目　　录

珠峰培训 JavaScript 面试宝典..... 1
1　原生 JS 部分.....2
2　ajax&http.....19
3　vue 相关.....27
4　react 相关..... 39
5 项目相关.....50

珠峰培训前端面试题精选

原生 JS

1.ES6 的新特性：

let(声明变量)

const(声明常量,常量不能修改的量)

var、let、const 的区别

1. let 和 const 声明变量不存在变量提升，如果要使用这个变量，我们需要在变量定义之后使用;

2. let 和 const 不能重复声明变量，如果重复声明会报错;

3. 用 let 和 const 在全局声明变量不会给 window 增加属性;

4. let 和 const 出现在代码块中，会把代码块(字面量声明对象除外)变成块级作用域,并且出现暂时 性死区 class(创建类)

import/export(基于 ES6 的模块规范创建导入/导出模块(文件/组件))

new set(数组去重)

Symbol(唯一的值) var a = Symbol('qqq')

...ary(展开运算符、剩余运算符)

${} 模板字符串

解构赋值 let {a} = obj; let [b] = ary

for of 循环

()=>{} 箭头函数

箭头函数与普通函数的区别：

1. 箭头函数是匿名函数，不能作为构造函数，不能使用 new

2. 箭头函数没有原型属性

3.this 指向不同，箭头函数的 this 是定义时所在的对象，普通函数看前面有没有.,点前面是谁 this 就是谁,没有.就是 window

4. 不可以使用 arguments 对象，该对象在函数体内不存在。

数组新增方法：flat、find、findIndex

对象新增方法：

Object.assign() Object.values() Object.keys() Object.create()...

2.JS 的数据类型

基本数据类型：number 数字； boolean 布尔值 :有两个值 true、false ；string 字符串

null 空对象； undefined 未定义的值(很多浏览器的初始值是 undefined)

Symbol() 产生一个唯一的值,和谁都不重复

null 和 undefined 的区别：

null 是一个表示"无"的对象，转为数值时为 0

undefined 是一个表示"无"的原始值，转为数值时为 NaN

当声明的变量还未被初始化时，变量的默认值为 undefined

null 用来表示尚未存在的对象，常用来表示函数企图返回一个不存在的对象

undefined 表示 “缺少值”，就是此处应该有一个值，但是还没有定义。

典型用法是：

1. 变量被声明了，但没有赋值时，就等于 undefined
2. 调用函数时，应该提供的参数没有提供，该参数等于 undefined
3. 对象没有赋值的属性，该属性的值为 undefined
4. 函数没有返回值时，默认返回 undefined

null 表示“没有对象”，即该处不应该有值。

典型用法是：

1. 作为函数的参数，表示该函数的参数不是对象
2. 作为对象原型链的终点

引用数据类型：

对象

.普通对象

.数组对象

.正则对象(匹配字符串的规则)

.日期对象

.函数对象

...

对象的存储过程：

1. 开辟一个空间地址
2. 把键值对存储到这个空间地址的堆内存中
3. 把这个对象指针赋值给变量名

```
let obj = {
  a:1,
  fn:(function (val) {
    // 赋给 fn 的是自执行函数的执行结果 也就是一个 undefined
    // 该自执行函数只会执行一次
    console.log(val);
```

```
  })(obj.a)
};
let obj2 = obj;// 两者代表了同一个地址;
// 获取属性的值 obj.fn 或者 obj['fn']
// 新增属性： obj.c = 100 或者 obj['c'] = 100
// 真删除 delete obj.a (在严格模式下不支持该方法)； 假删除： obj.a = null;

// 引用类型小习题
let a = 3;
let b = new Number(3);
let c = 3;
console.log(a == b);
console.log(a === b);
console.log(b === c);
//========================
const a = {};
const b = { key: "b" };
const c = { key: "c" };
a[b] = 123;
a[c] = 456;
console.log(a[b]);
```

基本数据类型与引用数据类型的区别：

基本数据类型是操作值,引用数据类型操作的是堆内存空间地址

```
布尔值转换:0 NaN'' null undefined 转化成布尔值是 false，其余的都是 true
检验有效数字的方法：isNaN
常用的数据类型检测方式: typeof constructor instanceof
Object.prototype.toString.call()
```

比较运算符：

== 相对比较：会进行默认的类型转化； 若转换之后的值相等，则结果就是 true

=== 绝对比较，值不但要相同、类型也得相同。

引用数据类型之间的比较，就看是不是同一个地址；

逻辑运算符：

|| 表示或者,前边成立给前边，前边不成立给后边

&& 表示并且前边成立给后边，前边不成立给前边

3.定义函数的方法

- 1.function 声明

```
//ES5
function getSum(){}
function (){}//匿名函数
//ES6
()=>{}
```

- 2.函数表达式

```
//ES5
var getSum=function(){}
//ES6
let getSum=()=>{}
```

- 3.构造函数

```
const getSum = new Function('a', 'b' , 'return a + b')
```

4.JS 作用域的理解

JS 中的作用域分为两种：
全局作用域和函数作用域。
函数作用域中定义的变量，只能在函数中调用，外界无法访问。
没有块级作用域导致了 if 或 for 这样的逻辑语句中定义的变量可以被外界访问，
因此 ES6 中新增了 let 和 const 命令来进行块级作用域的声明。

```
  //循环绑定的问题
  for (var i = 0; i < 3; i++) {
    setTimeout(() => console.log(i), 1);
  }
  for (let i = 0; i < 3; i++) {
    setTimeout(() => console.log(i), 1);
  }

//作用域链  变量的查找机制
// 上级作用域  函数在哪里定义的，那么该函数执行形成的作用的上级作用域就是谁
// 了解了上级作用域，  就比较容易查找变量对应的值
```

5.闭包的理解

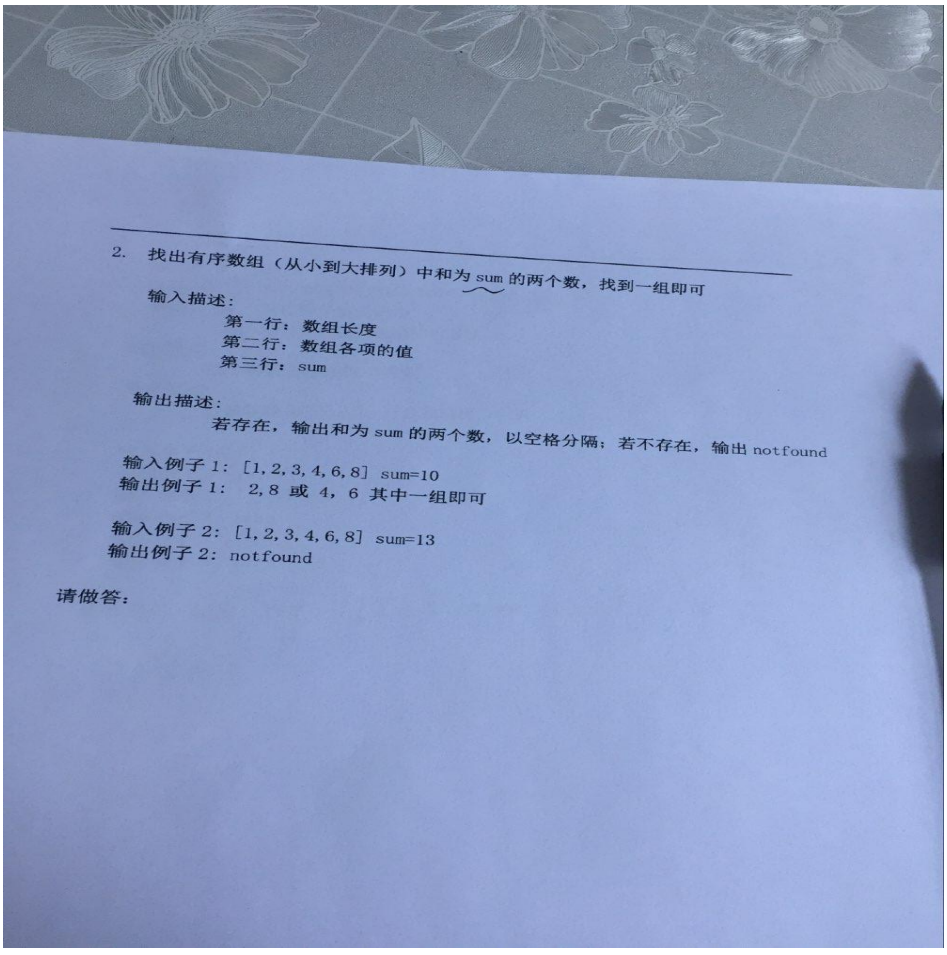
简单来说闭包就是在函数里面声明函数，本质上说就是在函数内部和函数外部搭建起一座桥梁，使得子函数可以访问父函数中所有的局部变量，但是反之不可以，这只是闭包的作用之一，另一个作用，则是保护变量不受外界污染，使其一直存在内存中，在工作中我们还是少使用闭包的好，因为闭包太消耗内存，不到万不得已的时候尽量不使用。

6.数组

```
// 数组去重
1、双 for 循环去重
2、利用对象的属性名不能重复去重
3、利用 es6 的 Set 不能重复去重
（具体代码自己查）
```

```
// 数组重组  （将name值相同的合并，并去除age的属性）
let ary = [
    {name:1,age:2,number:1,son:'son1'},
    {name:2,age:23,number:2,son:'son2'},
    {name:2,age:22,number:3,son:'son3'},
    {name:1,age:12,number:4,son:'son4'},
    {name:1,age:42,number:5,son:'son5'}
]
fn(ary) // 结果为
[
  {
    "name":1,
    "list":[{"number":1,"son":"son1"},{"number":4,"son":"son4"},{"number":
  },
  {
    "name":2,
    "list":[{"number":2,"son":"son2"},{"number":3,"son":"son3"}]
  }
]
function fn(ary){
    let arr = [];
    ary.forEach(item=>{
        let bol = arr.some(val=>{
            if(val.name===item.name){
                let obj = {};
                Object.keys(item).forEach(v=>{
                    if(v!='name'&&v!='age'){
                        obj[v] = item[v]
                    }
                })
                val.list.push(obj);
                return true
            }
        })
        if(!bol){
            let obj = {};
            Object.keys(item).forEach(v=>{
                if(v!='name'&&v!='age'){
                    obj[v] = item[v]
                }
            })
            arr.push({name:item.name,list:[obj]});
        }
    })
    return arr;
}
fn(ary)
```

```
49
50 // 数组扁平化
51 var arr = [
52   [1, 2, 2],
53   [3, 4, 5, 5],
54   [6, 7, 8, 9, [11, 12, [12, 13, [14]]]], 10
55 ];
56
57 function flat1(arr) {
58   let temp = [];
59   function fn(ary) {
60     ary.forEach(item => {
61       if (typeof item == 'object') {
62         fn(item)
63       } else {
64         temp.push(item)
65       }
66     })
67   }
68   fn(arr)
69   return temp;
70 }
71
72 function flat2() {
73   return [].concat(...this.map(item => (Array.isArray(item) ? item.flat()
74 }
```



2. 找出有序数组（从小到大排列）中和为 sum 的两个数，找到一组即可

输入描述：
第一行：数组长度
第二行：数组各项的值
第三行：sum

输出描述：
若存在，输出和为 sum 的两个数，以空格分隔；若不存在，输出 notfound

输入例子 1：[1, 2, 3, 4, 6, 8] sum=10
输出例子 1： 2, 8 或 4, 6 其中一组即可

输入例子 2：[1, 2, 3, 4, 6, 8] sum=13
输出例子 2：notfound

请做答：

7.原型及原型链

原型

- 函数都带有一个 prototype 属性，这是属性是指向构造函数的原型对象，这个对象包含所有实例共享的属性和方法。

- 原型对象都有一个 constructor 属性，这个属性指向所关联的构造函数。

- 每个对象都有一个__proto__ 属性[非标准的方法]，这个属性指向构造函数的原型 prototype

原型链

- 当访问实例对象的某个属性时，会先在这个对象本身的属性上查找，如果没有找到，则会 通过 __proto__ 属性去原型上查找，如果还没有 找到则会在构造函数的原型的__ proto__中查 找， 这样一层层向上查找就会形成一个作用域链，称为原型链

