目录

第 1 题：写 React / Vue 项目时为什么要在列表组件中写 key，其作用是什么？

key 是给每一个 vnode 的唯一 id,可以依靠 key,更准确,更快的拿到 oldVnode 中对应的 vnode 节点

第 2 题：['1', '2', '3'].map(parseInt) what & why ？

第一眼看到这个题目的时候，脑海跳出的答案是 [1, 2, 3]，但是真正的答案是[1, NaN, NaN]。
首先让我们回顾一下，map 函数的第一个参数 callback。这个 callback 一共可以接收三个参数，其中第一个参数代表当前被处理的元素，而第二个参数代表该元素的索引。

```
arr.map(callback: (value: T, index: number, array: T[]) => U, thisArg?: any);
```

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

```
parseInt(string, radix)
```

了解这两个函数后，我们可以模拟一下运行情况

```
parseInt('1', 0) //radix 为 0 时，且 string 参数不以“0x”和“0”开头时，按照 10 为基数处理。这个时候返回 1parseInt('2', 1) //基数为 1（1 进制）
```

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

第 3 题：什么是防抖和节流？有什么区别？如何实现？

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

```
function debounce(fn) {
  let timeout = null
 // 创建一个标记用来存放定时器的返回值
  return function() {
    clearTimeout(timeout)
     // 每当用户输入的时候把前一个 setTimeout clear 掉
        timeout = setTimeout(() => {
          // 然后又创建一个新的 setTimeout, 这样就能保证输入字符后的 interval 间隔内如果还有字符输入的话，就不会执行 fn 函数
fn.apply(this, arguments)
        }, 500)
      }}function sayHi() {
      console.log('防抖成功')}var inp =
     document.getElementById('inp')inp.addEventListener('input', debounce(sayHi)) // 防抖
```

节流——高频事件触发，但在 n 秒内只会执行一次，所以节流会稀释函数的执行频率。

```
function throttle(fn) {
  let canRun = true // 通过闭包保存一个标记
  return function() {
    if (!canRun) return
     // 在函数开头判断标记是否为 true，不为 true 则 return
        canRun = false // 立即设置为 false
        setTimeout(() => {
          // 将外部传入的函数的执行放在 setTimeout 中
fn.apply(this, arguments)
          // 最后在 setTimeout 执行完毕后再把标记设置为 true(关键) 表示可以执行下一次循环了。当定时器没有执行的时候标记永远是 false，在开头被 return 掉
          canRun = true
        }, 500)
      }}function sayHi(e) {
      console.log(e.target.innerWidth, e.target.innerHeight)}window.addEventListener('resize', throttle(sayHi))
```

第 4 题：介绍下 Set、Map、WeakSet 和 WeakMap 的区别？

Set——对象允许你存储任何类型的唯一值，无论是原始值或者是对象引用
WeakSet——成员都是对象；成员都是弱引用，可以被垃圾回收机制回收，可以用来保存 DOM 节点，不容易造成内存泄漏；
Map——本质上是键值对的集合，类似集合；可以遍历，方法很多，可以跟各种数据格式转换。
WeakMap——只接受对象最为键名（null 除外），不接受其他类型的值作为键名；键名是弱引用，键值可以是任意的，键名所指向的对象可以被垃圾回收，此时键名是无效的；不能遍历，方法有 get、set、has、delete。

第 5 题：介绍下深度优先遍历和广度优先遍历，如何实现？

深度优先遍历——是指从某个顶点出发，首先访问这个顶点，然后找出刚访问这个结点的第一个未被访问的邻结点，然后再以此邻结点为顶点，继续找它的下一个顶点进行访问。重复此步骤，直至所有结点都被访问完为止。
广度优先遍历——是从某个顶点出发，首先访问这个顶点，然后找出刚访问这个结点所有未被访问的邻结点，访问完后再访问这些结点中第一个邻结点的所有结点，重复此方法，直到所有结点都被访问完为止。

```
//1.深度优先遍历的递归写法 function deepTraversal(node) {
  let nodes = []
  if (node != null) {
   nodes.push[node]
    let childrens = node.children
        for (let i = 0;
    i < childrens.length; i++) deepTraversal(childrens[i])
     }  return nodes}
    //2.深度优先遍历的非递归写法 function deepTraversal(node) {
     let nodes = []
     if (node != null) {
       let stack = []
    //同来存放将来要访问的节点
       stack.push(node)
       while (stack.length != 0) {
         let item = stack.pop()
    //正在访问的节点
         nodes.push(item)
         let childrens = item.children
         for (
           let i = childrens.length - 1;
           i >= 0;
           i--
    //将现在访问点的节点的子节点存入 stack，供将来访问      )
           stack.push(childrens[i])
       }
     }
```

```
    return nodes}
  //3.广度优先遍历的递归写法 function wideTraversal(node) {
    let nodes = [],
      i = 0
    if (node != null) {
      nodes.push(node)
      wideTraversal(node.nextElementSibling)
      node = nodes[i++]
      wideTraversal(node.firstElementChild)
    }
    return nodes}//4.广度优先遍历的非递归写法 function
wideTraversal(node) {
    let nodes = [],
      i = 0  while (node != null) {
      nodes.push(node)
      node = nodes[i++]
      let childrens = node.children
      for (let i = 0;
   i < childrens.length;
   i++) {
        nodes.push(childrens[i])
      }
    }
    return nodes
  }
```

第 6 题：请分别用深度优先思想和广度优先思想实现一个拷贝函数？

深复制 深度优先遍历

```
let DFSdeepClone = (obj, visitedArr = []) => {
  let _obj = {}
  if (isTypeOf(obj, 'array') || isTypeOf(obj, 'object')) {
    let index = visitedArr.indexOf(obj)
    _obj = isTypeOf(obj, 'array') ? [] : {}
    if (~index) { // 判断环状数据
      _obj = visitedArr[index]
    } else {
      visitedArr.push(obj)
      for (let item in obj) {
        _obj[item] = DFSdeepClone(obj[item], visitedArr)
      }
    }
  } else if (isTypeOf(obj, 'function')) {
```

```
        _obj = eval('(' + obj.toString() + ')');
    } else {
        _obj = obj
    }
    return _obj
}
```

