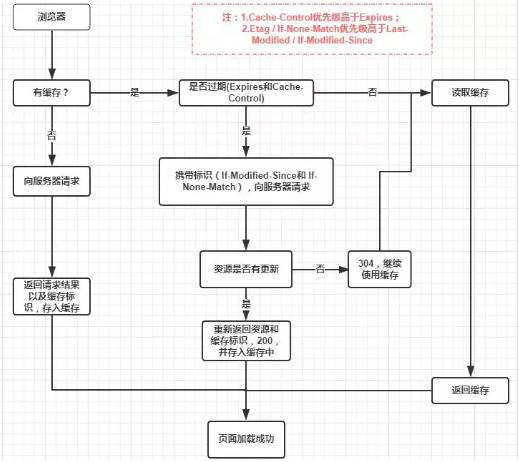
HTTP响应与HTTP请求相似，HTTP响应也由3个部分构成，分别是：

1.状态行 2.响应头(Response Header) 3.响应正文

7. 客户端浏览器解析HTML内容

客户端浏览器读取响应数据HTML，根据HTML的语法对其进行格式化，并在浏览器窗口中显示。解析css，js等文件。

8、浏览器发送请求获取其他在HTML中的资源。



六.axios的工作原理

1. axios 原理还是属于 XMLHttpRequest， 因此需要实现一个ajax。还需要但会一个promise对象来对结果进行处理。

2. Ajax是无需刷新页面就能从服务器取得数据的一种方法。

3. Ajax通过XmlHttpRequest对象来向服务器发异步请求，从服务器获得数据，然后用javascript来操作DOM更新页面。

过程

(1). 创建XMLHttpRequest对象。

(2). 设置响应HTTP请求的回调函数。

(3). 创建一个HTTP请求，指定相应的请求方法、url等。

(4). 发送HTTP请求。

(5). 获取服务器端返回的数据。

(6). 使用JavaScript操作DOM更新页面

var Axios = {

get: function(url) {

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

var xhr = new XMLHttpRequest();

xhr.open('GET', url, true);

xhr.onreadystatechange = function() {

// readyState == 4说明请求已完成

if (xhr.readyState == 4 && xhr.status == 200) {

// 从服务器获得数据

resolve(xhr.responseText)

}

};

xhr.send();

})

},

}

七.axios的基本使用

请求的方式

(0). axios(config)

(1). axios.request(config)

(2). axios.get(url[, config])

(3). axios.delete(url[, config])

(4). axios.head(url[, config])

(5). axios.post(url[, data[, config]])

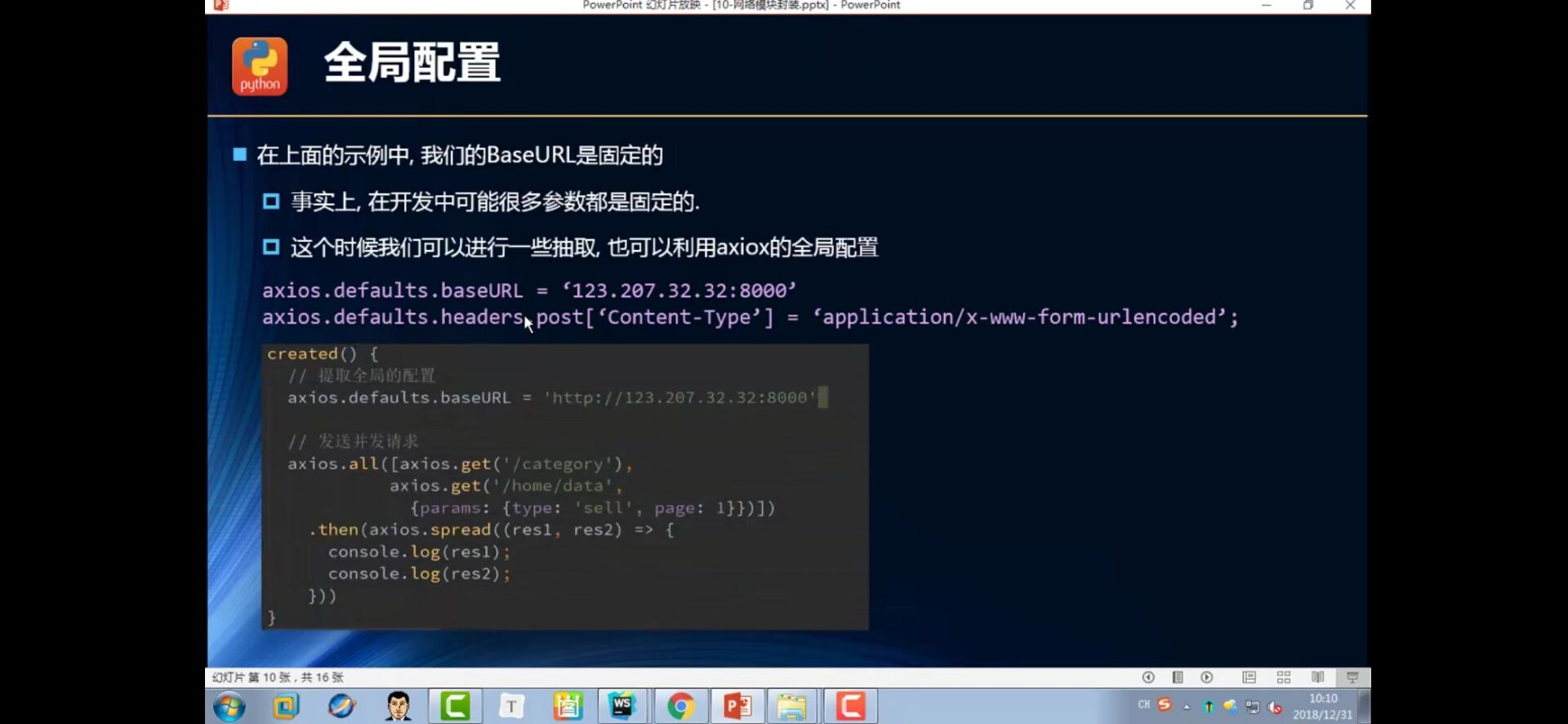
(6). axios.put(url[, data[, config]])

(7). axios.patch(url[, data[, config]])

axios发送并发请求

1. 使用axios.all, 可以放入多个请求的数组
2. axios.all([])返回的结果是一个数组，使用axios.spread可将数组[res1,res2]展开为res1,res2, 如下图



3. axios的全局baseUrl的设置

4. axios的基本配置，默认是get请求，get请求的参数是放在params，如果是post则参数请求体是放在data里的,例如可以创建多个instance1或者intanse2

const instance1 = axiom.create({

baseUrl: ‘http://172.32.32.67:8080’,  
 tiemOut: 1000  
});

instance1({  
 url: ‘/home/multidate’,

params: {  
 type: ‘pop’,

page: 1  
 }  
}).then(res => {  
 console.log(res)  
});

5. 封装一个axios的请求，供调用

export function request(config){  
 const instance = axios.create({  
 baseUrl: ‘http://172.32.32.67:8080’,

tiemOut: 5000  
 });  
 return instance(config)  
}

使用

request({  
 url: ‘/home/mulitdate’  
}).then((res) => {  
 console.log(res)  
}).catch((err) => {  
 console.log(err)  
}).