原型相关习题

```
// 第一题
function Fn() {
    this.x = 100;
    this.y = 200;
    this.getX = function () {
        console.log(this.x);
    }
}
Fn.prototype = {
    y: 400,
    getX: function () {
        console.log(this.x);
    },
    getY: function () {
        console.log(this.y);
    },
    sum: function () {
        console.log(this.x + this.y);
    }
};
var f1 = new Fn();
var f2 = new Fn;
console.log(f1.getX === f2.getX);
console.log(f1.getY === f2.getY);
console.log(f1.__proto__.getY === Fn.prototype.getY);
console.log(f1.__proto__.getX === f2.getX);
console.log(f1.getX === Fn.prototype.getX);
console.log(f1.constructor);
console.log(Fn.prototype.__proto__.constructor);
f1.getX();
f1.__proto__.getX();
f2.getY();
Fn.prototype.getY();
f1.sum();
Fn.prototype.sum();
```

```
// 第二题
function Foo() {
    getName = function () {console.log(1);};
    return this;
}
Foo.getName = function () {console.log(2);};
Foo.prototype.getName = function () {console.log(3);};
var getName = function () {console.log(4);};
function getName() {console.log(5);}

Foo.getName();
getName();
Foo().getName();
getName();
var a = new Foo.getName(); //
var b = new Foo().getName();
var c = new new Foo().getName();
console.log(a,b,c);

// 第三题
function Person() {
    this.name = 'zhufeng'
};
Person.prototype.getName = function () {
    console.log(this.name)
    console.log(this.age)
};
Person.prototype.age = 5000;

var per1 = new Person;
per1.getName();
per1.age = 9;
per1.getName();
console.log(per1.age);
var per2 = new Person;
console.log(per2.age);
```

Object.create 的作用：

```
let obj = {a:123};
let o = Object.create(obj);
//该函数返回了一个新的空对象，但是该空对象的__proto__是指向了 obj 这个参数
// 手写 Object.create
function create(proto) {
  function F() {}
  F.prototype = proto;

  return new F();
}
```

new 的执行过程是怎么回事？

new 操作符做了这些事：

- 它创建了一个全新的对象
- 它会被执行[[Prototype]]（也就是__proto__）链接
- 它使 this 指向新创建的对象
- 通过 new 创建的每个对象将最终被[[Prototype]]链接到这个函数的 prototype 对象上
- 如果函数没有返回对象类型 Object(包含 Functoin, Array, Date, RegExg, Error)，那么 new 表达式中的函数调用将返回该对象引用

```
//模拟 new
function objectFactory() {
  const obj = new Object();
  const Constructor = [].shift.call(arguments);

  obj.__proto__ = Constructor.prototype;

  const ret = Constructor.apply(obj, arguments);

  return typeof ret === "object" ? ret : obj;
}
```

call,apply,bind 三者的区别？

apply() 方法调用一个函数，其具有一个指定的 this 值，以及作为一个数组（或类似数组的对象）提供的参数 fun.apply(thisArg, [argsArray])

apply 和 call 基本类似，他们的区别只是传入的参数不同。
apply 和 call 的区别是 call 方法接受的是若干个参数列表，而 apply 接收的是一个包含多个参数的数组。

bind()方法创建一个新的函数，当被调用时，将其 this 关键字设置为提供的值，在调用新函数时，在任何提供之前提供一个给定的参数序列。

call 做了什么:

```
  将函数设为对象的属性
  执行&删除这个函数
  指定 this 到函数并传入给定参数执行函数
  如果不传入参数，默认指向为 window

//实现一个 call 方法：
```

```
Function.prototype.myCall = function(context) {
    //此处没有考虑 context 非 object 情况
    context.fn = this;
    let args = [];
    for (let i = 1, len = arguments.length; i < len; i++) {
        args.push(arguments[i]);
    }
    context.fn(...args);
    let result = context.fn(...args);
    delete context.fn;
    return result;
};

/ 模拟 apply
Function.prototype.myapply = function(context, arr) {
    var context = Object(context) || window;
    context.fn = this;

    var result;
    if (!arr) {
        result = context.fn();
    } else {
        var args = [];
        for (var i = 0, len = arr.length; i < len; i++) {
            args.push("arr[" + i + "]");
        }
        result = eval("context.fn(" + args + ")");
    }

    delete context.fn;
    return result;
};
```

实现 bind 要做什么
返回一个函数，绑定 this，传递预置参数
bind 返回的函数可以作为构造函数使用。故作为构造函数时应使得 this 失效，但是传入的参数依然有效

```
// mdn 的实现
if (!Function.prototype.bind) {
    Function.prototype.bind = function(oThis) {
        if (typeof this !== 'function') {
            // closest thing possible to the ECMAScript 5
```

```
        // internal IsCallable function
        throw new TypeError('Function.prototype.bind - what is trying to be bound is not callable');
    }

    var aArgs   = Array.prototype.slice.call(arguments, 1),
        fToBind = this,
        fNOP    = function() {},
        fBound  = function() {
          // this instanceof fBound === true 时,说明返回的 fBound 被当做 new 的构造函数调用
          return fToBind.apply(this instanceof fBound
                 ? this
                 : oThis,
                 // 获取调用时(fBound)的传参.bind 返回的函数入参往往是这么传递的
                 aArgs.concat(Array.prototype.slice.call(arguments)));
        };

    // 维护原型关系
    if (this.prototype) {
      // Function.prototype doesn't have a prototype property
      fNOP.prototype = this.prototype;
    }
    // 下行的代码使 fBound.prototype 是 fNOP 的实例,因此
    // 返回的 fBound 若作为 new 的构造函数,new 生成的新对象作为 this 传入 fBound,新对象的__proto__就是 fNOP 的实例
    fBound.prototype = new fNOP();

    return fBound;
  };
}


// 简单版
    Function.prototype.myBind = function(context,...arg){
        // this --->   fn
        var _this = this;
        return function(...ary){
            // _this(...arg)
            return _this.apply(context,arg.concat(ary))
        }
    }
```

实现类的继承

类的继承在几年前是重点内容，有 n 种继承方式各有优劣，es6 普及后越来越不重要，那么多种写法有点『回字有四样写法』的意思，如果还想深入理解的去看红宝书即可，我们目前只实现一种最理想的继承方式。

```
function Parent(name) {
    this.parent = name
}
Parent.prototype.say = function() {
    console.log(`${this.parent}: 你打篮球的样子像 kunkun`)
}
function Child(name, parent) {
    // 将父类的构造函数绑定在子类上
    Parent.call(this, parent)
    this.child = name
}
/**
 1. 这一步不用 Child.prototype =Parent.prototype 的原因是怕共享内存，修改父类原型对象就会影响子类
 2. 不用 Child.prototype = new Parent()的原因是会调用 2 次父类的构造方法（另一次是 call），会存在一份多余的父类实例属性
3. Object.create 是创建了父类原型的副本，与父类原型完全隔离
*/
Child.prototype = Object.create(Parent.prototype);
Child.prototype.say = function() {
    console.log(`${this.parent}好，我是练习时长两年半的${this.child}`);
}
// 注意记得把子类的构造指向子类本身
Child.prototype.constructor = Child;
var parent = new Parent('father');
parent.say() // father: 你打篮球的样子像 kunkun
var child = new Child('cxk', 'father');
child.say() // father 好，我是练习时长两年半的 cxk
```

谈谈你对 this 指向的理解

this 的指向，始终坚持一个原理：**this 永远指向最后调用它的那个对象**

改变 this 的指向我总结有以下几种方法：

- 使用 ES6 的箭头函数
- 在函数内部使用 _this = this
- 使用 apply、call、bind
- new 实例化一个对象

全局作用域下的 this 指向 window
如果给元素的事件行为绑定函数，那么函数中的 this 指向当前被绑定的那个元素
函数中的 this，要看函数执行前有没有 . ，有 . 的话，点前面是谁，this 就指向谁，如果没有点，指向 window
自执行函数中的 this 永远指向 window
定时器中函数的 this 指向 window
构造函数中的 this 指向当前的实例
call、apply、bind 可以改变函数的 this 指向
箭头函数中没有 this,如果输出 this,就会输出箭头函数定义时所在的作用域中的 this

8.DOM

1).新建节点

document.createElement("元素名") // 新建一个元素节点
document.createAttribute("属性名") // 新建一个属性节点
document.createTextNode("文本内容") // 创建一个文本节点
document.createDocumentFragment() // 新建一个 DOM 片段

2).添加、移除、替换、插入：

appendChild() // 向节点的子节点末尾添加新的子节点
removerChild() // 移除
parentNode.replaceChild(newChild, oldChild);用新节点替换父节点中已有的子节点
insertBeform() // 在已有的子节点前插入一个新的子节点

3).查找

document.getElementById() // 通过元素 id 查找,唯一性
document.getElementByClassName() // 通过 class 名称查找
document.getElementsByTagName() // 通过标签名称查找
document.getElementsByName() // 通过元素的 Name 属性的值查找

DOM 回流、重绘

DOM 回流(reflow)：页面中的元素增加、删除、大小、位置的改变，会引起浏览器重新计算 其他元素的位置，这种现象称为 DOM 回流。DOM 回流非常消耗性能，尽量避免 DOM 回流

DOM 重绘：元素的某些 css 样式如背景色、字体颜色等发生改变时，浏览器需要重新描绘渲 染这个元素，这种现象称为 DOM 重绘。

DOM 操作的读写分离

在 JS 中把设置样式和获取样式的两种操作分来来写， 设置样式的操作放在一起，读取样式的操作放在一起，这样可以有效的减少 DOM 的回流和重绘；

Box.style.background = ‘red’;
For(){}

Box.style.color = ‘green’

DOM 事件：

事件的传播机制：先冒泡，然后是目标阶段 然后再去捕获，我们可以利用事件的冒泡来进行事件委托，、也就是可以在父元素上绑定事件，通过事件对象 e 来判断点击的具体元素；可以提供性能；

我们可以利用的 e.stopPropagation()来阻止冒泡；利用 e.preventDefault()来阻止默认事件；

事件中有 0 级事件绑定和 2 级事件绑定

JS 盒子模型

```
// client offset scroll width height left top
// clientWidth 内容宽度 + 左右 padding
// offsetWidth clientWidth + 左右 border
// offsetTop 当前盒子的外边框到上级参照物的内边框的偏移量
// offsetParent 上级参照物：有定位的上级（包含 父级，祖父，曾祖父...）元素，所有所有上级都没有定位， 则参照物就是 body
// scroll 内容不溢出 等同于 client
// 内容溢出时 没有设置 overflow 值是内容宽高 + 上或左 padding
// 内容溢出时 有设置 overflow 时 值是内容宽高 + 上下或左右 padding
// scrollTop 卷去内容的高度
// 13 个属性 只有 scrollTop 和 scrollLeft 时可以设置值的， 其他的都是只读属性
```

9.JS 的异步编程

因为 js 是单线程的。浏览器遇到 setTimeout 和 setInterval 会先执行完当前的代码块，在此之前会把定时器推入浏览器的待执行时间队列里面，等到浏览器执行完当前代码之后会看下事件队列里有没有任务，有的话才执行定时器里的代码

常用的方式：setTimeout setIntervel ajax Promise asyc/await

宏任务(marcotask)微任务(microtask) 的执行顺序

先执行微任务，然后在执行宏任务；

JS 中的宏任务：setTimeout setIntervel ajax

JS 中的微任务：Promise.then Promise.catch await(可以理解成 Promise.then)