广度优先遍历

```
let BFSdeepClone = (obj) => {
        let origin = [obj],
            copyObj = {},
            copy = [copyObj]
            //  去除环状数据
        let visitedQueue = [],
            visitedCopyQueue = []
        while (origin.length > 0) {
            let items = origin.shift(),
                _obj = copy.shift()
            visitedQueue.push(items)
            if (isTypeOf(items, 'object') || isTypeOf(items, 'array')) {
                for (let item in items) {
                    let val = items[item]
                    if (isTypeOf(val, 'object')) {
                        let index = visitedQueue.indexOf(val)
                        if (!~index) {
                            _obj[item] = {}
                                //下次 while 循环使用给空对象提供数据
                            origin.push(val)
                                //  推入引用对象
                            copy.push(_obj[item])
                        } else {
                            _obj[item] = visitedCopyQueue[index]
                            visitedQueue.push(_obj)
                        }
                    } else if (isTypeOf(val, 'array')) {
                        //  数组类型在这里创建了一个空数组
                        _obj[item] = []
                        origin.push(val)
                        copy.push(_obj[item])
                    } else if (isTypeOf(val, 'function')) {
                        _obj[item] = eval('(' + val.toString() + ')');
                    } else {
                        _obj[item] = val
                    }
                }
```

```
            // 将已经处理过的对象数据推入数组 给环状数据使用
            visitedCopyQueue.push(_obj)
          } else if (isTypeOf(items, 'function')) {
            copyObj = eval('(' + items.toString() + ')');
          } else {
            copyObj = obj
          }
        }
    return copyObj
}
```

第 7 题：ES5/ES6 的继承除了写法以外还有什么区别？

1. ES5 的继承实质上是先创建子类的实例对象，然后再将父类的方法添加到 this 上（Parent.apply(this)）.
2. ES6 的继承机制完全不同，实质上是先创建父类的实例对象 this（所以必须先调用父类的 super()方法），然后再用子类的构造函数修改 this。
3. ES5 的继承时通过原型或构造函数机制来实现。
4. ES6 通过 class 关键字定义类，里面有构造方法，类之间通过 extends 关键字实现继承。
5. 子类必须在 constructor 方法中调用 super 方法，否则新建实例报错。因为子类没有自己的 this 对象，而是继承了父类的 this 对象，然后对其进行加工。如果不调用 super 方法，子类得不到 this 对象。
6. 注意 super 关键字指代父类的实例，即父类的 this 对象。
7. 注意：在子类构造函数中，调用 super 后，才可使用 this 关键字，否则报错。

function 声明会提升，但不会初始化赋值。Foo 进入暂时性死区，类似于 let、const 声明变量。

```
const bar = new Bar();
 // it's ok
function Bar() {
  this.bar = 42;
}const foo = new Foo();
 // ReferenceError: Foo is not definedclass Foo {
  constructor() {
    this.foo = 42;
      }
    }
```

class 声明内部会启用严格模式。

```
// 引用一个未声明的变量 function Bar() {
  baz = 42;
 // it's ok}const bar = new Bar();
```

```
class Foo {
  constructor() {
    fol = 42;
     // ReferenceError: fol is not defined
      }
    }const foo = new Foo();
```

class 的所有方法（包括静态方法和实例方法）都是不可枚举的。

```
// 引用一个未声明的变量function Bar() {
  this.bar = 42;}Bar.answer = function()
 {  return 42;
};
Bar.prototype.print = function() {
  console.log(this.bar);
};
const barKeys = Object.keys(Bar);
 // ['answer']const barProtoKeys = Object.keys(Bar.prototype);
 // ['print']class Foo {
  constructor() {
   this.foo = 42;
  }
  static answer() {
   return 42;
  }
  print() {
   console.log(this.foo);
  }}const fooKeys = Object.keys(Foo);
 // []const fooProtoKeys = Object.keys(Foo.prototype);
 // []
```

class 的所有方法（包括静态方法和实例方法）都没有原型对象 prototype，所以也没有[[construct]]，不能使用 new 来调用。

```
function Bar() {
  this.bar = 42;
}Bar.prototype.print = function() {
  console.log(this.bar);
};
const bar = new Bar();
const barPrint = new bar.print();
 // it's okclass Foo {
  constructor() {
    this.foo = 42;
      }  print() {
        console.log(this.foo);
      }}const foo = new Foo();
```

```
    const fooPrint = new foo.print();
     // TypeError: foo.print is not a constructor
```

必须使用 new 调用 class。

```
function Bar() {
  this.bar = 42;
}const bar = Bar();
 // it's okclass Foo {
  constructor() {
    this.foo = 42;
     }}const foo = Foo();
     // TypeError: Class constructor Foo cannot be invoked without 'new'
```

class 内部无法重写类名。

```
function Bar() {
  Bar = 'Baz';
// it's ok  this.bar = 42;
}const bar = new Bar();
// Bar: 'Baz'
// bar: Bar {bar: 42}
  class Foo {
  constructor() {
    this.foo = 42;
        Foo = 'Fol';
     // TypeError: Assignment to constant variable
      }}const foo = new Foo();
    Foo = 'Fol';
     // it's ok
```

第 8 题：setTimeout、Promise、Async/Await 的区别

https://blog.csdn.net/yun_hou/article/details/88697954

第 9 题：Async/Await 如何通过同步的方式实现异步

async 起什么作用——输出的是一个 Promise 对象

第 10 题：异步笔试题请写出下面代码的运行结果

```
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)
```

```
1)      {
  console.log('promise1')
  resolve()}).then(function()
 {
  console.log('promise2')})console.log('script end')
//输出
//script start
//async1 start
//async2
//promise1
//script end
//async1 end
//promise2
//setTimeout
```

第 11 题：算法手写题

已知如下数组，编写一个程序将数组扁平化去并除其中重复部分数据，最终得到一个升序且不重复的数组

var arr = [[1, 2, 2], [3, 4, 5, 5], [6, 7, 8, 9, [11, 12, [12, 13, [14]]]], 10];

答：使用 Set 方法去重，flat(Infinity)扁平化

```
Array.from(new Set(arr.flat(Infinity))).sort((a,b)=>{ return a-b})//[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

第 12 题：JS 异步解决方案的发展历程以及优缺点。

1、回调函数（callback）

优点：解决了同步的问题（只要有一个任务耗时很长，后面的任务都必须排队等着，会拖延整个程序的执行。）

缺点：回调地狱，不能用 try catch 捕获错误，不能 return

2、Promise

优点：解决了回调地狱的问题

缺点：无法取消 Promise ，错误需要通过回调函数来捕获

3、Generator

特点：可以控制函数的执行，可以配合 co 函数库使用

4、Async/await

优点：代码清晰，不用像 Promise 写一大堆 then 链，处理了回调地狱的问题

缺点：await 将异步代码改造成同步代码，如果多个异步操作没有依赖性而使用 await 会导致性能上的降低。

第 13 题：Promise 构造函数是同步执行还是异步执行，那么 then 方法呢？

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