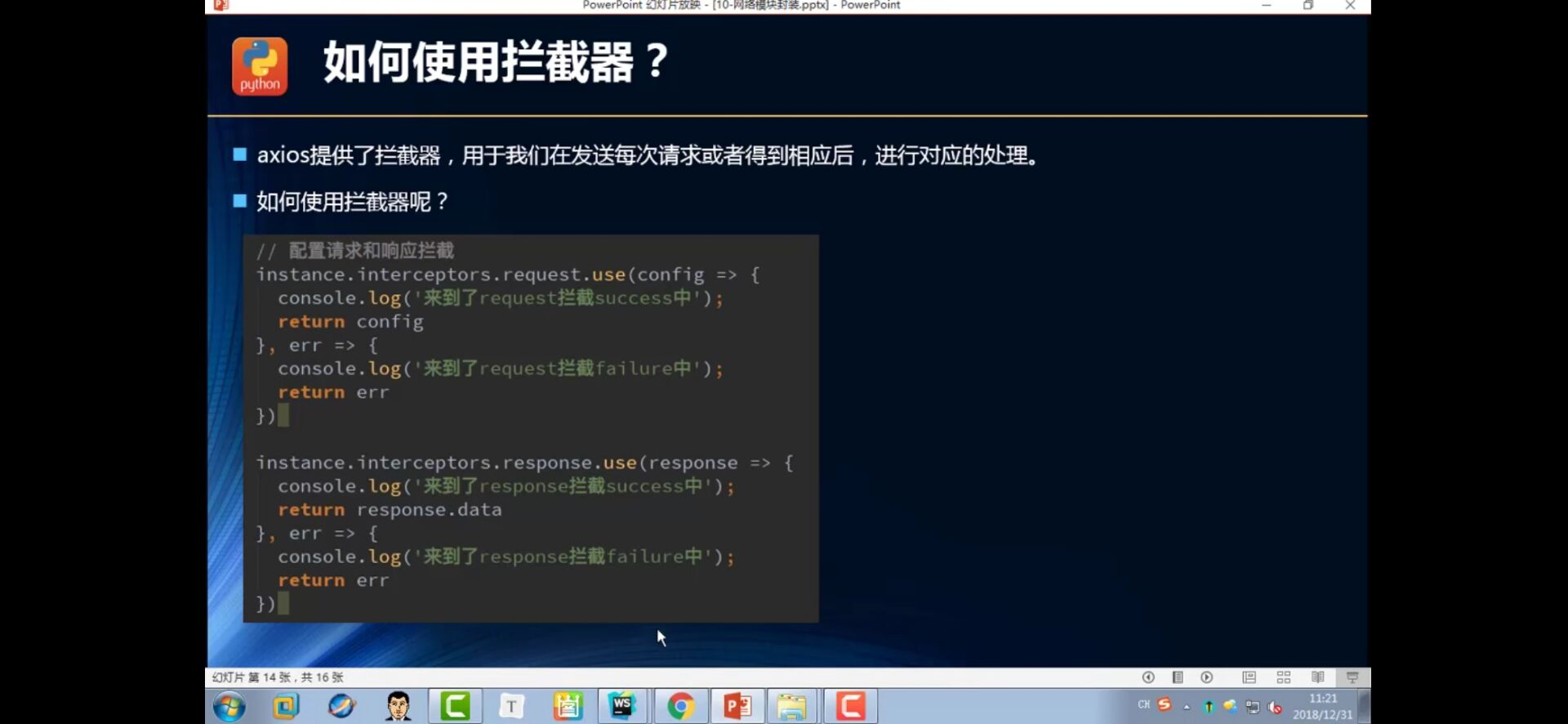
5. axios的拦截器，一般有以下几种情况使用。处理过后一个要return回去

以下为axios.interceptots.request.use请求拦截时会用到的

1.返回config中的信息不符合服务器要求

2.比如每次发送网络请求时，都希望在界面中显示一个在请求中的图标

3.某些网络请求（比如token）必须携带某些特殊的东西。

以下为axios.interceptors.response.use响应拦截

八.浏览器的缓存

浏览器缓存策略分为两种：强缓存和协商缓存，并且缓存策略都是通过设置 HTTP Header 来实现的。

浏览器每次发起请求，都会先在浏览器缓存中查找该请求的结果以及缓存标识。第一次向服务器发起该请求后拿到请求结果后，将请求结果和缓存标识存入浏览器缓存。

(1). 强缓存

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

①Expires: 缓存过期时间，用来指定资源到期的时间，是服务器端的具体的时间点。受限于本地时间，如果修改了本地时间，可能会造成缓存失效。

②Cache-Conctrol: 主要用于控制网页缓存。比如当Cache-Control:max-age=300时，则代表在这个请求正确返回时间（浏览器也会记录下来）的5分钟内再次加载资源，就会命中强缓存。

(2). 协商缓存

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

①协商缓存生效，返回304和Not Modified，该资源无更新。

②协商缓存失效，返回200和请求结果，资源更新了。

九.跨域问题

跨域：请求的接口地址必须与页面url地址处于同域上（即域名，端口，协议相同），不能执行其它网站上的脚本。

同源策略：它用于限制一个origin的文档或者它加载的脚本如何能与另一个源的资源进行交换，能帮助阻隔恶意文档。

同源：两个url的端口，协议，域名都要相同

(1). 解决本地开发接口请求跨域问题的4种方法

1.在node的环境里，使用webpack-dev-server配置跨域方法，也就是在webpack的配置文件中的devSerber:{ proxy:{}}下配置代理路径

2.在node的环境下，通过http-proxy-middleware模块设置代理服务器

3.设置新版Chrome支持本地跨域请求调试

4.nginx反向代理的解决跨域设置

(2). 代理和反向代理

代理：也称正向代理，是指一个位于客户端和目标服务器之间的服务器。为了从目标服务器取得内容，客户端向代理服务器发送一个请求并指定目标服务器，然后代理向目标服务器转交请求并将获得分内容返回给客户端。

反向代理：是指以代理服务器来接收internet上的请求，然后将请求转发给内部网络上的服务器，并将从服务器上得到的结果返回internet上请求的客户端，此时代理服务器就变现为一个反向代理服务器。

(3). 解决跨域的方法

1.前端的jsonp方法：原理就是html中的link，src，href属性不受跨域的影响。script的src属性可以随意的引入不同源的js文件

2.通过修改document.domain来跨子域：前提条件这两个域名必须属于同一个基础域名，而且协议，端口都要一致，否则无法跨域，如下。

常用于处理 iframe 下跨域请求 DOM 资源（如提交表单等），该方式只能用于二级域名相同的情况下，比如 a.test.com 和 b.test.com 适用于该方式。

只需要给页面添加 document.domain = 'test.com' 表示二级域名都相同就可以实现跨域。a.html和b.html

<body>

<h1>Hi, this is A html.</h1>

<iframe

id="frame"

src="http://www.test.com:8001/b.html"

frameborder="0"

onload="load()"

></iframe>

<script>

document.domain = "test.com"; //设置domain

function load() {

let frame = document.getElementById("frame");

console.log(frame.contentWindow.data); // This is b html content.