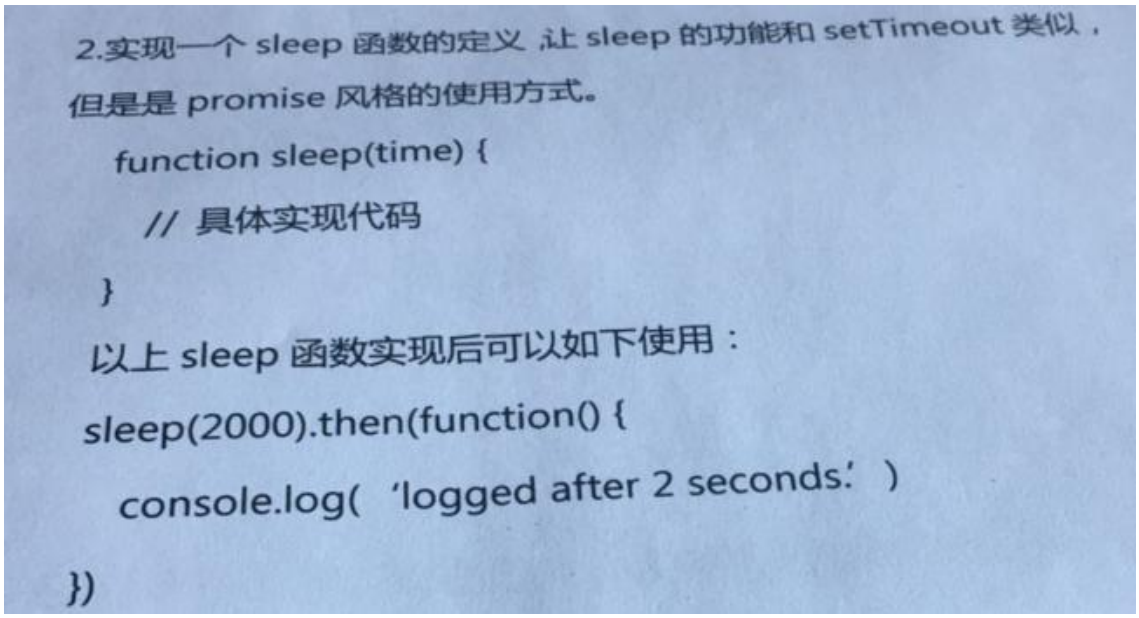
JS 的执行顺序是先同步 再异步；同步执行完成之前 异步不会执行

EventLoop 事件循环

EventQueue 事件队列

```
// 第二题
async function async1() {
    console.log("async1 start");
    await  async2();
    console.log("async1 end");
}
async  function async2() {
    console.log( 'async2');
}
console.log("script start");
setTimeout(function () {
    console.log("settimeout");
});
async1()
new Promise(function (resolve) {
    console.log("promise1");
    resolve();
}).then(function () {
    console.log("promise2");
});
setImmediate(()=>{
    console.log("setImmediate")
})
process.nextTick(()=>{
    console.log("process")
})
console.log('script end');
```

```
// 第一题
async function async1() {
    console.log("async1 start");
    await  async2();
    console.log("async1 end");
}

async  function async2() {
    console.log( 'async2');
}

console.log("script start");

setTimeout(function () {
    console.log("settimeout");
},0);

async1();

new Promise(function (resolve) {
    console.log("promise1");
    resolve();
}).then(function () {
    console.log("promise2");
});
console.log('script end');

```

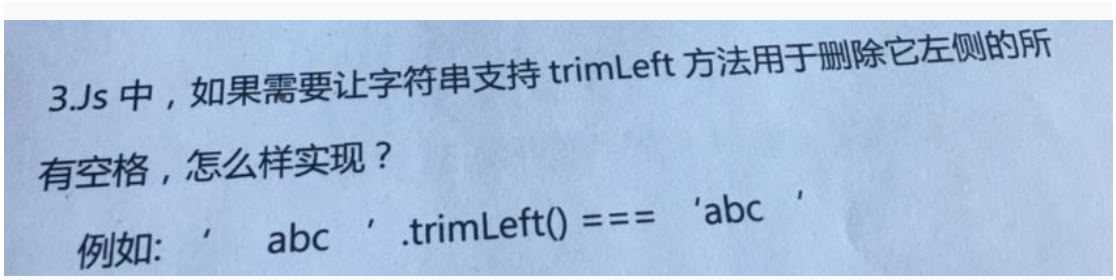


10.正则

```
//解析 URL Params 为对象
 var str = 'http://www.zhufengpeixun.cn/?lx=1&from=wx&b=12&c=13#qqqq';
function getParam(url){
  var reg = /([^?=&]+)=([^?=&#]+)/g;
```

```
    let obj = {};
    url.match(reg).forEach(item=>{
        let a = item.split('='); // ['lx','1']
        obj[a[0]] = a[1]
    })
    return obj
}
getParam(str);
//================================================

//模板引擎实现
let template = '我是{{name}}，年龄{{age}}，性别{{sex}}';
let data = {
    name: '姓名',
    age: 18
}
render(template, data); // 我是姓名，年龄 18，性别 undefined

function render(template, data) {
    const reg = /\{\{(\w+)\}\}/; // 模板字符串正则
    if (reg.test(template)) { // 判断模板里是否有模板字符串
        const name = reg.exec(template)[1]; // 查找当前模板里第一个模板字符串的字段
        template = template.replace(reg, data[name]); // 将第一个模板字符串渲染
        return render(template, data); // 递归的渲染并返回渲染后的结构
    }
    return template; // 如果模板没有模板字符串直接返回
}
//================================================

// 出现次数最多的字符
 var str = 'sfgsdfgsertdgfsdfgsertwegdsfgertewgsdfgsdg';
function getMax2(str) {
    str = str.split('').sort().join('');// 把字符串进行排序
    let key = '',num = 0;
    str.replace(/(\w)\1*/g,function($0,$1){
        if($0.length > num){
            num = $0.length;
            key = $1;
        }
    })
    return{
        key,num
    }
```

```
}
getMax2(str);
//================================================

// 千分符的实现
// 100,000,00
//方法 1
var str = '1234567'; // 1,234,567
function moneyFormate(str){
   str = str.split('').reverse().join('')
   let s = '';
   for(let i = 0; i < str.length ; i++){
      i%3 == 2 ? s+=str[i]+',' : s+=str[i]
   }
   s = s.split('').reverse().join('')
   return s
}
moneyFormate(str);// 1,234,567




// 方法 2
var str = '1234567';
function moneyFormate2(str){
   let s = '';
   // s = str.replace(/\d{1,3}(?=(\d{3})+$)/g,function(a){
   //       console.log(arguments)
   //       return a + ','
   // })
   s = str.replace(/(\d{1,3})(?=(\d{3})+$)/g,'$1,');
   return s;
}
moneyFormate2(str);
//================================================


var str = '    sdfgsg    fsgfsd      ';
// 使用正则去除字符串的首尾空格
// 以 1 到 多个 空格开头或者结尾的 都替换成空;
var res = str.replace(/^ +| +$/g,'')
```



10.http&ajax

1.TCP/IP 的三次握手和四次挥手

三次握手:

第一次握手：客户端向服务端发送 SYN 码数据包，表示客户端要求和服务端建立连接；

第二次握手：服务端收到客户端的连接请求后，会发送 ACK 数据包给客户端，表示你的连接 请求已经收到，询问客户端是否真的需要建立连接；

第三次握手：客户端收到 ACK 码以后会检验是否正确，如果正确，客户端会再次发送 ACK 码给 服务端，表示确认建立连接； (三次握手都成功以后才会建立连接，然后才会发送数据；)

四次挥手：

第一次挥手：当客户端发送数据结束后，会发送 FIN 码数据包给服务端，表示告知服务端客 户端的数据已经传递完了。

第二次挥手：当服务端收到 FIN 后，会发送 ACK 给客户端，表示服务端已经知道客户端传完 了。客户端收到 ACK 以后就会把传递数据给服务端的通道关闭；

第三次挥手：当服务端把响应的数据发送完毕后，会发送一个 FIN 给客户端，告知客户端响 应的数据已经发送完毕；

第四次挥手：当客户端收到 FIN 后，会发送一个 ACK 码数据包给服务端，告知服务端客户端已 经知道数据发送完毕；服务端收到 ACK 码后，可以安心的把数据传递通道关闭掉。

2.http 常用状态码(http-status-code):

2xx:表示成功

200 OK 表示所有东西都正常

204 表示请求成功,但是服务端没有内容给你

3xx: 表示重定向

301 永久重定向(当访问一个永久重定向的网站的时候,一个域名被指向一个其他网站,且是永久的)

302 临时重定向

304 走缓存(服务端觉得你之前请求过这个东西,而且服务器上的那一份没有发生变化,告诉客户端用缓存 就行)

- 301，Moved Permanently。永久重定向，该操作比较危险，需要谨慎操作：如果设置了 301，但是一段时间后又想取消，但是浏览器中已经有了缓存，还是会重定向。
- 302,Fount。临时重定向,但是会在重定向的时候改变 method: 把 POST 改成 GET，于是有了 307
- 307，Temporary Redirect。临时重定向，在重定向时不会改变 method

4xx: 表示客户端错误

400 参数传递不当,导致的错误

401 权限不够导致的

403 服务端已经理解请求,但是拒绝响应

404 客户端请求的资源或者数据不存在(发现请求接口 404,有两种情况一种是咱们写错接口了或者服 务端还没部署)

5xx: 表示服务端错误(遇到以 5 开头的错误去找服务端错误)

500 服务端内部错误

502 网关错误

3.从浏览器输入 URL 按回车到页面显示都发生了什么?

- 浏览器根据 URL 进行 DNS 查询

- 首先从 DNS 缓存中查询
- 若未在缓存中找到，则不停的向上一级级请求 DNS 服务器

- 取得 IP 地址，建立 TCP 连接
- 构造 HTTP 请求报

- 添加一些 HTTP 首部
- 根据同源策略添加 cookie

- 在 TCP 连接上发送 HTTP 报文，等待响应
- 服务器处理 HTTP 请求报文，返回响应 HTTP 响应报文
- 浏览器处理服务器返回的 HTTP 响应报文，若为 HTML 则渲染页面，不包括脚本的简单渲染流程如下

    1. 解析 DOM、CSSOM
    2. 根据 DOM、CSSOM 计算 render tree
    3. 根据 render tree 进行 layout

4. paint，至此，用户可以看到页面了

4.HTTPS 和 HTTP 的区别主要如下？

HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，要比 http 协议安全。

1、https 协议需要到 ca 申请证书，一般免费证书较少，因而需要一定费用。

2、http 是超文本传输协议，信息是明文传输，https 则是具有安全性的 ssl 加密传输协议。

3、http 和 https 使用的是完全不同的连接方式，用的端口也不一样，前者是 80，后者是 443。

4、http 的连接很简单，是无状态的；HTTPS 协议是由 SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，比 http 协议安全。

https 主要解决三个安全问题：

1. 内容隐私
2. 防篡改
3. 确认对方身份

https 并不是直接通过非对称加密传输过程，而是有握手过程，握手过程主要是和服务器做通讯，生成私有秘钥，最后通过该秘钥对称加密传输数据。还有验证证书的正确性。 证书验证过程保证了对方是合法的，并且中间人无法通过伪造证书方式进行攻击。

5.浏览器缓存？

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

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

协商缓存生效，返回 304 和 Not Modified

协商缓存失效，返回 200 和请求结果协商缓存可以通过设置两种 HTTP Header 实现：Last-Modified 和 ETag 。

强制缓存优先于协商缓存进行，若强制缓存(Expires 和 Cache-Control)生效则直接使用缓存，若不生效则进行协商缓存(Last-Modified / If-Modified-Since

和 Etag / If-None-Match)，协商缓存由服务器决定是否使用缓存，若协商缓存失效，那么代表该请求的缓存失效，返回 200，重新返回资源和缓存标识，再存入浏览器缓存中；生效则返回 304，继续使用缓存。

6.ajax 四步

1. 创建 XMLHttpRequest 对象,也就是创建一个异步调用对象
2. 创建一个新的 HTTP 请求,并指定该 HTTP 请求的方法、URL 及验证信息
3. 设置响应 HTTP 请求状态变化的函数
4. 发送 HTTP 请求

你使用过哪些 ajax?