```
  console.log(2)})promise.then(() => {
  console.log(3)})console.log(4)
```

执行结果是：1243，promise 构造函数是同步执行的，then 方法是异步执行的

第 14 题：情人节福利题，如何实现一个 new

第 15 题：简单讲解一下 http2 的多路复用

HTTP2 采用二进制格式传输，取代了 HTTP1.x 的文本格式，二进制格式解析更高效。

多路复用代替了 HTTP1.x 的序列和阻塞机制，所有的相同域名请求都通过同一个 TCP 连接并发完成。

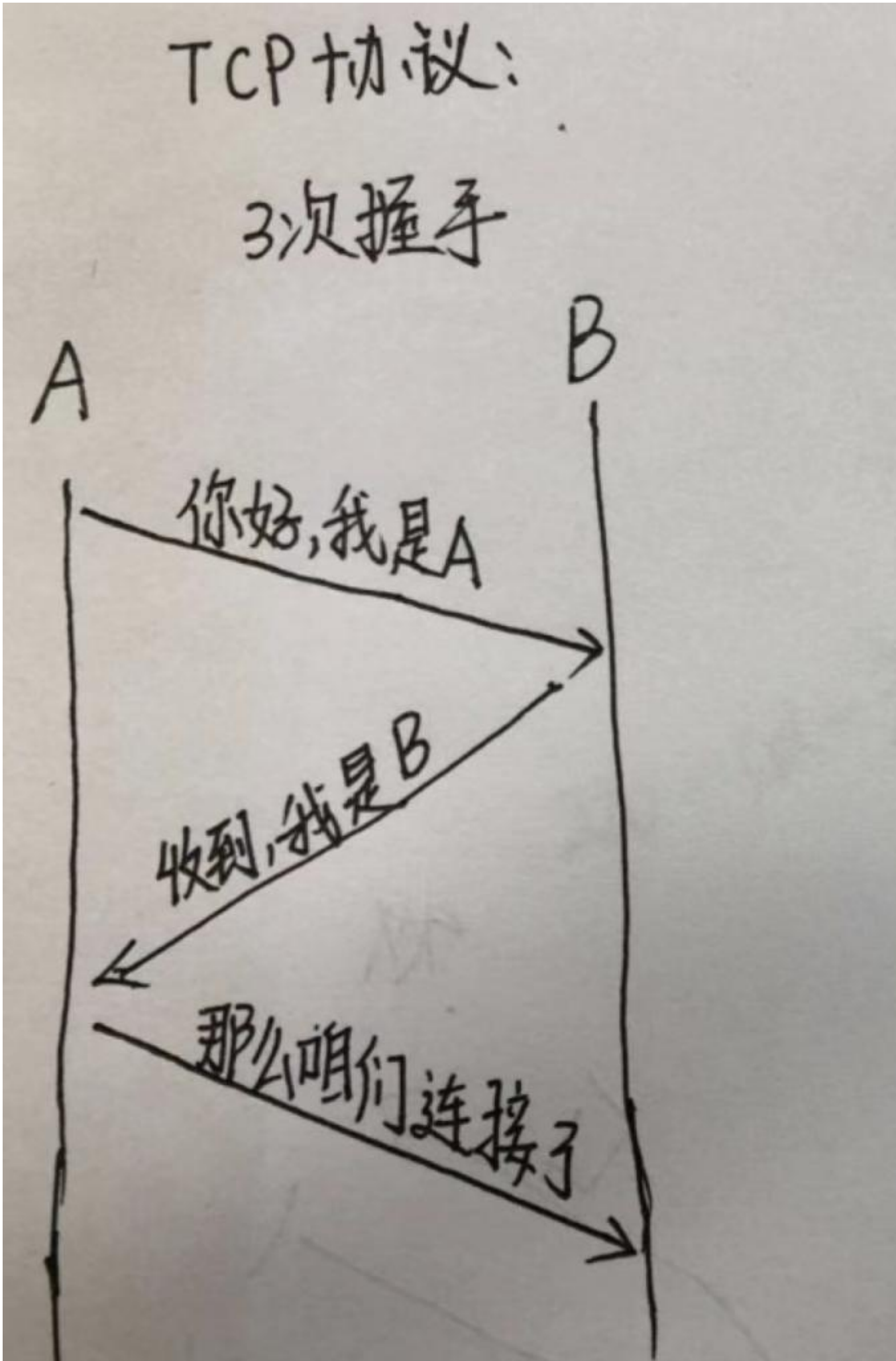
在 HTTP1.x 中，并发多个请求需要多个 TCP 连接，浏览器为了控制资源会有 6-8 个 TCP 连接都限制。

HTTP2 中

同域名下所有通信都在单个连接上完成，消除了因多个 TCP 连接而带来的延时和内存消耗。

单个连接上可以并行交错的请求和响应，之间互不干扰

第 16 题：谈谈你对 TCP 三次握手和四次挥手的理


TCP协议：
3次握手
A
B
你好，我是A
收到，我是B
那么咱们连接了

第 17 题：A、B 机器正常连接后，B 机器突然重启，问 A 此时处于 TCP 什么状态

如果 A 与 B 建立了正常连接后，从未相互发过数据，这个时候 B 突然机器重启，问 A 此时处于 TCP 什么状态？如何消除服务器程序中的这个状态？（超纲题，了解即可）
因为 B 会在重启之后进入 tcp 状态机的 listen 状态，只要当 a 重新发送一个数据包（无论是 syn 包或者是应用数据），b 端应该会主动发送一个带 rst 位的重置包来进行连接重置，所以 a 应该在 syn_sent 状态

第 18 题：React 中 setState 什么时候是同步的，什么时候是异步的？

1、由 React 控制的事件处理程序，以及生命周期函数调用 setState 不会同步更新 state 。
2、React 控制之外的事件中调用 setState 是同步更新的。比如原生 js 绑定的事件，setTimeout/setInterval 等。

第 19 题：React setState 笔试题，下面的代码输出什么？

```
class Example extends React.Component {
  constructor() {
    super()
      this.state = {
        val: 0
      }
    }
    componentDidMount() {
      this.setState({ val: this.state.val + 1 })
console.log(this.state.val)
    // 第 1 次 log
      this.setState({ val: this.state.val + 1 })
      console.log(this.state.val)
    // 第 2 次 log
      setTimeout(() => {
        this.setState({ val: this.state.val + 1 })
        console.log(this.state.val)
    // 第 3 次 log
        this.setState({ val: this.state.val + 1 })
        console.log(this.state.val)
    // 第 4 次 log
```

```
    }, 0)
  }
  render() {
    return null
  }
}
```

答：
0 0 2 3

第 20 题：介绍下 npm 模块安装机制，为什么输入 npm install 就可以自动安装对应的模块？1. npm 模块安装机制：

- 发出 npm install 命令 1 查询 node_modules 目录之中是否已经存在指定模块
- 若存在，不再重新安装
- 若不存在
- npm 向 registry 查询模块压缩包的网址
- 下载压缩包，存放在根目录下的.npm 目录里
- 解压压缩包到当前项目的 node_modules 目录

第 21 题：有以下 3 个判断数组的方法，请分别介绍它们之间的区别和优劣

Object.prototype.toString.call() 、 instanceof 以及 Array.isArray()

- **Object.prototype.toString.call()**

每一个继承 Object 的对象都有 toString 方法，如果 toString 方法没有重写的话，会返回 [Object type]，其中 type 为对象的类型。但当除了 Object 类型的对象外，其他类型直接使用 toString 方法时，会直接返回都是内容的字符串，所以我们需要使用 call 或者 apply 方法来改变 toString 方法的执行上下文。