}

</script>

</body>

b.html的页面

<body>

<h1>Hi, this is B html.</h1>

<script>

document.domain = "test.com"; //设置domain

var data = "This is b html content.";

</script>

</body>

3.通过nginx的反向代理，之前有介绍

proxy\_pass 后面跟着一个 URL，用来将请求反向代理到 URL 参数指定的服务器上。例如我们上面例子中的 proxy\_pass https://api.shanbay.com，则将匹配的请求反向代理到 https://api.shanbay.com。

通过在配置文件中增加proxy\_pass 你的服务器ip,例如这里的扇贝服务器地址，就可以完成反向代理。

server {

listen 80;

server\_name localhost;

## 用户访问 localhost，则反向代理到https://api.shanbay.com

location / {

root html;

index index.html index.htm;

proxy\_pass https://api.shanbay.com;

}

}，更多操作查看百度

4.通过页面嵌套iframe + location. hash的方法

5.通过iframe + window. name的方法

6.通过iframe + window. postMessage方法来跨域。比较狭窄的跨域

7.通过cors解决跨域

cors:跨域资源共享(CORS) 是一种机制，它使用额外的 HTTP 头来告诉浏览器 让运行在一个 origin (domain) 上的 Web 应用被准许访问来自不同源服务器上的指定的资源，主要设置的是：

"Access-Control-Allow-Origin": “\*”,(指定的源如http://. www.protal.fat.qa.pab.com.cn,只对该源生效)

"Access-Control-Allow-Methods": "PUT, POST, GET”(方法)

"Access-Control-Allow-Headers": “X-Custom-Header”

8.nodejs中间件代理跨域

Node + express + http-proxy-middleware搭建proxy服务器

十.jsonp的封装

JSONP的核心在于通过创建<script>标签的src来帮助我们请求数据，因为src不受跨域的限制,我们通过src去服务器请求数据，将数据当作一个javascript的函数来执行，并且在执行中传入我们需要的json，只能实现 GET 跨域请求，具体的封装和调用如下。

const jsonp = ({ url, params, callbackName }) => {

const generateURL = () => {

let dataStr = "";

for (let key in params) {

dataStr += `${key}=${params[key]}&`;

};

dataStr += `callback=${callbackName}`;

return `${url}?${dataStr}`;

};

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

// 初始化回调函数名称

callbackName = callbackName || "cb" + Math.random().toString()

.replace(".", "");

let scriptEle = document.createElement("script");

scriptEle.src = generateURL();

document.body.appendChild(scriptEle);

// 绑定到 window 上，为了后面调用

window[callbackName] = data => {

resolve(data);

// script 执行完了，成为无用元素，需要清除

document.body.removeChild(scriptEle);

};

});

};

调用时

jsonp({

url: "http://127.0.0.1:8000/",

params: {

name: "Yang Min",

age: "8"

},

callbackName: "getData"

}) .then(data => JSON.parse(data)).then(data => {

console.log(data); // {name: "Yang Min", age: "8"}

});

vue的基本原理以及细节

一.vue的优缺点

1. 不利于seo的优化，报错不明显
2. 不支持ie8以及以下版本浏览器，例如不支持Object.defineProperty的属性
3. 缺乏类型的校验
4. 可扩展性差

二.vue核心之虚拟dom

里面涉及包括性能，虚拟dom，diff算法，深度优先遍历以及key等内容，具体的参考：

<https://zoyi14.smartapps.cn/pages/note/index?slug=af0b398602bc&origin=share&hostname=baiduboxapp&_swebfr=1>

三.vue中钩子函数mounted和created的区别。

created: 在模板渲染成html前调用，即通常初始化某些属性值，然后再渲染成视图。

mounted: 在模板渲染成html后调用，通常是初始化页面完成后，再对html的dom节点进行一些需要的操作。

四.vue中computed和watch的区别

computed：在computed中定义的每一个计算属性，都会被缓存起来，如果所依赖的属性没有发生改变，那么调用当前函数的时候会从缓存中读取。只有当计算属性里面依赖的一个或多个属性变化了，才会重新计算当前计算属性的值，在computed中的属性都有一个get和一个set方法，当数据变化时，调用set方法。通过如计算data中的值返回后直接在模版上使用。（多对一）

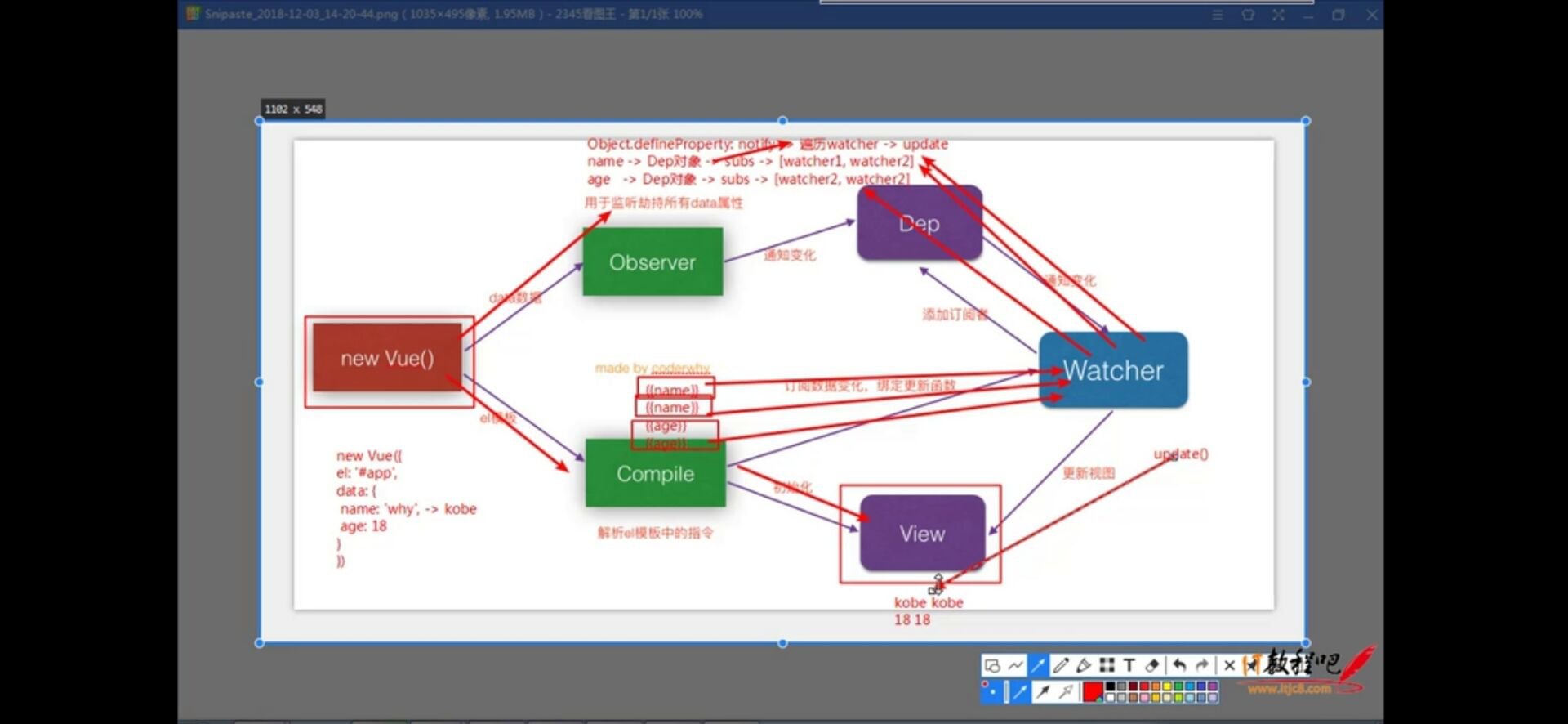
watch：是属性监听器，一般用来监听属性的变化（也可用来监听计算属性函数）并做一些逻辑；每次监听的值发生变化的时候都会执行回调。watch只会监听数据的值是否发生改变，而不会去监听数据的地址是否发生改变。特殊情况就是说更改数组中的数据时，数组已经更改，但是视图没有更新；更改数组必须要用splice()或者$set。用于一个数据来影响多个数据的操作（一对多）。