从原生的 XHR 到 jquery ajax，再到现在的 axios 和 fetch。

axios 和 fetch 都是基于 Promise 的，一般我们在使用时都会进行二次封装

讲到 fetch 跟 jquery ajax 的区别，这也是它很奇怪的地方

当接收到一个代表错误的 HTTP 状态码时，从 fetch()返回的 Promise 不会被标记为 reject， 即使该 HTTP 响应的状态码是 404 或 500。相反，它会将 Promise 状态标记为 resolve （但是会将 resolve 的返回值的 ok 属性设置为 false ）， 仅当网络故障时或请求被阻止时，才会标记为 reject。 默认情况下，fetch 不会从服务端发送或接收任何 cookies, 如果站点依赖于用户 session，则会导致未经认证的请求（要发送 cookies，必须设置 credentials 选项）

一般我们再拦截器中都会写什么代码？

请求拦截中我们一半会把 token 写在这里，这样的话就不用每次请求都要写这个参数

还会做一个数据格式的处理，假如某个参数需要统一处理 可以放在这里，

响应拦截一半会做一个判断 请求失败的话直接调用失败提示框 这样不用每个接口都写同样的代码

也会再 return 时 return reponse.data;这样就可以不用每个数据接受的时候都加一个 data.data

get 请求和 post 请求有什么区别？什么时候使用 post?

GET：一般用于信息获取，使用 URL 传递参数，对所发送信息的数量也有限制，一般在 2000 个字符 POST：一般用于修改服务器上的资源，对所发送的信息没有限制

在以下情况中，请使用 POST 请求： 1. 无法使用缓存文件（更新服务器上的文件或数据库） 2. 向服务器发送大量数据（POST 没有数据量限制） 3. 发送包含未知字符的用户输入时，POST 比 GET 更稳定也更可靠

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

1、HTTP 协议 未规定 GET 和 POST 的长度限制

2、GET 的最大长度显示是因为 浏览器和 web 服务器限制了 URI 的长度

3、不同的浏览器和 WEB 服务器，限制的最大长度不一样

4、要支持 IE,则最大长度为 2083byte,若只支持 Chrome,则最大长度 8182byte

Cookie 和 Session 的区别?

- **安全性：** Session 比 Cookie 安全,Session 是存储在服务器端的,Cookie 是存储在客户端的。
- **存取值的类型不同：**Cookie 只支持存字符串数据，想要设置其他类型的数据，需要将其转换成字符串，Session 可以存任意数据类型。
- **有效期不同：** Cookie 可设置为长时间保持，比如我们经常使用的默认登录功能，Session 一般失效时间较短，客户端关闭（默认情况下）或者 Session 超时都会失效。
- **存储大小不同：** 单个 Cookie 保存的数据不能超过 4K，Session 可存储数据远高于 Cookie，但是当访问量过多，会占用过多的服务器资源。

- cookie、localStorage 以及 sessionStorage 的异同点:

分类	生命周期	存储容量	存储位置
cookie	默认保存在内存中，随浏览器关闭失效（如果设置过期时间 在到达过期时间后 效）	KB	保存在客户端，每次请求时都会带上
localStorage	理论上永久有效的，除非主动清除。	4.98MB（不同浏览器情况不同，safari 2.49M）	保存在客户端，不与服务端交互。节省网络流量
sessionStorage	仅在当前网页会话下有效，关闭页面或浏览器后会被清除。	4.98MB（部分浏览器没有限制）	同上

- 应用场景：localStorage 适合持久化缓存数据，比如页面的默认偏好配置等；sessionStorage 适合一次性临时数据保存。

Token 相关?

1. 客户端使用用户名跟密码请求登录
2. 服务端收到请求，去验证用户名与密码
3. 验证成功后，服务端会签发一个 token 并把这个 token 发送给客户端

4. 客户端收到 token 以后，会把它存储起来，比如放在 cookie 里或者 localStorage 里
5. 客户端每次向服务端请求资源的时候需要带着服务端签发的 token
6. 服务端收到请求，然后去验证客户端请求里面带着的 token ，如果验证成功，就向客户端返回请求的数据

- **每一次请求都需要携带 token，需要把 token 放到 HTTP 的 Header 里**
- **基于 token 的用户认证是一种服务端无状态的认证方式，服务端不用存放 token 数据。用解析 token 的计算时间换取 session 的存储空间，从而减轻服务器的压力，减少频繁的查询数据库**
- **token 完全由应用管理，所以它可以避开同源策略**

什么是同源策略？

同源策略是客户端脚本（尤其是 Javascript）的重要的安全度量标准。其目的是防止某个文档或脚本从多个不同源装载。 这里的同源策略指的是：协议，域名，端口相同，同源策略是一种安全协议，指一段脚本只能读取来自同一来源的窗口和文档的属性。

为什么要有同源限制？

我们举例说明：比如一个黑客程序，他利用 Iframe 把真正的银行登录页面嵌到他的页面上，当你使用真实的用户名，密码登录时，他的页面就可以通过 Javascript 读取到你的表单中 input 中的内容，这样用户名，密码就轻松到手了

工作中是怎么解决跨域的？

1.jsonp

1) JSONP 原理

利用 <script> 标签没有跨域限制的漏洞，网页可以得到从其他来源动态产生的 JSON 数据。JSONP 请求一定需要对方的服务器做支持才可以。

2.cors

CORS 需要浏览器和后端同时支持。浏览器会自动进行 CORS 通信，实现 CORS 通信的关键是后端。只要后端实现了 CORS，就实现了跨域。服务端设置 Access-Control-Allow-Origin 就可以开启 CORS。

3. proxy 代理 （适用于本地开发）

。。。（其他的方式 可自行去掘金上搜 9 种跨域的方式）

- CORS 支持所有类型的 HTTP 请求，是跨域 HTTP 请求的根本解决方案
- JSONP 只支持 GET 请求，JSONP 的优势在于支持老式浏览器，以及可以向不支持 CORS 的网站请求数据。
- 不管是 Node 中间件代理还是 nginx 反向代理，主要是通过同源策略对服务器不加限制。
- 日常工作中，用得比较多的跨域方案是 cors 和 nginx 反向代理

XSS 和 CSRF 区别

跨站脚本攻击（Cross Site Scripting)，为了不和层叠样式表 CSS 混淆，故将跨站脚本攻击缩写为 XSS。恶意攻击者往 Web 页面里插入恶意 Script 代码，当用户浏览该页之时，嵌入其中 Web 里面的 Script 代码会被执行，从而达到恶意攻击用户的目的。

跨站请求伪造（Cross-site request forgery），是伪造请求，冒充用户在站内的正常操作。我们知道，绝大多数网站是通过 cookie 等方式辨识用户身份，再予以授权的。所以要伪造用户的正常操作，最好的方法是通过 XSS 或链接欺骗等途径，让用户在本机（即拥有身份 cookie 的浏览器端）发起用户所不知道的请求。

区别：

原理不同，CSRF 是利用网站 A 本身的漏洞，去请求网站 A 的 api；XSS 是向目标网站注入 JS 代码，然后执行 JS 里的代码。

CSRF 需要用户先登录目标网站获取 cookie，而 XSS 不需要登录

CSRF 的目标是用户，XSS 的目标是服务器

XSS 是利用合法用户获取其信息，而 CSRF 是伪造成合法用户发起请求

XSS 攻击的防范

现在主流的浏览器内置了防范 XSS 的措施，例如 CSP。但对于开发者来说，也应该寻找可靠的解决方案来防止 XSS 攻击。

HttpOnly 防止劫取 Cookie

HttpOnly 最早由微软提出，至今已经成为一个标准。浏览器将禁止页面的 Javascript 访问带有 HttpOnly 属性的 Cookie。

上文有说到，攻击者可以通过注入恶意脚本获取用户的 Cookie 信息。通常 Cookie 中都包含了用户的登录凭证信息，攻击者在获取到 Cookie 之后，则可以发起 Cookie 劫持攻击。所以，严格来说，HttpOnly 并非阻止 XSS 攻击，而是能阻止 XSS 攻击后的 Cookie 劫持攻击。

输入检查

不要相信用户的任何输入。 对于用户的任何输入要进行检查、过滤和转义。建立可信任的字符和 HTML 标签白名单，对于不在白名单之列的字符或者标签进行过滤或编码。

在 XSS 防御中，输入检查一般是检查用户输入的数据中是否包含 <，> 等特殊字符，如果存在，则对特殊字符进行过滤或编码，这种方式也称为 XSS Filter。
而在一些前端框架中，都会有一份 `decodingMap`， 用于对用户输入所包含的特殊字符或标签进行编码或过滤，如 <，>，`script`，防止 XSS 攻击

输出检查

用户的输入会存在问题，服务端的输出也会存在问题。一般来说，除富文本的输出外，在变量输出到 HTML 页面时，可以使用编码或转义的方式来防御 XSS 攻击。例如利用 sanitize-html 对输出内容进行有规则的过滤之后再输出到页面中。

11.编程题

实现防抖函数（debounce）

```
// 防抖函数
const debounce = (fn, delay) => {
  let timer = null;
  return (...args) => {
    clearTimeout(timer);
    timer = setTimeout(() => {
      fn.apply(this, args);
    }, delay);
  };
};
```

实现节流函数（throttle）

```
// 节流函数
const throttle = (fn, delay = 500) => {
  let flag = true;
  return (...args) => {
    if (!flag) return;
    flag = false;
    setTimeout(() => {
      fn.apply(this, args);
      flag = true;
    }, delay);
  };
};
```

实现深克隆（deepclone）

```
function isArray (arr) {
    return Object.prototype.toString.call(arr) === '[object Array]';
}
// 深度克隆
function deepClone (obj) {
    if(typeof obj !== "object" && typeof obj !== 'function') {
        return obj;         //原始类型直接返回
    }
    var o = isArray(obj) ? [] : {};
    for(i in obj) {
        if(obj.hasOwnProperty(i)){
            o[i] = typeof obj[i] === "object" ? deepClone(obj[i]) : obj[i];
        }
    }
    return o;
}
```

其他手写系列 请在掘金上搜索 （22 道高频 JavaScript 手写面试题及答案）
https://juejin.im/post/5e100cdef265da5d75243229

有精力的可以再去掘金上搜索 （前端 100 问）
https://juejin.im/post/5d23e750f265da1b855c7bbe

Vue 相关

1.vue 生命周期

什么是 Vue 生命周期？

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

Vue 生命周期的作用是什么？

它的生命周期中有多个事件钩子，让我们在控制整个 Vue 实例的过程时更容易形成好的逻辑

Vue 生命周期总共有几个阶段？

它可以总共分为 8 个阶段：创建前/后，载入前/后,更新前/后,销毁前/销毁后

第一次页面加载会触发哪几个钩子？

第一次页面加载时会触发 beforeCreate, created, beforeMount, mounted 这几个钩子

DOM 渲染在哪个周期中就已经完成？

DOM 渲染在 mounted 中就已经完成了

每个生命周期适合哪些场景？

生命周期钩子的一些使用方法：

beforecreate ： 可以在这加个 loading 事件，在加载实例时触发

created ： 初始化完成时的事件写在这里，如在这结束 loading 事件，异步请求也适宜在这里调用

mounted ： 挂载元素，获取到 DOM 节点

updated ： 如果对数据统一处理，在这里写上相应函数

beforeDestroy ： 可以做一个确认停止事件的确认框

nextTick ： 更新数据后立即操作 dom

- beforeCreate 阶段：vue 实例的挂载元素 el 和数据对象 data 都是 undefined，还没有初始化。
- created 阶段：vue 实例的数据对象 data 有了，可以访问里面的数据和方法，未挂载到 DOM，el 还没有
- beforeMount 阶段：vue 实例的 el 和 data 都初始化了，但是挂载之前为虚拟的 dom 节点
- mounted 阶段：vue 实例挂载到真实 DOM 上，就可以通过 DOM 获取 DOM 节点
- beforeUpdate 阶段：响应式数据更新时调用，发生在虚拟 DOM 打补丁之前，适合在更新之前访问现有的 DOM，比如手动移除已添加的事件监听器
- updated 阶段：虚拟 DOM 重新渲染和打补丁之后调用，组成新的 DOM 已经更新，避免在这个钩子函数中操作数据，防止死循环