```
const an = ['Hello','An'];an.toString();
 // "Hello,An"Object.prototype.toString.call(an);
 // "[object Array]"
```

这种方法对于所有基本的数据类型都能进行判断，即使是 null 和 undefined 。

```
Object.prototype.toString.call('An')
 // "[object String]"Object.prototype.toString.call(1)
 // "[object Number]"Object.prototype.toString.call(Symbol(1))
 // "[object Symbol]"Object.prototype.toString.call(null)
```

```
// "[object Null]"Object.prototype.toString.call(undefined)
// "[object Undefined]"Object.prototype.toString.call(function(){})
// "[object Function]"Object.prototype.toString.call({name: 'An'})
// "[object Object]"
```

Object.prototype.toString.call() 常用于判断浏览器内置对象时。

- **instanceof**

instanceof 的内部机制是通过判断对象的原型链中是不是能找到类型的 prototype。
使用 instanceof 判断一个对象是否为数组，instanceof 会判断这个对象的原型链上是否会找到对应的 Array 的原型，找到返回 true，否则返回 false。

```
[] instanceof Array; // true
```

但 instanceof 只能用来判断对象类型，原始类型不可以。并且所有对象类型 instanceof Object 都是 true。

```
[] instanceof Object; // true
```

- **Array.isArray()**

功能：用来判断对象是否为数组
instanceof 与 isArray
当检测 Array 实例时，Array.isArray 优于 instanceof ，因为 Array.isArray 可以检测出 iframes

```
var iframe = document.createElement('iframe');
document.body.appendChild(iframe);xArray =
window.frames[window.frames.length-1].Array;
var arr = new xArray(1,2,3);
// [1,2,3]// Correctly checking for ArrayArray.isArray(arr);
 // trueObject.prototype.toString.call(arr);
// true
// Considered harmful, because doesn't work though iframesarr instanceof
Array;
// false
```

Array.isArray() 与 Object.prototype.toString.call()
Array.isArray()是 ES5 新增的方法，当不存在 Array.isArray() ，可以用 Object.prototype.toString.call() 实现。

```
if (!Array.isArray) {  Array.isArray = function(arg) {
    return Object.prototype.toString.call(arg) === '[object Array]';
      };
    }
```

第 22 题：介绍下重绘和回流（Repaint & Reflow），以及如何进行优化

1. 浏览器渲染机制

浏览器采用流式布局模型（Flow Based Layout）
浏览器会把 HTML 解析成 DOM，把 CSS 解析成 CSSOM，DOM 和 CSSOM 合并就产生了渲染树（Render Tree）。
有了 RenderTree，我们就知道了所有节点的样式，然后计算他们在页面上的大小和位置，最后把节点绘制到页面上。
由于浏览器使用流式布局，对 Render Tree 的计算通常只需要遍历一次就可以完成，但 table 及其内部元素除外，他们可能需要多次计算，通常要花 3 倍于同等元素的时间，这也是为什么要避免使用 table 布局的原因之一。

2. 重绘

由于节点的几何属性发生改变或者由于样式发生改变而不会影响布局的，称为重绘，例如 outline, visibility, color、background-color 等，重绘的代价是高昂的，因为浏览器必须验证 DOM 树上其他节点元素的可见性。

3. 回流

回流是布局或者几何属性需要改变就称为回流。回流是影响浏览器性能的关键因素，因为其变化涉及到部分页面（或是整个页面）的布局更新。一个元素的回流可能会导致了其所有子元素以及 DOM 中紧随其后的节点、祖先节点元素的随后的回流。

```
<body><div class="error">
    <h4>我的组件</h4>
        <p><strong>错误：</strong>错误的描述…</p>
        <h5>错误纠正</h5>
        <ol>
            <li>第一步</li>
            <li>第二步</li>
        </ol></div></body>
```

在上面的 HTML 片段中，对该段落(<p>标签)回流将会引发强烈的回流，因为它是一个子节点。这也导致了祖先的回流（div.error 和 body – 视浏览器而定）。此外，<h5>和<ol>也会有简单的回流，因为其在 DOM 中在回流元素之后。大部分的回流将导致页面的重新渲染。
回流必定会发生重绘，重绘不一定会引发回流。

4. 浏览器优化

现代浏览器大多都是通过队列机制来批量更新布局，浏览器会把修改操作放在队列中，至少一个浏览器刷新（即 16.6ms）才会清空队列，但当你获取布局信息的时候，队列中可能有会影响这些属性或方法返回值的操作，即使没有，浏览器也会强制清空队列，触发回流与重绘来确保返回正确的值。

主要包括以下属性或方法：

1、offsetTop、offsetLeft、offsetWidth、offsetHeight
2、scrollTop、scrollLeft、scrollWidth、scrollHeight
3、clientTop、clientLeft、clientWidth、clientHeight
4、width、height
5、getComputedStyle()
6、getBoundingClientRect()

所以，我们应该避免频繁的使用上述的属性，他们都会强制渲染刷新队列。

5. 减少重绘与回流

CSS

1、使用 transform 替代 top
2、使用 visibility 替换 display: none ，因为前者只会引起重绘，后者会引发回流（改变了布局
3、避免使用 table 布局，可能很小的一个小改动会造成整个 table 的重新布局。
4、尽可能在 DOM 树的最末端改变 class，回流是不可避免的，但可以减少其影响。尽可能在 DOM 树的最末端改变 class，可以限制了回流的范围，使其影响尽可能少的节点。
5、避免设置多层内联样式，CSS 选择符从右往左匹配查找，避免节点层级过多。