五.vue双向绑定的原理

双向绑定的原理主要分为两部分：

(1)observe数据劫持(观察者)

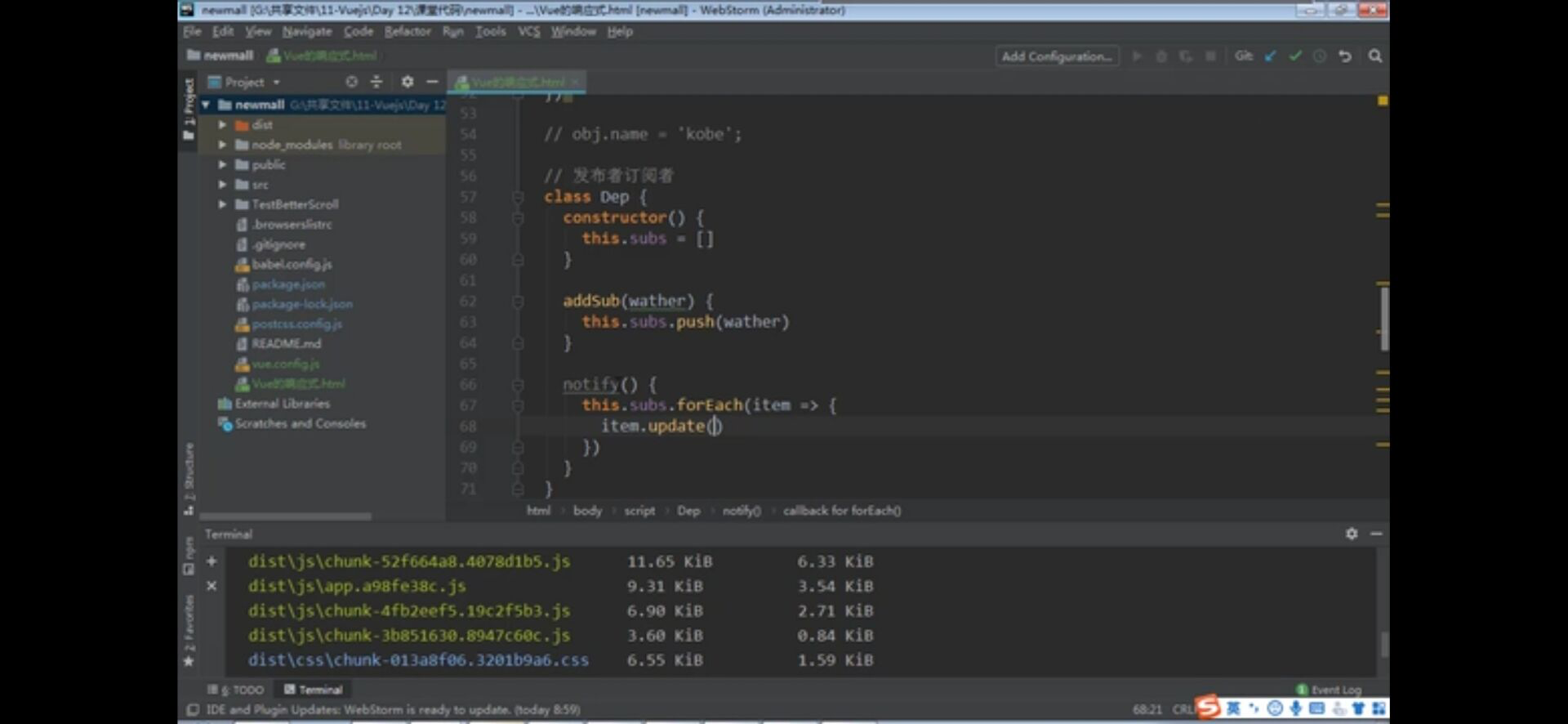
(2)发布订阅者模式

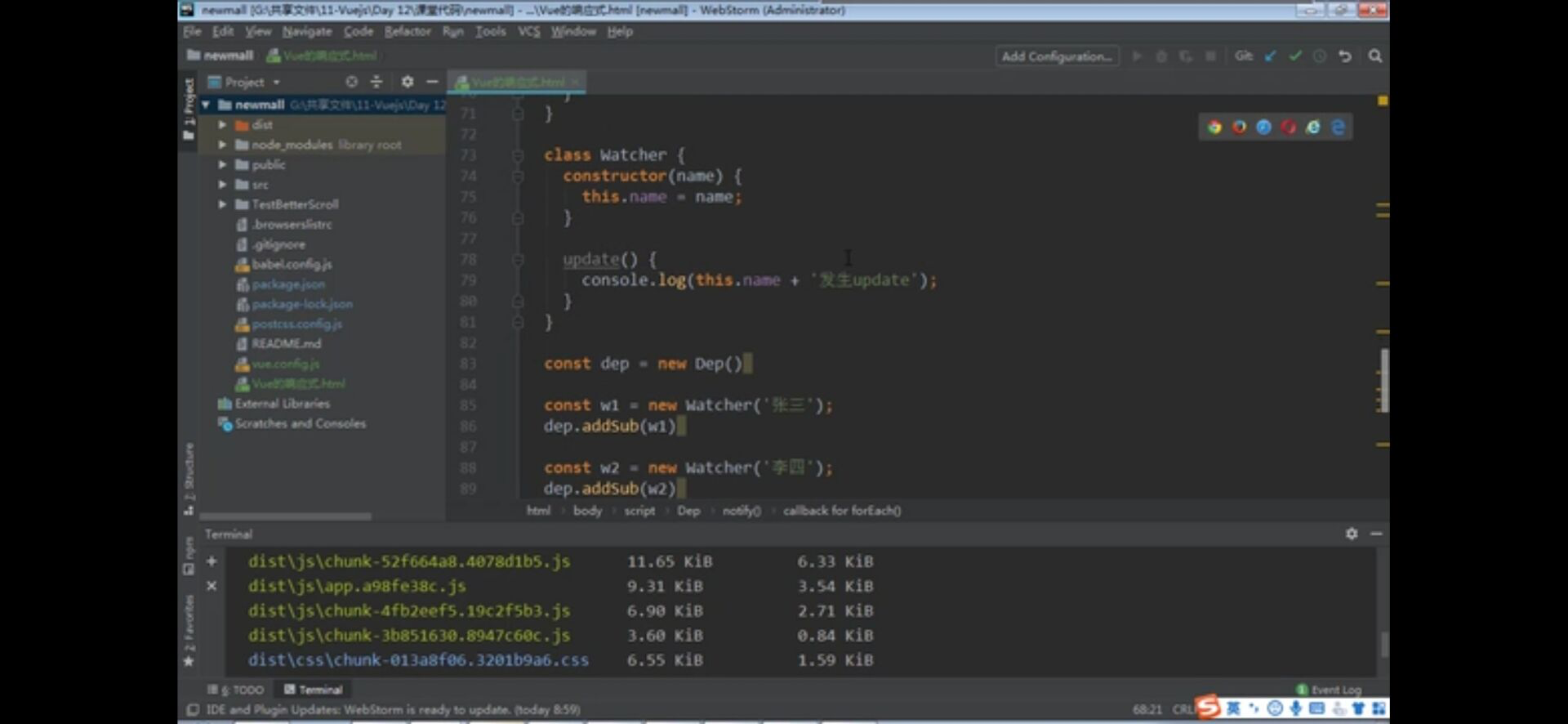
整体的流程图如下：

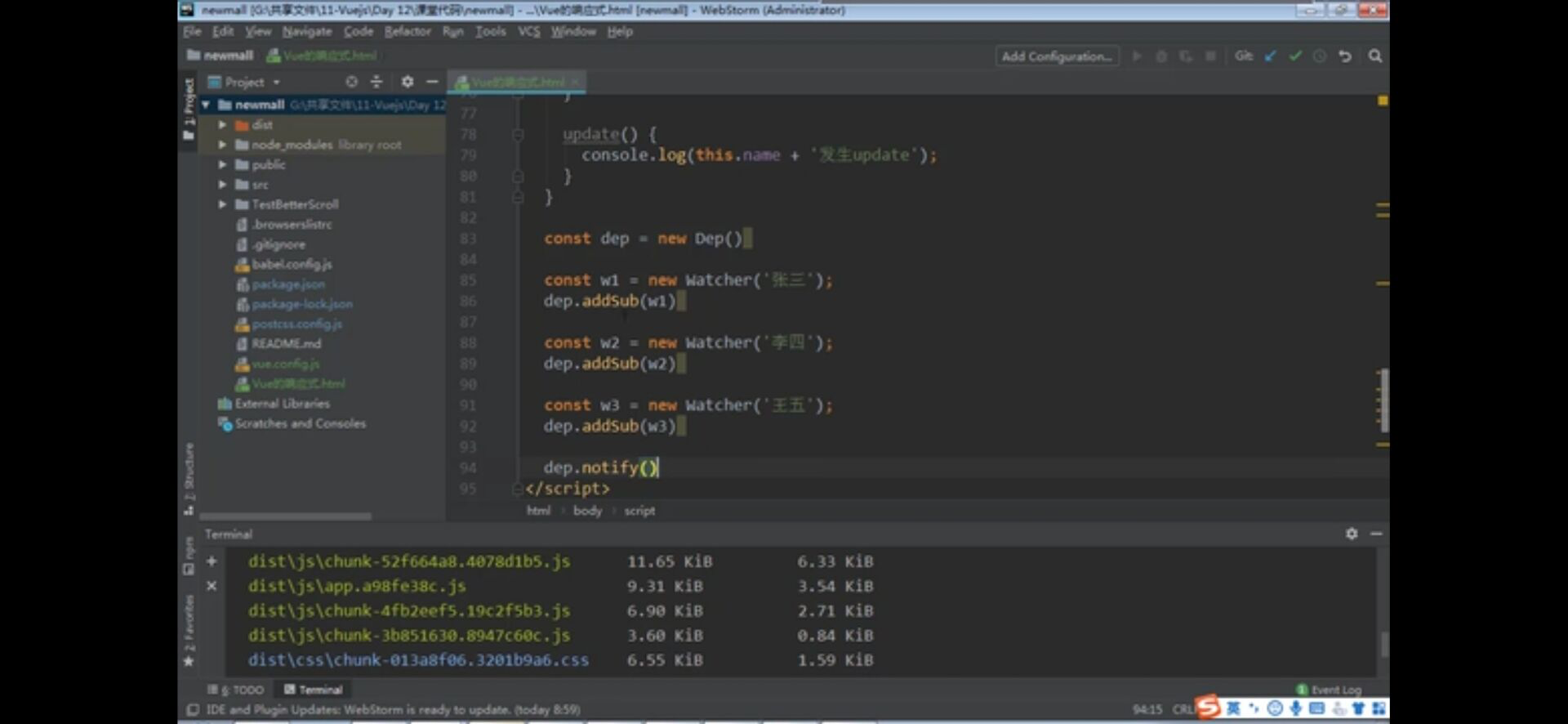
实现过程如下图

数据的劫持以及defineproperty属性，数据发生改变会触发defineproperty里的set函数通过调用notify函数去通知订阅者更新视图(下图的调用dep.notify()函数没有写)。



发布订阅者实现的第一步，定义dep函数，里面主要是对将所有的订阅者加入到数组中，并到时候对订阅者上的数据进行一个更新。

发布订阅者实现的第二步，定义watcher函数，也就是订阅者，指的是html页面上所有订阅了该属性值的节点。里面主要提供一个update更新的方法。到时候用于调用编辑模版内的回调函数，促使页面订阅者视图的更新。

具体的调用情况

六.vue自定义指令的方法

通过vue.directive()去定义。如下面的例子

<script>

//注册一个全局自定义指令 v-focus

Vue.directive(‘focus’, {

//将绑定的元素插入到dom中

inserted: function(el) {

// 聚焦元素  
 el.focus()  
 }

})

//创建根实例

new Vue({  
 el: ‘#app’  
 })

</script>

七.vue的nextTick的由来