- beforeDestroy 阶段：实例销毁前调用，实例还可以用，this 能获取到实例，常用于销毁定时器，解绑事件
- destroyed 阶段：实例销毁后调用，调用后所有事件监听器会被移除，所有的子实例都会被销毁

2.v-show 与 v-if 区别

v-show 是 css 切换，v-if 是完整的销毁和重新创建

使用 频繁切换时用 v-show，运行时较少改变时用 v-if

v-if='false' v-if 是条件渲染，当 false 的时候不会渲染

3.MVVM 相关

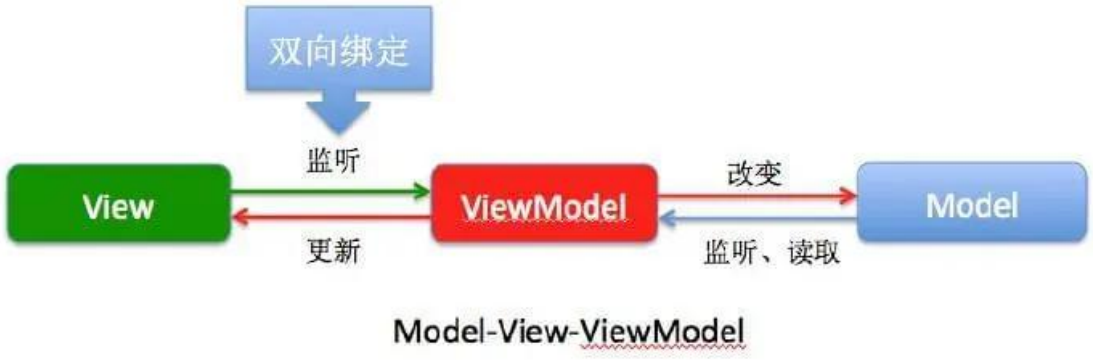
vue 采用数据劫持结合发布者-订阅者模式的方式，通过 Object.defineProperty 劫持 data 属性的 setter，getter，在数据变动时发布消息给订阅者，触发相应的监听回调。

MVVM

M - Model，Model 代表数据模型，也可以在 Model 中定义数据修改和操作的业务逻辑

V - View，View 代表 UI 组件，它负责将数据模型转化为 UI 展现出来

VM - ViewModel，ViewModel 监听模型数据的改变和控制视图行为、处理用户交互，简单理解就是一个同步 View 和 Model 的对象，连接 Model 和 View



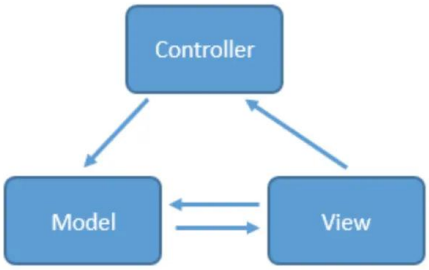
Model-View-ViewModel

- View 接收用户交互请求
- View 将请求转交给 ViewModel
- ViewModel 操作 Model 数据更新

- Model 更新完数据，通知 ViewModel 数据发生变化
- ViewModel 更新 View 数据

MVC

- View 接受用户交互请求
- View 将请求转交给 Controller 处理
- Controller 操作 Model 进行数据更新保存
- 数据更新保存之后，Model 会通知 View 更新
- View 更新变化数据使用户得到反馈



MVVM 模式和 MVC 有些类似，但有以下不同

- ViewModel 替换了 Controller，在 UI 层之下
- ViewModel 向 View 暴露它所需要的数据和指令对象
- ViewModel 接收来自 Model 的数据

概括起来，MVVM 是由 MVC 发展而来，通过在 Model 之上而在 View 之下增加一个非视觉的组件将来自 Model 的数据映射到 View 中。

4.说说你对 SPA 单页面的理解，它的优缺点分别是什么？

SPA（ single-page application ）仅在 Web 页面初始化时加载相应的 HTML、JavaScript 和 CSS。一旦页面加载完成，SPA 不会因为用户的操作而进行页面的重新加载或跳转；取而代之的是利用路由机制实现 HTML 内容的变换，UI 与用户的交互，避免页面的重新加载。

优点：

- 用户体验好、快，内容的改变不需要重新加载整个页面，避免了不必要的跳转和重复渲染；

- 基于上面一点，SPA 相对对服务器压力小；
- 前后端职责分离，架构清晰，前端进行交互逻辑，后端负责数据处理；

缺点：

- 初次加载耗时多：为实现单页 Web 应用功能及显示效果，需要在加载页面的时候将 JavaScript、CSS 统一加载，部分页面按需加载；
- 前进后退路由管理：由于单页应用在一个页面中显示所有的内容，所以不能使用浏览器的前进后退功能，所有的页面切换需要自己建立堆栈管理；
- SEO 难度较大：由于所有的内容都在一个页面中动态替换显示，所以在 SEO 上其有着天然的弱势。

5、computed 和 watch 的区别和运用的场景？

computed： 是计算属性，依赖其它属性值，并且 computed 的值有缓存，只有它依赖的属性值发生改变，下一次获取 computed 的值时才会重新计算 computed 的值；

watch： 更多的是「观察」的作用，类似于某些数据的监听回调 ，每当监听的数据变化时都会执行回调进行后续操作；

运用场景：

- 当我们需要进行数值计算，并且依赖于其它数据时，应该使用 computed，因为可以利用 computed 的缓存特性，避免每次获取值时，都要重新计算；
- 当我们需要在数据变化时执行异步或开销较大的操作时，应该使用 watch，使用 watch 选项允许我们执行异步操作 (访问一个 API)，限制我们执行该操作的频率，并在我们得到最终结果前，设置中间状态。这些都是计算属性无法做到的。

computed 的原理？

computed 本质是一个惰性求值的观察者。

computed 内部实现了一个惰性的 watcher，也就是 computed watcher，computed watcher 不会立刻求值，同时持有一个 dep 实例。

其内部通过 this.dirty 属性标记计算属性是否需要重新求值。

当 computed 的依赖状态发生改变时，就会通知这个惰性的 watcher，

computed watcher 通过 this.dep.subs.length 判断有没有订阅者，

有的话，会重新计算，然后对比新旧值，如果变化了，会重新渲染。（**Vue 想确保不仅仅是计算属性依赖的值发生变化，而是当计算属性最终计算的值发生变化时才会触发渲染 watcher 重新渲染，本质上是一种优化。**）

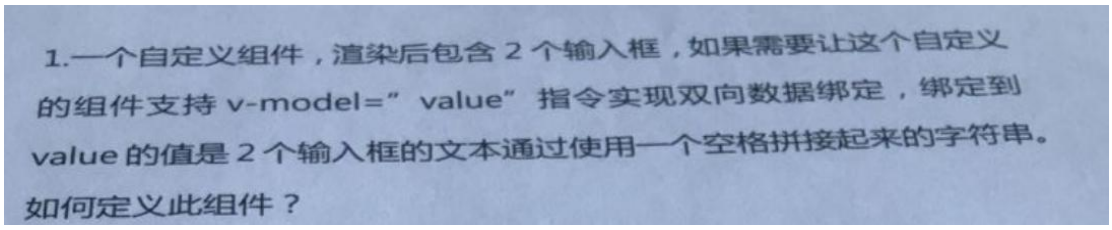
没有的话,仅仅把 this.dirty = true。 **(当计算属性依赖于其他数据时，属性并不会立即重新计算，只有之后其他地方需要读取属性的时候，它才会真正计算，即具备 lazy（懒计算）特性。)**

6.v-model 的原理

我们在 vue 项目中主要使用 v-model 指令在表单 input、textarea、select 等元素上创建双向数据绑定，我们知道 v-model 本质上不过是语法糖，v-model 在内部为不同的输入元素使用不同的属性并抛出不同的事件：

- text 和 textarea 元素使用 value 属性和 input 事件；
- checkbox 和 radio 使用 checked 属性和 change 事件；
- select 字段将 value 作为 prop 并将 change 作为事件。

```
<input v-model='something'>

相当于

<input v-bind:value="something" v-on:input="something = $event.target.value">
```



7.VUE 和 REACT 的区别？

react 整体是函数式的思想,把组件设计成纯组件,状态和逻辑通过参数传入,所以在 react 中，是单向数据流；

vue 的思想是响应式的，也就是基于是数据可变的，通过对每一个属性建立 Watcher 来监听，当属性变化的时候，响应式的更新对应的虚拟 dom。

核心思想：

vue 的整体思想仍然是拥抱经典的 html(结构)+css(表现)+js(行为)的形式,vue 鼓励开发者使用 template 模板，并提供指令供开发者使用(v-if、v-show、v-for 等等)，因此在开发 vue 应用的时候会有一种在写经典 web 应用（结构、表现、行为分离）的感觉。另一方面，在针对组件数据上，vue2.0 通过 Object.defineProperty 对数据做到了更细致的监听，精准实现组件级别的更新。

react 整体上是函数式的思想，组件使用 jsx 语法，all in js，将 html 与 css 全都融入 javaScript，jsx 语法相对来说更加灵活，我一开始刚转过来也不是很适应，感觉写 react 应用感觉就像是在写 javaScript。当组件调用 setState 或 props 变化的时候，组件内部 render 会重新渲染，子组件也会随之重新渲染，可以通过 shouldComponentUpdate 或者 PureComponent 可以避免不必要的重新渲染（个人感觉这一点上不如 vue 做的好）。

具体可参考掘金文章（关于 Vue 和 React 的一些对比及个人思考：https://juejin.im/post/5e153e096fb9a048297390c1）

8. 为什么在 Vue3.0 采用了 Proxy,抛弃 Object.defineProperty？

Object.defineProperty 只能劫持对象的属性,因此我们需要对每个对象的每个属性进行遍历。Vue 2.x 里,是通过 递归 + 遍历 data 对象来实现对数据的监控的,如果属性值也是对象那么需要深度遍历,显然如果能劫持一个完整的对象是才是更好的选择。

Proxy 可以劫持整个对象,并返回一个新的对象。Proxy 不仅可以代理对象,还可以代理数组。还可以代理动态增加的属性。

Proxy 的优势如下:

- Proxy 可以直接监听对象而非属性;
- Proxy 可以直接监听数组的变化;
- Proxy 有多达 13 种拦截方法,不限于 apply、ownKeys、deleteProperty、has 等等是 Object.defineProperty 不具备的;
- Proxy 返回的是一个新对象,我们可以只操作新的对象达到目的,而 Object.defineProperty 只能遍历对象属性直接修改;
- Proxy 作为新标准将受到浏览器厂商重点持续的性能优化,也就是传说中的新标准的性能红利;

Object.defineProperty 的优势如下:

- 兼容性好，支持 IE9，而 Proxy 的存在浏览器兼容性问题,而且无法用 polyfill 磨平，因此 Vue 的作者才声明需要等到下个大版本(3.0)才能用 Proxy 重写。

9. Vue 组件 data 为什么必须是函数 ？

因为组件是可以复用的,JS 里对象是引用关系,如果组件 data 是一个对象,那么子组件中的 data 属性值会互相污染,产生副作用。

所以一个组件的 data 选项必须是一个函数,因此每个实例可以维护一份被返回对象的独立的拷贝。new Vue 的实例是不会被复用的,因此不存在以上问题。

10、谈谈你对 keep-alive 的了解？

keep-alive 是 Vue 内置的一个组件，可以使被包含的组件保留状态，避免重新渲染 ，其有以下特性：

- 一般结合路由和动态组件一起使用，用于缓存组件；
- 提供 include 和 exclude 属性，两者都支持字符串或正则表达式，include 表示只有名称匹配的组件会被缓存，exclude 表示任何名称匹配的组件都不会被缓存 ，其中 exclude 的优先级比 include 高；
- 对应两个钩子函数 activated 和 deactivated ，当组件被激活时，触发钩子函数 activated，当组件被移除时，触发钩子函数 deactivated。