```
<div>
  <a> <span></span> </a></div><style>
  span {
    color: red;
      }  div > a > span {
        color: red;
      }</style>
```

对于第一种设置样式的方式来说，浏览器只需要找到页面中所有的 span 标签然后设置颜色，但是对于第二种设置样式的方式来说，浏览器首先需要找到所有的 span 标签，然后找到 span 标签上的 a 标签，最后再去找到 div 标签，然后给符合这种条件的 span 标签设置颜色，这样的递归过程就很复杂。所以我们应该尽可能的避免写过于具体的 CSS 选择器，然后对于 HTML 来说也尽量少的添加无意义标签，保证层级扁平。

将动画效果应用到 position 属性为 absolute 或 fixed 的元素上，避免影响其他元素的布局，这样只是一个重绘，而不是回流，同时，控制动画速度可以选择 requestAnimationFrame，详见探讨 requestAnimationFrame。

避免使用 CSS 表达式，可能会引发回流。

将频繁重绘或者回流的节点设置为图层，图层能够阻止该节点的渲染行为影响别的节点，例如 will-change、video、iframe 等标签，浏览器会自动将该节点变为图层。

CSS3 硬件加速（GPU 加速），使用 css3 硬件加速，可以让 transform、opacity、filters 这些动画不会引起回流重绘 。但是对于动画的其它属性，比如 background-color 这些，还是会引起回流重绘的，不过它还是可以提升这些动画的性能。

JavaScript

1、避免频繁操作样式，最好一次性重写 style 属性，或者将样式列表定义为 class 并一次性更改 class 属性。

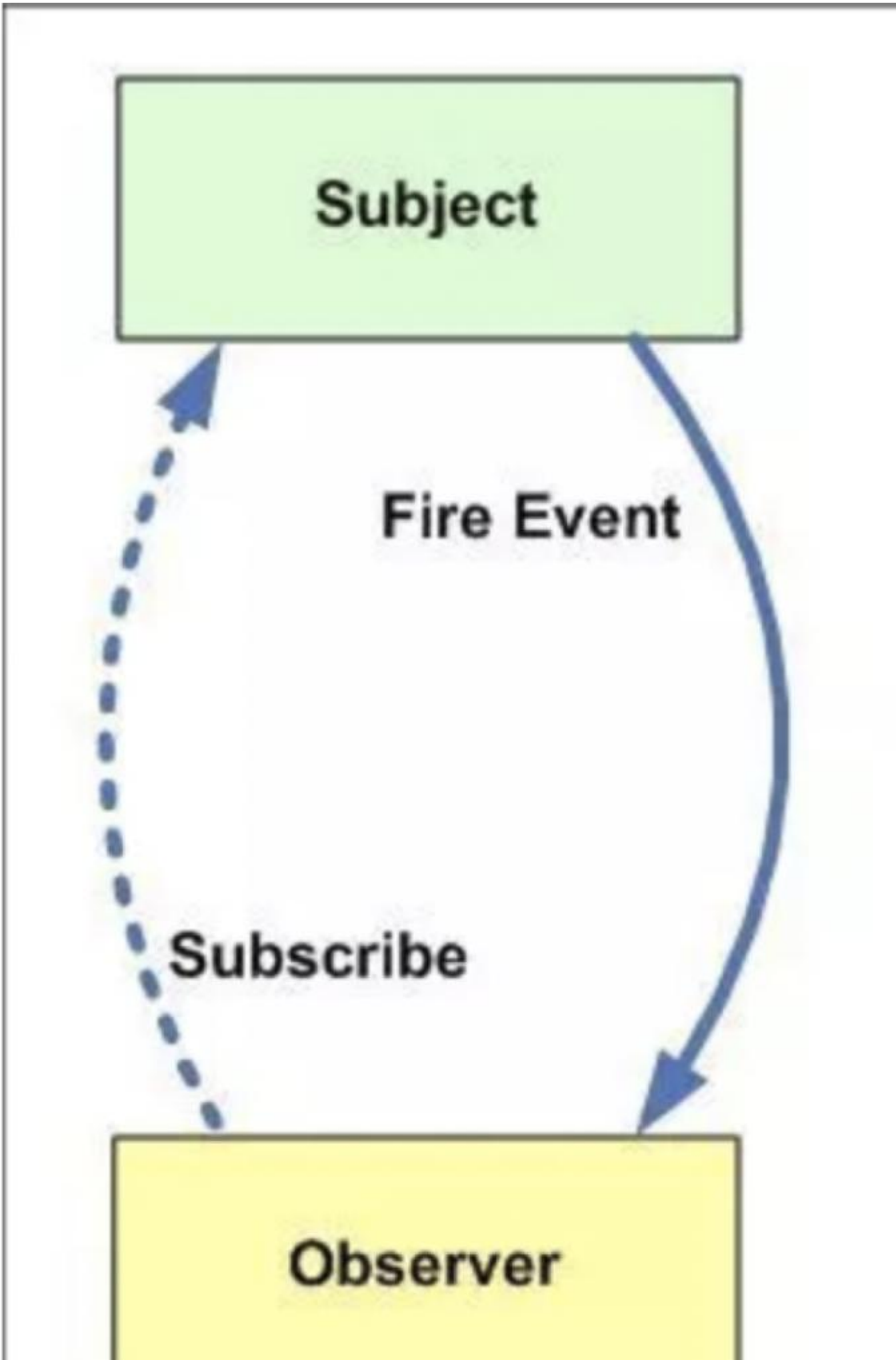
2、避免频繁操作 DOM，创建一个 documentFragment，在它上面应用所有 DOM 操作，最后再把它添加到文档中。

3、避免频繁读取会引发回流/重绘的属性，如果确实需要多次使用，就用一个变量缓存起来。

4、对具有复杂动画的元素使用绝对定位，使它脱离文档流，否则会引起父元素及后续元素频繁回流。

第 23 题：介绍下观察者模式和订阅-发布模式的区别，各自适用于什么场景

观察者模式中主体和观察者是互相感知的，发布-订阅模式是借助第三方来实现调度的，发布者和订阅者之间互不感知


Subject
Fire Event
Subscribe
Observer

第 24 题：聊聊 Redux 和 Vuex 的设计思想

不管是 Vue，还是 React，都需要管理状态（state），比如组件之间都有共享状态的需要。什么是共享状态？比如一个组件需要使用另一个组件的状态，或者一个组件需要改变另一个组件的状态，都是共享状态。
父子组件之间，兄弟组件之间共享状态，往往需要写很多没有必要的代码，比如把状态提升到父组件里，或者给兄弟组件写一个父组件，听听就觉得挺啰嗦。如果不对状态进行有效的管理，状态在什么时候，由于什么原因，如何变化就会不受控制，就很难跟踪和测试了。如果没有经历过这方面的困扰，可以简单理解为会搞得很乱就对了。
在软件开发里，有些通用的思想，比如隔离变化，约定优于配置等，隔离变化就是说做好抽象，把一些容易变化的地方找到共性，隔离出来，不要去影响其他的代码。约定优于配置就是很多东西我们不一定要写一大堆的配置，比如我们几个人约定，view 文件夹里只能放视图，不能放过滤器，过滤器必须放到 filter 文件夹里，那这就是一种约定，约定好之后，我们就不用写一大堆配置文件了，我们要找所有的视图，直接从 view 文件夹里找就行。
根据这些思想，对于状态管理的解决思路就是：把组件之间需要共享的状态抽取出来，遵循特定的约定，统一来管理，让状态的变化可以预测。根据这个思路，产生了很多的模式和库，我们来挨个聊聊。

第 25 题：说说浏览器和 Node 事件循环的区别

其中一个主要的区别在于浏览器的 event loop 和 nodejs 的 event loop 在处理异步事件的顺序是不同的,nodejs 中有 micro event;其中 Promise 属于 micro event 该异步事件的处理顺序就和浏览器不同.nodejs V11.0 以上 这两者之间的顺序就相同了.