由于vue的数据驱动视图更新是异步的，即修改数据的当下，视图不会立即更新，而是等同一事件循环中的所有数据变化完成后，再统一进行视图更新（虚拟dom机制）。

nextTick的触发时机：在同一事件循环中的数据变化后，dom完成更新，立即执行nextTick(callback)内的回调。

八.vue.set()的方法

用于修改data中的数组或者是对象中的属性。

九.vue组件里的data为什么是函数return回来的而根组件的是一个对象

JS中的对象是引用类型的数据，当多个实例引用同一个对象时，只要一个实例对这个对象进行操作，其他实例中的数据也会发生变化。

而在Vue中，我们更多的是想要复用组件，那就需要每个组件都有自己的数据，这样组件之间才不会相互干扰。所以组件的数据不能写成对象的形式，而是要写成函数的形式。返回一个新的data数据以函数返回值的形式定义，这样当我们每次复用组件的时候，就会返回一个新的data，维护自己的数据。

十.vue里的mixins混入的应用

组件在引用之后相当于在父组件内开辟了一块单独的空间，来根据父组件props过来的值进行相应的操作，单本质上两者还是泾渭分明，相对独立。

而mixins则是在引入组件之后，则是将组件内部的内容如data等方法、method等属性与父组件相应内容进行合并。相当于在引入后，父组件的各种属性方法都被扩充了。例如：

新建文件命名为mixins.js

<script>

export default {

name: 'mixins-test-main',

components: {},

props: {},

data () {

return {

mixinData: 'mixin中的变量'

}

},

methods: {

mixinFunction () {

return '我是mixins里面的公共方法'