11、Vue 组件间通信有哪几种方式？

（1）props / $emit 适用 父子组件通信

这种方法是 Vue 组件的基础，相信大部分同学耳闻能详，所以此处就不举例展开介绍。

（2）ref 与 $parent / $children 适用 父子组件通信

- ref：如果在普通的 DOM 元素上使用，引用指向的就是 DOM 元素；如果用在子组件上，引用就指向组件实例
- $parent / $children：访问父 / 子实例

（3）EventBus （$emit / $on） 适用于 父子、隔代、兄弟组件通信

这种方法通过一个空的 Vue 实例作为中央事件总线（事件中心），用它来触发事件和监听事件，从而实现任何组件间的通信，包括父子、隔代、兄弟组件

（4）$attrs/$listeners 适用于 隔代组件通信

- $attrs：包含了父作用域中不被 prop 所识别 (且获取) 的特性绑定 (class 和 style 除外)。当一个组件没有声明任何 prop 时，这里会包含所有父作用域的绑定 (class 和 style 除外)，并且可以通过 v-bind="$attrs" 传入内部组件。通常配合 inheritAttrs 选项一起使用。
- $listeners：包含了父作用域中的 (不含 .native 修饰器的) v-on 事件监听器。它可以通过 v-on="$listeners" 传入内部组件

（5）provide / inject 适用于 隔代组件通信

祖先组件中通过 provider 来提供变量，然后在子孙组件中通过 inject 来注入变量。 provide / inject API 主要解决了跨级组件间的通信问题，不过它的使用场景，主要是子组件获取上级组件的状态，跨级组件间建立了一种主动提供与依赖注入的关系。

（6）Vuex 适用于 父子、隔代、兄弟组件通信

Vuex 是一个专为 Vue.js 应用程序开发的状态管理模式。每一个 Vuex 应用的核心就是 store（仓库）。“store” 基本上就是一个容器，它包含着你的应用中大部分的状态 (state)。

- 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。
- 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。

12.请介绍一下你对 vuex 的理解？

Vuex 是一个专为 Vue.js 应用程序开发的状态管理模式。每一个 Vuex 应用的核心就是 store（仓库）。“store” 基本上就是一个容器，它包含着你的应用中大部分的状态 (state)。

（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）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。

主要包括以下几个模块：

- State：定义了应用状态的数据结构，可以在这里设置默认的初始状态。
- Getter：允许组件从 Store 中获取数据，mapGetters 辅助函数仅仅是将 store 中的 getter 映射到局部计算属性。
- Mutation：是唯一更改 store 中状态的方法，且必须是同步函数。
- Action：用于提交 mutation，而不是直接变更状态，可以包含任意异步操作。
- Module：允许将单一的 Store 拆分为多个 store 且同时保存在单一的状态树中。

VUEX 实现原理？ （课程中的代码）

13.请介绍一下你对 vue-router 的理解？

vue-router 实现原理？ （课程中的代码）

vue-router 有 3 种路由模式：hash、history、abstract，

- hash: 使用 URL hash 值来作路由。支持所有浏览器，包括不支持 HTML5 History Api 的浏览器；
- history：依赖 HTML5 History API 和服务器配置。具体可以查看 HTML5 History 模式；
- abstract：支持所有 JavaScript 运行环境，如 Node.js 服务器端。如果发现没有浏览器的 API，路由会自动强制进入这个模式.

1）hash 模式的实现原理

早期的前端路由的实现就是基于 location.hash 来实现的。其实现原理很简单，location.hash 的值就是 URL 中 # 后面的内容。比如下面这个网站，它的 location.hash 的值为 '#search'：

hash 路由模式的实现主要是基于下面几个特性：

- URL 中 hash 值只是客户端的一种状态，也就是说当向服务器端发出请求时，hash 部分不会被发送；

- hash 值的改变，都会在浏览器的访问历史中增加一个记录。因此我们能通过浏览器的回退、前进按钮控制 hash 的切换；
- 可以通过 a 标签，并设置 href 属性，当用户点击这个标签后，URL 的 hash 值会发生改变；或者使用 JavaScript 来对 loaction.hash 进行赋值，改变 URL 的 hash 值；
- 我们可以使用 hashchange 事件来监听 hash 值的变化，从而对页面进行跳转（渲染）。

（2）history 模式的实现原理

HTML5 提供了 History API 来实现 URL 的变化。其中做最主要的 API 有以下两个：history.pushState() 和 history.repalceState()。这两个 API 可以在不进行刷新的情况下，操作浏览器的历史纪录。唯一不同的是，前者是新增一个历史记录，后者是直接替换当前的历史记录，如下所示：

```
window.history.pushState(null, null, path);
window.history.replaceState(null, null, path);
```

history 路由模式的实现主要基于存在下面几个特性：

- 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）；
- history.pushState() 或 history.replaceState() 不会触发 popstate 事件，这时我们需要手动触发页面跳转（渲染）。

导航钩子函数（导航守卫）？

- 全局守卫

1. router.beforeEach 全局前置守卫 进入路由之前
2. router.beforeResolve 全局解析守卫(2.5.0+) 在 beforeRouteEnter 调用之后调用
3. router.afterEach 全局后置钩子 进入路由之后

```
// main.js 入口文件
    import router from './router'; // 引入路由
    router.beforeEach((to, from, next) => {
      next();
    });
    router.beforeResolve((to, from, next) => {
      next();
    });
```

```
router.afterEach((to, from) => {
  console.log('afterEach 全局后置钩子');
});
```

- 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) => {
        // ...
      }
    }
  ]
})
```

- 组件内的守卫 你可以在路由组件内直接定义以下路由导航守卫

```
const Foo = {
  template: `...`,
  beforeRouteEnter (to, from, next) {
    // 在渲染该组件的对应路由被 confirm 前调用
    // 不! 能! 获取组件实例 `this`
    // 因为当守卫执行前，组件实例还没被创建
  },
  beforeRouteUpdate (to, from, next) {
    // 在当前路由改变，但是该组件被复用时调用
    // 举例来说，对于一个带有动态参数的路径 /foo/:id，在 /foo/1 和
/foo/2 之间跳转的时候，
    // 由于会渲染同样的 Foo 组件，因此组件实例会被复用。而这个钩子就会
在这个情况下被调用。
    // 可以访问组件实例 `this`
  },
  beforeRouteLeave (to, from, next) {
    // 导航离开该组件的对应路由时调用，我们用它来禁止用户离开
    // 可以访问组件实例 `this`
    // 比如还未保存草稿，或者在用户离开前，
```

```
    将 setInterval 销毁，防止离开之后，定时器还在调用。
  }
}
```

14、Vue 中的 key 有什么作用？

Vue 中 key 的作用是：key 是为 Vue 中 vnode 的唯一标记，通过这个 key，我们的 diff 操作可以更准确、更快速

更准确：因为带 key 就不是就地复用了，在 sameNode 函数 a.key === b.key 对比中可以避免就地复用的情况。所以会更加准确。

更快速：利用 key 的唯一性生成 map 对象来获取对应节点，比遍历方式更快

15.ref 的作用

1. 获取 dom 元素 this.$refs.box
2. 获取子组件中的 datathis.$refs.box.msg
3. 调用子组件中的方法 this.$refs.box.open()

---

30 道 Vue 面试题，内含详细讲解（涵盖入门到精通，自测 Vue 掌握程度）
面试完 50 个人后我写下这篇总结
公司要求会使用框架 vue，面试题会被问及哪些？
Vue 项目性能优化 — 实践指南

React 相关

1.React 中 keys 的作用是什么？

key 是用来帮助 react 识别哪些内容被更改、添加或者删除。key 需要写在用数组渲染出来的元素内部，并且需要赋予其一个**稳定**的值。稳定在这里很重要，因为如果 key 值发生了变更，react 则会触发 UI 的重渲染。这是一个非常有用的特性。

key 的唯一性

在相邻的元素间，key 值必须是唯一的，如果出现了相同的 key，同样会抛出一个 Warning，告诉相邻组件间有重复的 key 值。并且只会渲染第一个重复 key 值中的元素，因为 react 会认为后续拥有相同 key 的都是同一个组件。

key 值不可读

虽然我们在组件上定义了 key，但是在其子组件中，我们并没有办法拿到 key 的值，因为 key 仅仅是给 react 内部使用的。如果我们需要使用到 key 值，可以通过其他方式传入，比如将 key 值赋给 id 等。、

2.调用 setState 之后发生了什么？

`setState(updater, callback)`这个方法是用来告诉 react 组件数据有更新，有可能需要重新渲染。它是**异步**的，react 通常会**集齐一批需要更新的组件**，然后一次性更新来保证**渲染的性能；**

在 react 的生命周期和合成事件中，react 仍然处于他的更新机制中，这时更新标识为 true。按照上述过程，这时无论调用多少次 setState，都会不会执行更新，而是将要更新的 state 存入_更新队列中，将要更新的组件存入一个数组中；当上一次更新机制执行完毕后会将更新标识设置为 false。这时将一次性执行之前累积的 setState。由执行机制看，setState 本身并不是异步的，而是如果在调用 setState 时，如果 react 正处于更新过程，当前更新会被暂存，等上一次更新执行后在执行，这个过程给人一种异步的假象；可以理解成在生命周期和合成事件中，setState 是异步的，但是在原生事件中，setState 是同步的；

3：什么是虚拟 DOM？

虚拟 DOM (VDOM)是真实 DOM 在内存中的表示。UI 的表示形式保存在内存中，并与实际的 DOM 同步。这是一个发生在渲染函数被调用和元素在屏幕上显示之间的步骤，整个过程被称为调和。

为什么虚拟 dom 会提高性能?

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

具体实现步骤如下：

1. 用 JavaScript 对象结构表示 DOM 树的结构；然后用这个树构建一个真正的 DOM 树，插到文档当中
2. 当状态变更的时候，重新构造一棵新的对象树。然后用新的树和旧的树进行比较，记录两棵树差异
3. 把 2 所记录的差异应用到步骤 1 所构建的真正的 DOM 树上，视图就更新了。
4. 把树形结构按照层级分解，只比较同级元素。

diff 算法

- 给列表结构的每个单元添加唯一的 key 属性，方便比较。
- React 只会匹配相同 class 的 component（这里面的 class 指的是组件的名字）
- 合并操作，调用 component 的 setState 方法的时候，React 将其标记为 dirty.到每一个事件循环结束，React 检查所有标记 dirty 的 component 重新绘制.
- 选择性子树渲染。开发人员可以重写 shouldComponentUpdate 提高 diff 的性能。。

4：类组件和函数组件之间的区别是啥？

1. **类组件**可以使用其他特性，如状态 state 和生命周期钩子。
2. 当组件只是接收 props 渲染到页面时，就是无状态组件，就属于函数组件，也被称为哑组件或展示组件。

函数组件和类组件当然是有区别的，而且函数组件的性能比类组件的性能要高，因为类组件使用的时候要实例化，而函数组件直接执行函数取返回结果即可。为了提高性能，尽量使用函数组件。

区别	函数组件	类组件
是否有 this	没有	有
是否有生命周期	没有	有
是否有状态 state	没有	有

5：React 中 refs 干嘛用的？

Refs 提供了一种访问在 render 方法中创建的 DOM 节点或者 React 元素的方法。在典型的数据流中，props 是父子组件交互的唯一方式，想要修改子组件，需要使用新的 pros 重新渲染它。凡事有例外，某些情况下咱们需要在典型数据流外，强制修改子代，这个时候可以使用 Refs。

咱们可以在组件添加一个 ref 属性来使用，该属性的值是一个回调函数，接收作为其第一个参数的底层 DOM 元素或组件的挂载实例。