```
function test () {
   console.log('start')
    setTimeout(() => {
          console.log('children2')
          Promise.resolve().then(() =>
 {console.log('children2-1')})
      }, 0)
      setTimeout(() => {
          console.log('children3')
          Promise.resolve().then(() =>
 {console.log('children3-1')})
```

```
    }, 0)
    Promise.resolve().then(() =>
{console.log('children1')})
    console.log('end')
}test()// 以上代码在 node11 以下版本的执行结果(先执行所有的宏任务，再执行微任务)// start// end// children1// children2// children3// children2-1// children3-1// 以上代码在 node11 及浏览器的执行结果(顺序执行宏任务和微任务)// start// end// children1// children2// children2-1// children3// children3-1
```

第 26 题：介绍模块化发展历程

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

https://blog.csdn.net/dadadeganhuo/article/details/86777249

第 27 题：全局作用域中，用 const 和 let 声明的变量不在 window 上，那到底在哪里？如何去获取？。

在 ES5 中，顶层对象的属性和全局变量是等价的，var 命令和 function 命令声明的全局变量，自然也是顶层对象。

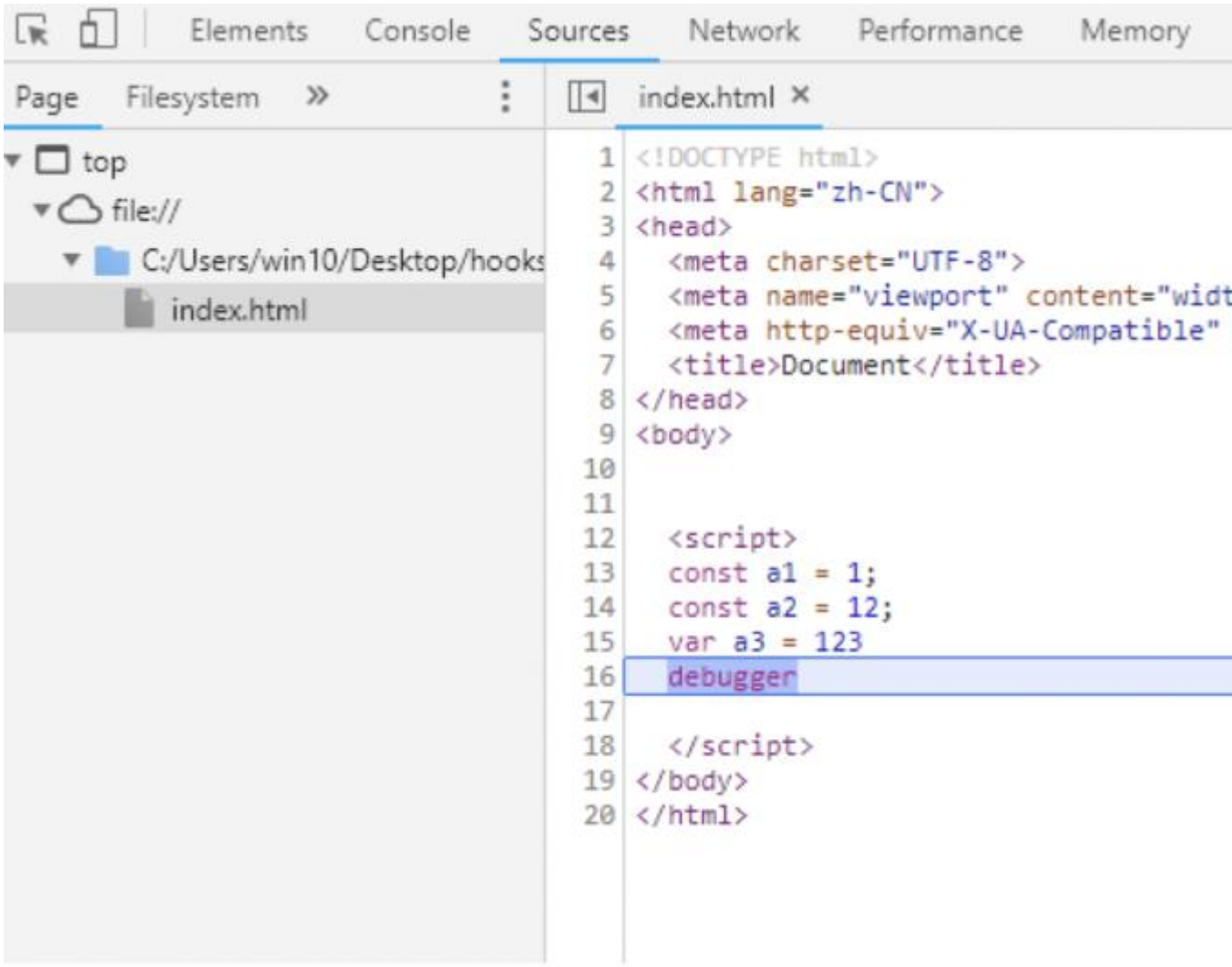
```
var a = 12;
function f(){};
console.log(window.a);
 // 12console.log(window.f);
 // f(){}
```

但 ES6 规定，var 命令和 function 命令声明的全局变量，依旧是顶层对象的属性，但 let 命令、const 命令、class 命令声明的全局变量，不属于顶层对象的属性。

```
let aa = 1;
const bb = 2;
console.log(window.aa);
 // undefinedconsole.log(window.bb);
```

```
// undefined
```

在哪里？怎么获取？通过在设置断点，看看浏览器是怎么处理的：



通过上图也可以看到，在全局作用域中，用 let 和 const 声明的全局变量并没有在全局对象中，只是一个块级作用域（Script）中
怎么获取？在定义变量的块级作用域中就能获取啊，既然不属于顶层对象，那就不加 window（global）呗。