},

},

mounted () {

}

}

</script>

引入mixins.js文件

<script>

import MixinItem from './mixins'

export default {

name: 'mixin-test-comp',

props: {},

mixins: [MixinItem],

components: {},

data () {

return {}

},

methods: {

},

mounted () {

},

computed: {}

}

</script>

十一. vue父组件传给子组件的props的使用，如果要修改这个props怎么去操作

父组件传给子组件props的使用，如果要修改这个props的解决方法，可以先在date上定义，然后赋予该值，为了监听父组件传过来的值是否变化，所以还要监听，并更新。如下面的例子：

子组件的部分代码

props:{  
 tempted: {  
 type: String  
 };

depByorgList: {  
 type: Arry  
 }  
}，

data:{  
 return {  
 dialog: false,

thedepByorgList: this.depByorgList,

deform: {}  
 }  
}

使用data一个变量thedepByorgList去接收，子组件就可以满足对父组件传过来的值进行操作了。但是这样会有一个问题。thedepByorgList这个属性只会在组件创建的时候赋一次值，后续如果depByorgList有变化，thedepByorgList并不会自动同步，这时候就需要引用watch的属性监听函数。对depByorgList进行监听动态改变关联的状态。

Watch: {  
 depByorgList: function(val) {  
 this.thedepByorgList = val;  
 }   
}

十二 .vue-router的工作原理

vue-router通过hash与History interface两种方式实现前端路由，更新视图但不重新请求页面”是前端路由原理的核心之一，window.location.href。目前在浏览器环境中这一功能的实现主要有两种方式：

1、Hash模式：

hash（#）是URL 的锚点，代表的是网页中的一个位置，单单改变#后的部分，浏览器只会滚动到相应位置，不会重新加载网页，也就是说 #是用来指导浏览器动作的，对服务器端完全无用，HTTP请求中也不会不包括#；同时每一次改变#后的部分，都会在浏览器的访问历史中增加一个记录，使用”后退”按钮，就可以回到上一个位置；HashHistory.push()将新路由添加到浏览器访问历史的栈顶。HashHistory.replace()替换掉当前的路由。

2、History模式：

HTML5 History API提供了一种功能，能让开发人员在不刷新整个页面的情况下修改站点的URL，就是利用 history.pushState API 来完成 URL 跳转而无须重新加载页面；通过back(), forward(), go()等方法可以读取浏览器历史记录栈的信息进行各种跳转操作，history. pushState()，history.replaceState()。与hash的push和replace方式相似。

pushState设置的新URL可以是与当前URL同源的任意URL；而hash只可修改#后面的部分，故只可设置与当前同文档的URL

十三.vue-router的导航守卫

全局守卫

const router = new VueRouter({

routes: [

{

path: '/foo',

component: Foo,

children: [ //路由嵌套路由在children里写

{

path: 'bar',

component: Bar,

meta: { requiresAuth: true }

}

]

}

]

})

router.beforeEach((to, from, next) => {

// …用户权限的验证、登录 next()必须要写才能执行下一步

console.log(‘a’)

next()

});

守卫方法接收三个参数:

to: Route: 即将要进入的目标 路由对象。

from: Route: 当前导航正要离开的路由。

next: Function: 一定要调用该方法来 resolve 这个钩子。执行效果依赖 next 方法的调用参数。 next(): 进行管道中的下一个钩子。

全局后置钩子

router.afterEach((to, from) => {

// ...

};

你也可以注册全局后置钩子，然而和守卫不同的是，这些钩子不会接受 next 函数也不会改变导航本身。

路由独享的守卫

你可以在路由配置上直接定义 beforeEnter 守卫

const router = new VueRouter({

routes: [

{

path: '/foo',

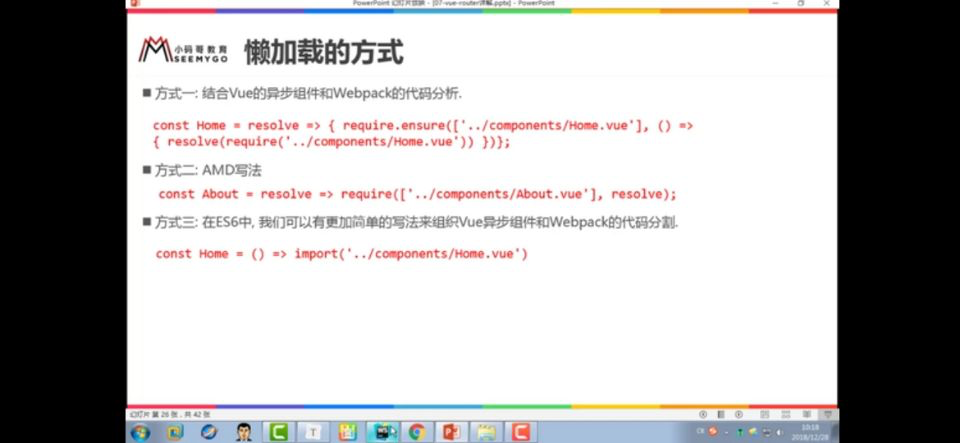
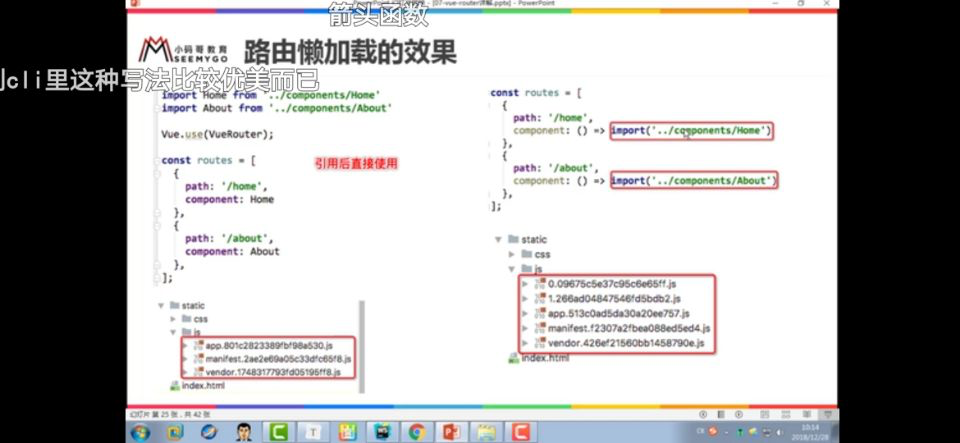
component: Foo,

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

}]

})

十四.路由的懒加载以及传参

vue路由的懒加载：即使用到对应的js文件时才去对应的服务器上拿取资源，避免一下子加载过多影响网页的加载，使用import如下图，加载方式

路由的传参：有两种方法

1. 一种是通过动态路由，如：

this.$router.push({

path: ‘/profile’,

query: {

name: ‘kobe’,

age: 22

}

})

2. 一种是通过query

<router-link :to=“{path: ‘/profile’, query: {name: ‘kobe’, age: 22}}”

十五.activated钩子函数与keep-alive的关系以及exclude和exclude

activated()和deactivated()这两个函数，只有组件被保持的状态使用了keep-alive时，才是有效的。keep-alive用于保存组件的渲染状态。还可以避免组件反复创建和渲染，有效提升系统性能。

在动态组件中的应用

<keep-alive :include="whiteList" :exclude="blackList" :max="amount">

<component :is="currentComponent"></component>

</keep-alive>

在vue-router中的应用

<keep-alive :include="whiteList" :exclude="blackList" :max="amount">

<router-view></router-view>

</keep-alive>

include定义缓存白名单，keep-alive会缓存命中的组件；exclude定义缓存黑名单，被命中的组件将不会被缓存；max定义缓存组件上限，超出上限使用LRU的策略置换缓存数据。