```
class UnControlledForm extends Component {
  handleSubmit = () => {
    console.log("Input Value: ", this.input.value)
  }
  render () {
    return (
      <form onSubmit={this.handleSubmit}>
        <input
          type='text'
          ref={(input) => this.input = input} />
        <button type='submit'>Submit</button>
      </form>
    )
  }
}
```

请注意，input 元素有一个 ref 属性，它的值是一个函数。该函数接收输入的实际 DOM 元素，然后将其放在实例上，这样就可以在 handleSubmit 函数内部访问它。

经常被误解的只有在类组件中才能使用 refs，但是 refs 也可以通过利用 JS 中的闭包与函数组件一起使用。

```
function CustomForm ({handleSubmit}) {
  let inputElement
  return (
    <form onSubmit={() => handleSubmit(inputElement.value)}>
      <input
        type='text'
        ref={(input) => inputElement = input} />
      <button type='submit'>Submit</button>
    </form>
  )
}
```

6：state 和 props 区别是啥？

props 和 state 是普通的 JS 对象。虽然它们都包含影响渲染输出的信息，但是它们在组件方面的功能是不同的。即

1. state 是组件自己管理数据，控制自己的状态，可变；
2. props 是外部传入的数据参数，不可变；
3. 没有 state 的叫做无状态组件，有 state 的叫做有状态组件；
4. 多用 props，少用 state，也就是多写无状态组件。

7.react 性能优化是哪个周期函数？

shouldComponentUpdate 这个方法用来判断是否需要调用 render 方法重新描绘 dom。因为 dom 的描绘非常消耗性能，如果我们能在 shouldComponentUpdate 方法中能够写出更优化的 dom diff 算法，可以极大的提高性能。

8：什么是高阶组件？

高阶组件(HOC)是接受一个组件并返回一个新组件的函数。基本上，这是一个模式，是从 React 的组合特性中衍生出来的，称其为纯组件，因为它们可以接受任何动态提供的子组件，但不会修改或复制输入组件中的任何行为。

```
const EnhancedComponent = higherOrderComponent(WrappedComponent);
```

HOC 可以用于以下许多用例

1. 代码重用、逻辑和引导抽象
2. 渲染劫持
3. state 抽象和操作
4. props 处理

8：在构造函数调用 super 并将 props 作为参数传入的作用是啥？

在调用 super() 方法之前，子类构造函数无法使用 this 引用，ES6 子类也是如此。将 props 参数传递给 super() 调用的主要原因是在子构造函数中能够通过 this.props 来获取传入的 props。

传递 props

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    console.log(this.props);  // { name: 'sudheer',age: 30 }
  }
}
```

没传递 props

```
class MyComponent extends React.Component {
  constructor(props) {
    super();
    console.log(this.props); // undefined
    // 但是 Props 参数仍然可用
    console.log(props); // Prints { name: 'sudheer',age: 30 }
  }
  render() {
    // 构造函数外部不受影响
    console.log(this.props) // { name: 'sudheer',age: 30 }
  }
}
```

上面示例揭示了一点。props 的行为只有在构造函数中是不同的，在构造函数之外也是一样的。

9：什么是受控组件？

在 HTML 中，表单元素如 <input>、<textarea>和<select>通常维护自己的状态，并根据用户输入进行更新。当用户提交表单时，来自上述元素的值将随表单一起发送。

而 React 的工作方式则不同。包含表单的组件将跟踪其状态中的输入值，并在每次回调函数(例如 onChange)触发时重新渲染组件，因为状态被更新。以这种方式由 React 控制其值的输入表单元素称为**受控组件**。

受控组件和非受控组件区别是啥？

1. **受控组件**是 React 控制中的组件，并且是表单数据真实的唯一来源。
2. 非受控组件是由 DOM 处理表单数据的地方，而不是在 React 组件中。

尽管非受控组件通常更易于实现，因为只需使用 refs 即可从 DOM 中获取值，但通常建议优先选择受控制的组件，而不是非受控制的组件。
这样做的主要原因是受控组件支持即时字段验证，允许有条件地禁用/启用按钮，强制输入格式。

10：如何 实现 React.createElement ？

```
class Element{
  constructor(tagName,props,children){
    this.tagName = tagName;
    this.props = props||{};
    this.children = children
  }
  render(){
    let ele = document.createElement(this.tagName);
    Object.keys(this.props).forEach(key=>{
      if(key === 'className'){
        ele.setAttribute('class',this.props[key])
      }else if(key === 'htmlFor'){
        ele.setAttribute('for',this.props[key])
      }else{
        ele.setAttribute(key,this.props[key])
      }

    })
    this.children.forEach(item=>{
      item instanceof Element ?
      ele.appendChild(item.render()):
      ele.appendChild(document.createTextNode(item))
    })
    return ele;
  }
}



const React = {
  createElement(tagName,props,...children){
    return new Element(tagName,props,children)
  }
}
```

11：讲讲什么是 JSX ？

当 Facebook 第一次发布 React 时，他们还引入了一种新的 JS 方言 JSX，将原始 HTML 模板嵌入到 JS 代码中。JSX 代码本身不能被浏览器读取，必须使用 Babel 和 webpack 等工具将其转换为传统的 JS。很多开发人员就能无意识使用 JSX，因为它已经与 React 结合在一直了。

```
class MyComponent extends React.Component {
  render() {
    let props = this.props;
    return (
      <div className="my-component">
      <a href={props.url}>{props.name}</a>
      </div>
    );
  }
}
```

12：React 的生命周期方法有哪些？

1. componentWillMount:在渲染之前执行，用于根组件中的 App 级配置。
2. componentDidMount：在第一次渲染之后执行，可以在这里做 AJAX 请求，DOM 的操作或状态更新以及设置事件监听器。
3. componentWillReceiveProps：在初始化 render 的时候不会执行，它会在组件接受到新的状态(Props)时被触发，一般用于父组件状态更新时子组件的重新渲染
4. shouldComponentUpdate：确定是否更新组件。默认情况下，它返回 true。如果确定在 state 或 props 更新后组件不需要在重新渲染，则可以返回 false，这是一个提高性能的方法。
5. componentWillUpdate:在 shouldComponentUpdate 返回 true 确定要更新组件之前件之前执行。
6. componentDidUpdate:它主要用于更新 DOM 以响应 props 或 state 更改。
7. componentWillUnmount：它用于取消任何的网络请求，或删除与组件关联的所有事件监听器

13：这三个点(...)在 React 干嘛用的？

... 在 React（使用 JSX）代码中做什么？它叫什么？

```
<Modal {...this.props} title='Modal heading' animation={false}/>
```

这个叫扩展操作符号或者展开操作符，例如，如果 this.props 包含 a:1 和 b:2，则

```
<Modal {...this.props} title='Modal heading' animation={false}>
```

等价于下面内容：

```
<Modal a={this.props.a} b={this.props.b} title='Modal heading'
animation={false}>
```

扩展符号不仅适用于该用例，而且对于创建具有现有对象的大多数（或全部）属性的新对象非常方便，在更新 state 咱们就经常这么做：

```
this.setState(prevState => {
    return {foo: {...prevState.foo, a: "updated"}};
});
```

14：使用 React Hooks 好处是啥？

首先，Hooks 通常支持提取和重用跨多个组件通用的有状态逻辑，而无需承担高阶组件或渲染 props 的负担。Hooks 可以轻松地操作函数组件的状态，而不需要将它们转换为类组件。

Hooks 在类中不起作用，通过使用它们，咱们可以完全避免使用生命周期方法，例如 componentDidMount、componentDidUpdate、componentWillUnmount。相反，使用像 useEffect 这样的内置钩子。

15：什么是 React Hooks？

Hooks 是 React 16.8 中的新添加内容。它们允许在不编写类的情况下使用 state 和其他 React 特性。使用 Hooks，可以从组件中提取有状态逻辑，这样就可以独立地测试和重用它。Hooks 允许咱们在不改变组件层次结构的情况下重用有状态逻辑，这样在许多组件之间或与社区共享 Hooks 变得很容易。

16：React 中的 useState() 是什么？

下面说明 useState(0)的用途：

```
...
const [count, setCounter] = useState(0);
const [moreStuff, setMoreStuff] = useState(...);
...
const setCount = () => {
    setCounter(count + 1);
    setMoreStuff(...);
    ...
};
```

useState 是一个内置的 React Hook。useState(0) 返回一个元组，其中第一个参数 count 是计数器的当前状态，setCounter 提供更新计数器状态的方法。

咱们可以在任何地方使用 setCounter 方法更新计数状态-在这种情况下，咱们在 setCount 函数内部使用它可以做更多的事情，使用 Hooks，能够使咱们的代码保持更多功能，还可以避免过多使用基于类的组件。

17：描述 Flux 与 MVC?

传统的 MVC 模式在分离数据(Model)、UI(View 和逻辑(Controller)方面工作得很好，但是 MVC 架构经常遇到两个主要问题:

数据流不够清晰:跨视图发生的级联更新常常会导致混乱的事件网络，难于调试。

缺乏数据完整性:模型数据可以在任何地方发生突变，从而在整个 UI 中产生不可预测的结果。

使用 Flux 模式的复杂用户界面不再遭受级联更新，任何给定的 React 组件都能够根据 store 提供的数据重建其状态。Flux 模式还通过限制对共享数据的直接访问来加强数据完整性。

18：如何避免在 React 重新绑定实例？

有几种常用方法可以避免在 React 中绑定方法：

1.将事件处理程序定义为内联箭头函数

```
class SubmitButton extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      isFormSubmitted: false
    };
  }
  render() {
```

```
    return (
      <button onClick={() => {
        this.setState({ isFormSubmitted: true });
      }}>Submit</button>
    )
  }
}
```

2.使用箭头函数来定义方法：

```
class SubmitButton extends React.Component {
  state = {
    isFormSubmitted: false
  }
  handleSubmit = () => {
    this.setState({
      isFormSubmitted: true
    });
  }
  render() {
    return (
      <button onClick={this.handleSubmit}>Submit</button>
    )
  }
}
```

3.使用带有 Hooks 的函数组件

```
const SubmitButton = () => {
  const [isFormSubmitted, setIsFormSubmitted] = useState(false);
  return (
    <button onClick={() => {
        setIsFormSubmitted(true);
    }}>Submit</button>
  )
};
```

19.了解 redux 么，说一下 redux 的作用和运用流程，redux 有什么缺点

Redux 是一个 **数据管理中心**，可以把它理解为一个全局的 data store 实例。它通过一定的使用规则和限制，保证着数据的健壮性、可追溯和可预测性。它与 React 无关，可以独立运行于任何 JavaScript 环境中，从而也为同构应用提供了更好的数据同步通道。

- **核心理念**:
  - **单一数据源**: 整个应用只有唯一的状态树，也就是所有 state 最终维护在一个根级 Store 中；
  - **状态只读**: 为了保证状态的可控性，最好的方式就是监控状态的变化。那这里就两个必要条件：
    - Redux Store 中的数据无法被直接修改；
    - 严格控制修改的执行；
  - **纯函数**: 规定只能通过一个纯函数（Reducer）来描述修改；

项目相关

1.组件化和模块化

为什么要组件化开发

有时候页面代码量太大，逻辑太多或者同一个功能组件在许多页面均有使用，维护起来相当复杂，这个时候，就需要组件化开发来进行功能拆分、组件封装，已达到组件通用性，增强代码可读性，维护成本也能大大降低

组件化开发的优点

很大程度上降低系统各个功能的耦合性，并且提高了功能内部的聚合性。这对前端工程化及降低代码的维护来说，是有很大的好处的，耦合性的降低，提高了系统的伸展性，降低了开发的复杂度，提升开发效率，降低开发成本

组件化开发的原则

- 专一
- 可配置性
- 标准性
- 复用性
- 可维护性

模块化

为什么要模块化

早期的 javascript 版本没有块级作用域、没有类、没有包、也没有模块，这样会带来一些问题，如复用、依赖、冲突、代码组织混乱等，随着前端的膨胀，模块化显得非常迫切