```
let aa = 1;
const bb = 2;
console.log(aa);
 // 1console.log(bb);
 // 2
```

第 28 题：cookie 和 token 都存放在 header 中，为什么不会劫持 token？

1、攻击者通过 xss 拿到用户的 cookie 然后就可以伪造 cookie 了。
2、或者通过 csrf 在同个浏览器下面通过浏览器会自动带上 cookie 的特性
在通过 用户网站-攻击者网站-攻击者请求用户网站的方式 浏览器会自动带上 cookie
但是 token
1、不会被浏览器带上 问题 2 解决
2、token 是放在 jwt 里面下发给客户端的 而且不一定存储在哪里 不能通过 document.cookie 直接拿到，通过 jwt+ip 的方式 可以防止 被劫持 即使被劫持也是无效的 jwt

第 29 题：聊聊 Vue 的双向数据绑定，Model 如何改变 View，View 又是如何改变 Model 的

1、从 M 到 V 的映射（Data Binding），这样可以大量节省你人肉来 update View 的代码
2、从 V 到 M 的事件监听（DOM Listeners），这样你的 Model 会随着 View 触发事件而改变

第 30 题：两个数组合并成一个数组

请把两个数组 ['A1', 'A2', 'B1', 'B2', 'C1', 'C2', 'D1', 'D2'] 和 ['A', 'B', 'C', 'D']，合并为 ['A1', 'A2', 'A', 'B1', 'B2', 'B', 'C1', 'C2', 'C', 'D1', 'D2', 'D']。

```
function concatArr (arr1, arr2) { const arr = [...arr1];
 let currIndex = 0;
 for (let i = 0; i < arr2.length; i++) {
    const RE = new RegExp(arr2[i])
        while(currIndex < arr.length) {
         ++currIndex
          if (!RE.test(arr[currIndex])) {
             arr.splice(currIndex, 0, a2[i])
             break;
           }
         }
```

```
    }
    return arr } var a1 = ['A1', 'A2', 'B1', 'B2', 'C1', 'C2', 'D1',
'D2']
    var a2 = ['A', 'B', 'C', 'D'] const arr = concatArr(a1, a2)
    console.log(a1)
   // ['A1', 'A2', 'B1', 'B2', 'C1', 'C2', 'D1', 'D2'] console.log(a2)
// ['A', 'B', 'C', 'D'] console.log(arr)
   // ['A1', 'A2', 'A', B1', 'B2', 'B', C1', 'C2', 'C', D1', 'D2', 'D']
```

第 31 题：改造下面的代码，使之输出 0 - 9，写出你能想到的所有解法。

```
for (var i = 0;
 i< 10; i++){
  setTimeout(() => {
      console.log(i);
   }, 1000)}
```

答：

```
// 解法一：for (let i = 0;
 i< 10;
 i++){
  setTimeout(() => {
    console.log(i);
     }, 1000)}
    // 解法二：for (var i = 0;
     i< 10; i++){  ((i) => {
        setTimeout(() => {
          console.log(i);
        }, 1000) })(i)}
```

第 32 题：Virtual DOM 真的比操作原生 DOM 快吗？谈谈你的想法。

1. 原生 DOM 操作 vs. 通过框架封装操作。

这是一个性能 vs. 可维护性的取舍。框架的意义在于为你掩盖底层的 DOM 操作，让你用更声明式的方式来描述你的目的，从而让你的代码更容易维护。

没有任何框架可以比纯手动的优化 DOM 操作更快，因为框架的 DOM 操作层需要应对任何上层 API 可能产生的操作，它的实现必须是普适的。针对任何一个 benchmark，我都可以写出比任何框架更快的手动优化，但是那有什么意义呢？在构建一个实际应用的时候，你难道为每一个地方都去做手动优化吗？出于可维护性的考虑，这显然不可能。框架给你的保证是，你在不需要手动优化的情况下，我依然可以给你提供过得去的性能。

2. 对 React 的 Virtual DOM 的误解。

React 从来没有说过 “React 比原生操作 DOM 快”。React 的基本思维模式是每次有变动就整个重新渲染整个应用。如果没有 Virtual DOM，简单来想就是直接重置 innerHTML。很多人都没有意识到，在一个大型列表所有数据都变了的情况下，重置 innerHTML 其实是一个还算合理的操作... 真正的问题是在 “全部重新渲染” 的思维模式下，即使只有一行数据变了，它也需要重置整个 innerHTML，这时候显然就有大量的浪费。

我们可以比较一下 innerHTML vs. Virtual DOM 的重绘性能消耗：

- innerHTML: render html string O(template size) + 重新创建所有 DOM 元素 O(DOM size)
- Virtual DOM: render Virtual DOM + diff O(template size) + 必要的 DOM 更新 O(DOM change)

Virtual DOM render + diff 显然比渲染 html 字符串要慢，但是！它依然是纯 js 层面的计算，比起后面的 DOM 操作来说，依然便宜了太多。可以看到，innerHTML 的总计算量不管是 js 计算还是 DOM 操作都是和整个界面的大小相关，但 Virtual DOM 的计算量里面，只有 js 计算和界面大小相关，DOM 操作是和数据的变动量相关的。前面说了，和 DOM 操作比起来，js 计算是极其便宜的。这才是为什么要有 Virtual DOM：它保证了 1）不管你的数据变化多少，每次重绘的性能都可以接受；2) 你依然可以用类似 innerHTML 的思路去写你的应用。

3. MVVM vs. Virtual DOM

相比起 React，其他 MVVM 系框架比如 Angular, Knockout 以及 Vue、Avalon 采用的都是数据绑定：通过 Directive/Binding 对象，观察数据变化并保留对实际 DOM 元素的引用，当有数据变化时进行对应的操作。MVVM 的变化检查是数据层面的，而 React 的检查是 DOM 结构层面的。

MVVM 的性能也根据变动检测的实现原理有所不同：Angular 的脏检查使得任何变动都有固定的 O(watcher count) 的代价；Knockout/Vue/Avalon 都采用了依赖收集，在 js 和 DOM 层面都是 O(change)：

- 脏检查：scope digest O(watcher count) + 必要 DOM 更新 O(DOM change)

- 依赖收集：重新收集依赖 O(data change) + 必要 DOM 更新 O(DOM change)可以看到，Angular 最不效率的地方在于任何小变动都有的和 watcher 数量相关的性能代价。但是！当所有数据都变了的时候，Angular 其实并不吃亏。依赖收集在初始化和数据变化的时候都需要重新收集依赖，这个代价在小量更新的时候几乎可以忽略，但在数据量庞大的时候也会产生一定的消耗。

MVVM 渲染列表的时候，由于每一行都有自己的数据作用域，所以通常都是每一行有一个对应的 ViewModel 实例，或者是一个稍微轻量一些的利用原型继承的 "scope" 对象，但也有一定的代价。所以，MVVM 列表渲染的初始化几乎一定比 React 慢，因为创建 ViewModel / scope 实例比起 Virtual DOM 来说要昂贵很多。
这里所有 MVVM 实现的一个共同问题就是在列表渲染的数据源变动时，尤其是当数据是全新的对象时，如何有效地复用已经创建的 ViewModel 实例和 DOM 元素。
假如没有任何复用方面的优化，由于数据是“全新”的，MVVM 实际上需要销毁之前的所有实例，重新创建所有实例，最后再进行一次渲染！这就是为什么题目里链接的 angular/knockout 实现都相对比较慢。相比之下，React 的变动检查由于是 DOM 结构层面的，即使是全新的数据，只要最后渲染结果没变，那么就不需要做无用功。

Angular 和 Vue 都提供了列表重绘的优化机制，也就是“提示”框架如何有效地复用实例和 DOM 元素。比如数据库里的同一个对象，在两次前端 API 调用里面会成为不同的对象，但是它们依然有一样的 uid。这时候你就可以提示 track by uid 来让 Angular 知道，这两个对象其实是同一份数据。那么原来这份数据对应的实例和 DOM 元素都可以复用，只需要更新变动了的部分。或者，你也可以直接 track by $index 来进行“原地复用”：直接根据在数组里的位置进行复用。在题目给出的例子里，如果 angular 实现加上 track by $index 的话，后续重绘是不会比 React 慢多少的。甚至在 dbmonster 测试中，Angular 和 Vue 用了 track by $index 以后都比 React 快: dbmon (注意 Angular 默认版本无优化，优化过的在下面）
顺道说一句，React 渲染列表的时候也需要提供 key 这个特殊 prop，本质上和 track-by 是一回事。

4. 性能比较也要看场合
在比较性能的时候，要分清楚初始渲染、小量数据更新、大量数据更新这些不同的场合。Virtual DOM、脏检查 MVVM、数据收集 MVVM 在不同场合各有不同的表现和不同的优化需求。Virtual DOM 为了提升小量数据更新时的性能，也需要针对性的优化，比如 shouldComponentUpdate 或是 immutable data。

- 初始渲染：Virtual DOM > 脏检查 >= 依赖收集
- 小量数据更新：依赖收集 >> Virtual DOM + 优化 > 脏检查（无法优化） > Virtual DOM 无优化

- 大量数据更新：脏检查 + 优化 >= 依赖收集 + 优化 > Virtual DOM（无法/无需优化）>> MVVM 无优化

不要天真地以为 Virtual DOM 就是快，diff 不是免费的，batching 么 MVVM 也能做，而且最终 patch 的时候还不是要用原生 API。在我看来 Virtual DOM 真正的价值从来都不是性能，而是它 1) 为函数式的 UI 编程方式打开了大门；2) 可以渲染到 DOM 以外的 backend，比如 ReactNative。

5. 总结

以上这些比较，更多的是对于框架开发研究者提供一些参考。主流的框架 + 合理的优化，足以应对绝大部分应用的性能需求。如果是对性能有极致需求的特殊情况，其实应该牺牲一些可维护性采取手动优化：比如 Atom 编辑器在文件渲染的实现上放弃了 React 而采用了自己实现的 tile-based rendering；又比如在移动端需要 DOM-pooling 的虚拟滚动，不需要考虑顺序变化，可以绕过框架的内置实现自己搞一个。

第 33 题：下面的代码打印什么内容，为什么？