十六.vue组件中name的作用

1.递归组件（递归组件是指组件自身调用自身）时会用到

2.取消缓存（exclude）时会用到

我们有个组件命名为detail,其中dom加载完毕后我们在钩子函数mounted中进行数据加载。

export default {

name:'Detail'

}，

mounted(){

this.getInfo();

}，

methods:{

getInfo(){

axios.get('/xx/detail.json',{

params:{

id:this.$route.params.id

}

}).then(this.getInfoSucc)

}

}

因为我们在App.vue中使用了keep-alive导致我们第二次进入的时候页面不会重新请求，即触发mounted函数。有两个解决方案,一个增加activated()函数,每次进入新页面的时候再获取一次数据。

还有个方案就是在keep-alive中增加一个过滤，如下图所示：

<div id="app">

<keep-alive exclude="Detail">

<router-view/>

</keep-alive>

</div>

3.使用vue-devtools时会用到

vue-devtools调试工具里显示的组见名称是由vue中组件name决定的

十七.vuex使用mapstate以及module和namespaced的关联以及父子组件访问的方法。

vuex 使用mapstate mapactions mapmutations mapgetters与modules中包涵多模块的关系 。

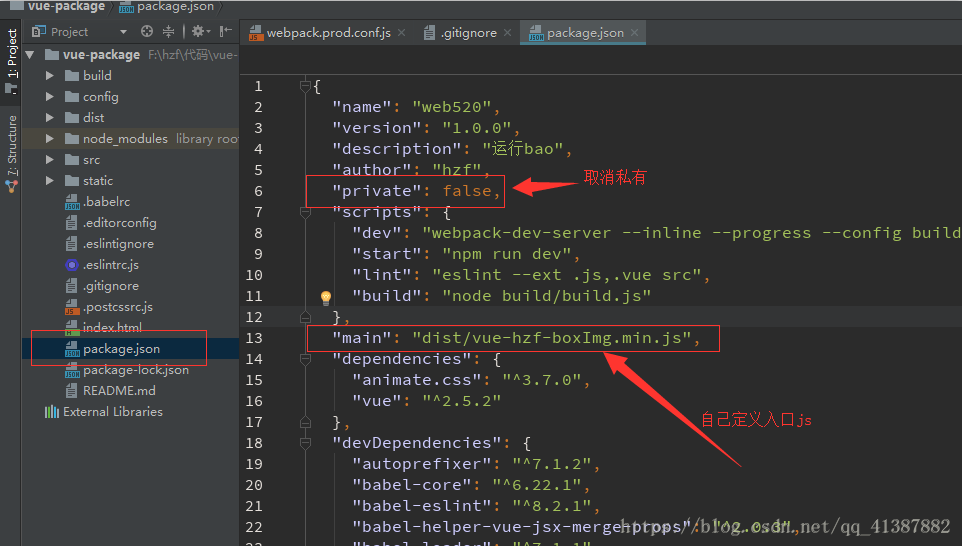
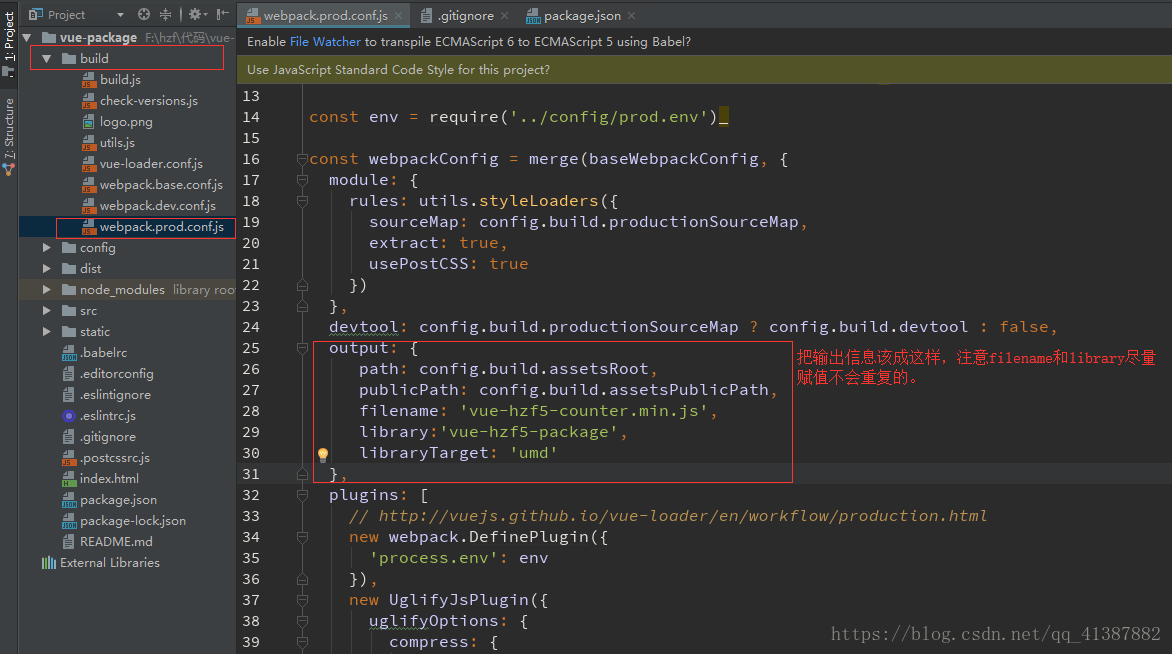
详细连接查看：

https://blog.csdn.net/lzb348110175/article/details/89387495

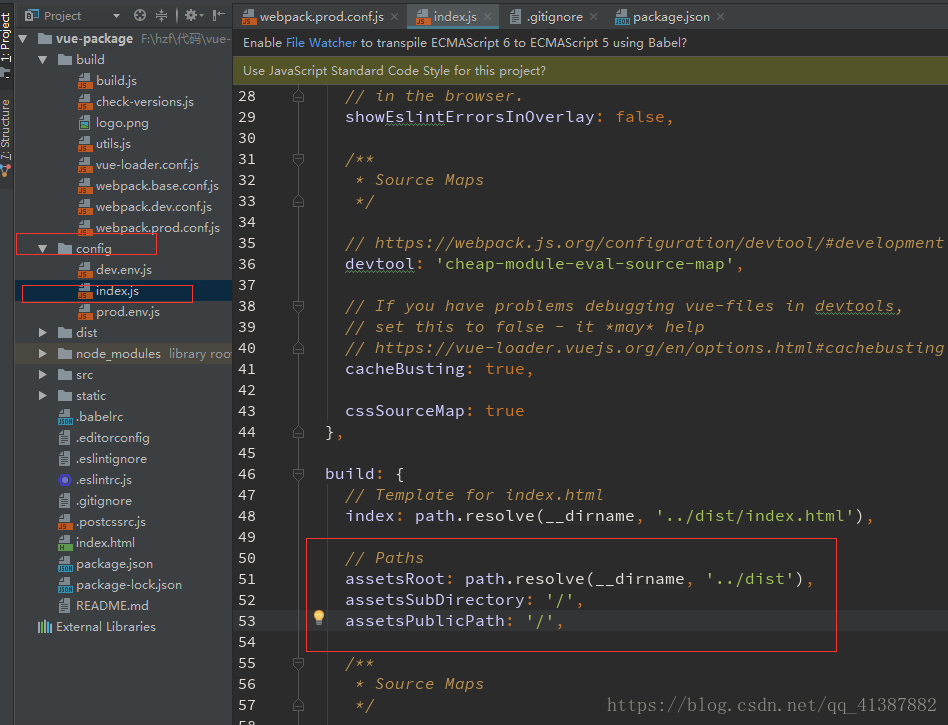
vue父组件访问子组件除了通过$emit去绑定。还可以通过this. $children这是个数组，子访问父组件一般用this.$parent(一般在开发中不建议使用)访问根组件用this. $root。