模块化的好处

- 避免变量污染，命名冲突

- 提高代码复用率
- 提高了可维护性
- 方便依赖关系管理

2.你如何对网站的文件和资源进行优化？

期待的解决方案包括：
1. 文件合并
2. 文件最小化/文件压缩
3. 使用 CDN 托管
4. 缓存的使用（多个域名来提供缓存）

3.请说出三种减少页面加载时间的方法

1. 优化图片
2. 图像格式的选择（GIF：提供的颜色较少，可用在一些对颜色要求不高的地方）
3. 优化 CSS（压缩合并 css，如 margin-top, margin-left...）
4. 网址后加斜杠（如 www.campr.com/目录，会判断这个目录是什么文件类型，或者是目录。）
5. 标明高度和宽度（如果浏览器没有找到这两个参数，它需要一边下载图片一边计算大小，如果图片很多， 浏览器需要不断地调整页面。这不但影响速度，也影响浏览体验。 当浏览器知道了高度和宽度参数后，即使图片暂时无法显示，页面上也会腾出图片的空位，然后继续加载 后面的内容。从而加载时间快了，浏览体验也更好了）
6. 减少 http 请求（合并文件，合并图片）

4.VUE 项目中的优化

1.不要在模板里面写过多表达式
2.循环调用子组件时添加 key
3.频繁切换的使用 v-show，不频繁切换的使用 v-if
4.尽量少用 float，可以用 flex
5.按需加载，可以用 require 或者 import()按需加载需要的组件
6.路由懒加载

7. 对于图片过多的页面，为了加速页面加载速度，所以很多时候我们需要将页面内未出现在可视区域内的图片先不做加载，等到滚动到可视区域后再去加载。这样对于页面加载性能上会有很大的提升，也提高了用户体验。我们在项目中使用 Vue 的 vue-lazyload 插件：

（1）安装插件

```
npm install vue-lazyload --save-dev
```

（2）在入口文件 man.js 中引入并使用

```
import VueLazyload from 'vue-lazyload'
```

然后再 vue 中直接使用

```
Vue.use(VueLazyload)
```

或者添加自定义选项

```
Vue.use(VueLazyload, {
  preLoad: 1.3,
  error: 'dist/error.png',
  loading: 'dist/loading.gif',
  attempt: 1
})
```

复制代码

（3）在 vue 文件中将 img 标签的 src 属性直接改为 v-lazy ，从而将图片显示方式更改为懒加载显示：

```
<img v-lazy="/static/img/1.png">
```

8. Vue 会通过 Object.defineProperty 对数据进行劫持，来实现视图响应数据的变化，然而有些时候我们的组件就是纯粹的数据展示，不会有任何改变，我们就不需要 Vue 来劫持我们的数据，在大量数据展示的情况下，这能够很明显的减少组件初始化的时间，那如何禁止 Vue 劫持我们的数据呢？可以通过 Object.freeze 方法来冻结一个对象，一旦被冻结的对象就再也不能被修改了。

9 减少 ES6 转为 ES5 的冗余代码

(1) 首先，安装 babel-plugin-transform-runtime：

```
npm install babel-plugin-transform-runtime --save-dev
```

(2) 然后，修改 .babelrc 配置文件为:

```
"plugins": [
    "transform-runtime"
]
```

5.用没用过 git，说几个常用的命令？

```
git add .
git commit -m 'm'
git push
git pull
git merge
git branch
git checkout xxx
```

6.VUE 项目路由权限

前后端分离的权限管理基本就以下两种方式：

1. 后端生成当前用户相应的路由后由前端（用 Vue Router 提供的 API）`addRoutes` 动态加载路由。
2. 前端写好所有的路由，后端返回当前用户的角色，然后根据事先约定好的每个角色拥有哪些路由对角色的路由进行分配。

- 第一种，完全由后端控制路由，但这也意味着如果前端需要修改或者增减路由都需要经过后端大大的同意；
- 第二种，相对于第一种，前端相对会自由一些，但是如果角色权限发生了改变就需要前后端一起修改，而且如果某些（技术型）用户在前端修改了自己的角色权限就可以通过路由看到一些本不被允许看到的页面，虽然拿不到数据，但是有些页面还是不希望被不相关的人看到。

7.你做过的项目中有什么亮点吗？有什么让你自豪的吗

独立封装了什么什么组件，提高了代码的复用率，减少了冗余代码；

图片资源懒加载

对于图片过多的页面，为了加速页面加载速度，所以很多时候我们需要将页面内未出现在可视区域内的图片先不做加载， 等到滚动到可视区域后再去加载。这样对于页面加载性能上会有很大的提升，也提高了用户体验。我们在项目中使用 Vue 的 vue-lazyload 插件：

（1）安装插件

```
npm install vue-lazyload --save-dev
```

（2）在入口文件 man.js 中引入并使用

```
import VueLazyload from 'vue-lazyload'
```

然后再 vue 中直接使用

```
Vue.use(VueLazyload)
```

路由懒加载：

```
const Foo = () => import('./Foo.vue')
const router = new VueRouter({
  routes: [
    { path: '/foo', component: Foo }
  ]
})
```

8. 问如果需要设计一个 dialog，你会怎么设计它，有哪些 API？

9. 讲解下自己的项目，遇到的项目难点、怎么解决的？

10 最近开发的项目是什么，在项目负责那些工作，都用到那些技术栈，分别作用是什么

算法性质

1. 8个外表一样的小球 其中7个球重量相同 1个球为[异常球] 可能重量比较重也可能比较轻 利用天平称重至少多少次可以确保找出这个[异常球]，并需要知道到底是轻了还是重了。

这是一道非常有意思的题，答案为3次。

答案

一、将8个球先取四个组成A、B两组，每组2个。

二、、将A、B组进行第一次称，若不同重则有一组有问题。

三、将重组两个球第二次称，若不同重则有一个有问题。 将重组重球（若选轻球则以下结论相反）与轻组一球进行第三次称。 若球重，则这粒为重球为异常球；若相同，则剩余那个为轻异常球。

2 实现超出整数存储范围的两个大整数相加 function add(a,b)。注意 a 和 b 以及函数的返回值都是字符串。

```
function add (a, b) {
    let lenA = a.length,
        lenB = b.length,
        len = lenA > lenB ? lenA : lenB;

    // 先补齐位数一致
    if(lenA > lenB) {
        for(let i = 0; i < lenA - lenB; i++) {
            b = '0' + b;
        }
    } else {
        for(let i = 0; i < lenB - lenA; i++) {
            a = '0' + a;
        }
    }

    let arrA = a.split('').reverse(),
        arrB = b.split('').reverse(),
        arr = [],
        carryAdd = 0;

    for(let i = 0; i < len; i++) {
        let temp = Number(arrA[i]) + Number(arrB[i]) + carryAdd;
        arr[i] = temp > 9 ? temp - 10 : temp;
        carryAdd = temp >= 10 ? 1 : 0;
    }

    if(carryAdd === 1) {
        arr[len] = 1;
    }

    return arr.reverse().join('');

}
```

非技术问题

1.自我介绍

2.你上家公司为什么离职？

3.你的期望月薪是多少？

4.你对我们有什么问题要问吗？

5.你都看过关于前端的哪些书？

6.你上家公司地址多少？

掘金 70 个常见的非技术问题：**https://juejin.im/entry/5bf12262e51d45042c66eef7**

已工作同学的项目总结（仅供参考）

git commit -m ‘xxx’备注不要瞎写，会被嫌弃的

提交代码时，最好查看一下修改了那些文件，如果没有修改的撤回一下 避免提交一大堆空的修改上去

项目中的公共样式千万别去动!!

工作难的不是开发是改 bug，一定及时沟通!! 需求不明确问产品，字段不确定问后端 不问卡在那 最后担责任的还是自己

尽量养成良好的代码书写习惯，别套一大堆 if else 真的会被嫌弃！

1、 碰到 5XX 的问题，别傻乎乎的直接去找后端，要先用 Swager 或者 postman 测试下，碰到 java 写的后端，map 类型，Swager 无法测试，要用 postman 去测试。

2、 碰到增删改查业务。先写增加，然后改， 再查，最后删除，这类业务不要让后端给你造假数据。

3、 学会定义全局变量和常量，看能否少写请求，减少代码冗余，不然 codeRivew 的时候会被鄙视死。

4、 拿到项目，先把全局看下，能否拆分出组件，抽取出公用 JS，不然后期再去拆很痛苦 。

5、 学会看数据库，自己学着去看自己处理的东西，尤其是后端定义的字段。

6、 请求回来后端的数据，一定要 CTRL+C 然后 CTRL+V。不要装 B 自己去写，报错，你能排查到死

7、 碰到稀奇古怪的问题，直接上 debugger，F10 单步调试

8、 自己解决的一些古怪问题或者请教别人，网上查的，一定要记录下，最好写个博客，不然下次再碰到，很少能想起来怎么解决的了。

9、 多看掘金，有些上面人写的技巧，最好复制到自己的博客和笔记中，很实用。

10、碰到大佬级后端，一定要抱紧大腿，他们对业务的需求理解，完全秒杀很多前端。许多不懂的，可以去咨询。

11、 原生真的很重要。

12、 多看下设计模式，推荐张榕明的《JavaScript 设计模式》配合

周老师的原生 JS 面向对象，可以让代码逻辑更精简。

13、最好能进一个技术氛围好的技术团队，自己一个前端就是神级坑。非常不推荐

14. git commit 规范

15. git 最好使用一款界面的管理工具 推荐 IDE 内置或者 source tree

16. git 禁止使用 reset，push -f

17. 下载一款翻译软件，选中翻译，截屏翻译很实用

18. 拿到新项目先整体看一遍 package webpack 配置 main.js router store api 权限管理 表单校验 过滤器

19. vue 推荐<<深入浅出 Vue.js>>

20. 写业务先想清楚，接口文档及时要，数据结构会影响代码设计的

21. 代码最好边写边重构

22. UI 库常用组件进行二次封装

23. 保持规范的命名 文件 变量 组件

24. 不要轻易修改别人的代码，熟悉代码是一个非常有必要的过程，修改前先思考会有哪些影响

25. 遇到问题先分析，搜索引擎推荐谷歌

26. 学习从三个地方思考 如：为什么要有 promise promise 解决了哪些问题 如何解决的

27. 保持控制台干净，找不到报错在哪就用排除法

28. 博客和书优先选择看书

29.get 和 post 分别在请求头中传参

以 vue-cli3.0 搭建的项目为例,get 方式要用 params 传,post 不用,写假数据的时候可以不开 mock，直接在 api 里面的 js 文件写 new Promise 即可，例如：即可根据接口文档格式及所需数据写前端项目了，完成后换成真的接口即可（白底图是假数据，黑底是真正接口）

```
function detailListInterface(params) {
  console.log('detailListInterface', params)
  return new Promise(resolve => {
    setTimeout(function() {
      resolve({
        code: 0,
        msg: 'success',
        result: {
          pageNum: params.pageNum,
          pageSize: 10,
          pages: 5,
          total: 12,
          list: [
            {
              number: '安徽省',
              title: 12,
              name: 10,
              mechNo: 14,
              account: '湖北销售部—组',
              province: '湖北销售部—组湖北销售部—组'

            },
```

```
// 下载失败原因
export function seaImportFailRecDownLoad(data) {
  return requestDownload({
    url: '/excel/seaImportFailRecDownLoad',
    method: 'post',
    data
  })
}
```

补充：调接口如果不通，如果后端发了 swagger，那么先去相对环境的 swagger 去 try，如果 swagger 通了，先检查前端看是否有问题。不通就直接找后端

30.分页的理解：

传给后台的参数 pageNum 和 pageSize 比如：1 和 50，就表示从第一页从头开始查起的 50000 条数据，如果 2 和 50，就表示从第二页开始查起，即第 51 条开始查 50 条

31.按钮要加防抖：具体百度即可

32　学会看 network　前后端联调很重要