十八.Vue项目打包组件并上传到npm在其他项目和下载引用 步骤流程

1. 修改你项目主目录下的package.json文件在文件添加一个程序的入口配置 如图：还有记得修改private 为 false

2.修改项目目录 -》bulid -》webpack.prod.conf.js文件 如图

3.第三步。修改 config -》index.js 文件 如图

第4步。修改main.js 在此文件中添加你的功能组件并export导出

/\* 封装的功能组件 \*/

import VueHzf5Package from './components/divImgBox.vue'

/\* 导出 \*/

export default VueHzf5Package

第5步.现在所有配置修改该的地方已经差不多，那么我们可以进行打包了。命令： npm run build

6.最后一步.发布到npm上。首先先登录并邮箱验证：命令 npm login 回车。

登录成功后就可以上传你的组件了 命令：npm publish ；

十九.Vue 的父组件和子组件生命周期钩子执行顺序

1.渲染过程：父组件挂载完成一定是等子组件都挂载完成后，才算是父组件挂载完，所以父组件的mounted在子组件mouted之后。父beforeCreate -> 父created -> 父beforeMount -> 子beforeCreate -> 子created -> 子beforeMount -> 子mounted -> 父mounted。

2.子组件更新过程：父 beforeUpdate -> 子 beforeUpdate -> 子 updated -> 父 updated。

3.父组件更新过程：父 beforeUpdate -> 父 updated

4.销毁过程：父 beforeDestroy -> 子 beforeDestroy -> 子 destroyed -> 父 destroyed。

二十.vuex中mutation和action的详细区别

1、流程顺序

“相应视图—>修改State”拆分成两部分，视图触发Action，Action再触发Mutation。

2、角色定位

基于流程顺序，二者扮演不同的角色。

Mutation：专注于修改State，理论上是修改State的唯一途径。

Action：业务代码、异步请求。

3、限制

角色不同，二者有不同的限制。

Mutation：必须同步执行。

Action：可以异步，但不能直接操作State。

二十一.vue3.0和2.0的区别。

一、vue3.0 的发布与 vue2.0 相比，优势主要体现在：更快、更小、更易维护、更易于原生。

更快

　　1、virtual DOM 完全重写，mounting & patching 提速 100%；

　　2、更多编译时 （compile-time）提醒以减少 runtime 开销；

　　3、基于 Proxy 观察者机制以满足全语言覆盖以及更好的性能；

　　4、放弃 Object.defineProperty ，使用更快的原生 Proxy；

　　5、组件实例初始化速度提高 100%;

　　6、提速一倍/内存使用降低一半；

更小

　　1、Tree-shaking 更友好；

　　2、新的 core runtime：~ 10kb gzipped；

二、新加入了 TypeScript 以及 PWA 的支持

三、默认进行懒观察（lazy observation）

四、更精准的变更通知。

二十二.md5加密算法在vue中的用法。

1.安装。 npm install js-md5 -D

2.项目中引入md5模块。- import md5 from “js-md5”

3. 使用：md5(“加密内容 “)

Vue.prototype.$ md5=md5 使用：this.$md5(” 加密内容”)

二十三.vue3.0新增的api。

（一）setup

setup执行时机: 在beforeCreated和created之间。

setup的特点：

1. 由于在执行setup函数的时候，还没有执行created生命周期方法，所有在setup函数中，是无法使用data和methods的。
2. 由于我们不能在setup函数中使用data和methods，所以vue为了避免我们错误的使用，它直接将setup函数中this修改成了undefined。
3. setup函数只能是同步的不能异步的。

（二）ref的使用例子

1.ref和reactive一样是用来实现响应式数据的方法，由于reactuve只接受对象，所以使用ref实现简单值的监听。ref的底层本质还是reactive。

2.ref(xx)的实质就是reactive({value: xx})，在js中使用ref的值必须通过value获取。

3.注意点：如果是通过ref创建的数据，那么在template中使用的时候不用通过.value来获取，因为vue会自动给我们添加.value。

Import {ref} from ‘vue’

export default {  
 name: ‘App’,

//setup函数是组合api的入口函数

setup(){  
 //定义一个名称叫count变量，这个变量的初始值是0

//这个变量的发生改变后，vue会自动更新ui

let count. = ref(0); //ref用来定义初始值

//在组合API中，如果想定义方法，不用定义到methods //中，直接定义即可

function myFun() {  
 count.value += 1;  
 }

//注意点：在组合API中定义的变量、方法，要想在外界使//用，要通过return {} 暴露出去

return {count, myFun}  
 }  
}

（三）**reactive**的使用例子

1.reactive是vue3提供的实现响应式数据的方法。

在vue2中响应式实现的方法是通过defineProperty来实现的，而在vue3中响应式数据是通过es6的proxy来实现的。

2.reactive的参数必须是对象(json/arr)。如果是其他对象：如Data，则默认情况修改对象是不会自动更新视图的，如果想更新，需要重新赋值。

使用例子

Import {reactive} from ‘vue’

export default {  
 name: ‘App’,

setup(){  
 let state = reactive({  
 stus: [  
 {id: ‘1’, name: ‘as’, age: ‘’10},

{id: ‘2’, name: ‘ls’, age: ‘20’},  
 {id: ‘3’, name: ‘ww’, age: ‘30’}  
 ]  
 });

function rmStu(index) {  
 state.stus = state.stus.filter((stu, idx) => {  
 return idx !== index  
 })  
 };

//let {state, rmStu} = useRemoveStudent(); //可以将上面的抽离出来

return {state, rmStu}  
 }  
}

（四）**ref**和**reactive**的区别

1.如果在template里使用的是ref类型的数据，那么vue会自动帮我们添加.value。

如果在template里使用的是reactive类型的数据，那么vue不会自动帮我们添加.value。

2.vue是如何决定是否需要自动添加.value？

vue在解析数据之前，会自动判断这个数据是否为ref类型，如果是自动添加.value，如果不是则不添加。

3.vue是如何判断当前的数据是否为ref类型的？

通过当前数据的\_\_v\_\_ref来判断的，如果有这个私有的属性，并且取值为true，那么就是一个ref类型的数据。

4.isRef和isReactive是用来判断ref和reactive的类型的。

（四）**shallowReactive**和**shallowRef**非递归监听

ref和reactive进行的是递归监听，shallowReactive和shallowRef是非递归监听，shallowReactive只监听第一层数据的变化，相比递归监听来说节省性能。只包装一个proxy。

如果是通过shallowRef创建的数据，那么vue监听的是.value的变化，并不是第一层的变化。