```
var b = 10;
(function b(){
    b = 20;
        console.log(b);
 })();
```

答：

```
f b(){
    b = 20;
        console.log(b);
 }
```

首先函数声明比变量要高，其次 b = 20 没有 var 获取其他，说明是 window 最外层定义的变量。

js 作用域中，先找最近的 那就是 b fn ，直接打印了，如果 b = 20 有 var 那就是打印 20

第 34 题：简单改造下面的代码，使之分别打印 10 和 20。

```
var b = 10;
(function b(){
    b = 20;
        console.log(b);
     })();
```

答：

```
var b = 10;
(function (){
  b = 20;
  console.log(b);
// 20})();
var b = 10;
(function (){
  console.log(b);
 // 10  b = 20;
})();
```

第 35 题：浏览器缓存读取规则

可以分成 Service Worker、Memory Cache、Disk Cache 和 Push Cache，那请求的时候 from memory cache 和 from disk cache 的依据是什么，哪些数据什么时候存放在 Memory Cache 和 Disk Cache 中？

https://www.jianshu.com/p/54cc04190252

第 36 题：使用迭代的方式实现 flatten 函数。

```
var arr=[1,2,3,[4,5],[6,[7,[8]]]]/** * 使用递归的方式处理 * wrap 内保存结果 ret  * 返回一个递归函数 * * @returns */function wrap(){
    var ret=[];
        return function flat(a){
            for(var item of
a){             if(item.constructor===Array){
                    ret.concat(flat(item))
                }else{
                    ret.push(item)
```

```
            }
        }
        return ret
    }}console.log(wrap()(arr));
```

第 37 题:为什么 Vuex 的 mutation 和 Redux 的 reducer 中不能做异步操作?

Mutation 必须是同步函数一条重要的原则就是要记住 mutation 必须是同步函数。为什么？请参考下面的例子：

```
mutations: {  someMutation (state) {    api.callAsyncMethod(() =>
{      state.count++    })  }}
```

现在想象,我们正在 debug 一个 app 并且观察 devtool 中的 mutation 日志。每一条 mutation 被记录，devtools 都需要捕捉到前一状态和后一状态的快照。然而，在上面的例子中 mutation 中的异步函数中的回调让这不可能完成：因为当 mutation 触发的时候，回调函数还没有被调用，devtools 不知道什么时候回调函数实际上被调用——实质上任何在回调函数中进行的状态的改变都是不可追踪的。
在组件中提交 Mutation 你可以在组件中使用 this.$store.commit('xxx') 提交 mutation，或者使用 mapMutations 辅助函数将组件中的 methods 映射为 store.commit 调用（需要在根节点注入 store）。

```
import { mapMutations } from 'vuex'export default {
  // ...  methods: {
    ...mapMutations([
        'increment', // 将 `this.increment()` 映射为
`this.$store.commit('increment')`
        // `mapMutations` 也支持载荷:
        'incrementBy' // 将 `this.incrementBy(amount)` 映射为
`this.$store.commit('incrementBy',
     amount)`
      ]),
      ...mapMutations({
        add: 'increment' // 将 `this.add()` 映射为
`this.$store.commit('increment')`
      })
    }
  }
```

第 38 题：（京东）下面代码中 a 在什么情况下会打印 1？

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

答：

```
var a = {
  i: 1,
  toString() {
    return a.i++;
      }}if( a == 1 && a == 2 && a == 3 ) {
      console.log(1);
    }let a = [1,2,3];
    a.toString = a.shift;if( a == 1 && a == 2 && a == 3 ) {
      console.log(1);}
```

第 39 题：介绍下 BFC 及其应用。

BFC 就是块级格式上下文，是页面盒模型布局中的一种 CSS 渲染模式，相当于一个独立的容器，里面的元素和外部的元素相互不影响。

创建 BFC 的方式有：

- html 根元素
- float 浮动
- 绝对定位
- overflow 不为 visiable
- display 为表格布局或者弹性布局

BFC 主要的作用是：

- 清除浮动
- 防止同一 BFC 容器中的相邻元素间的外边距重叠问题

第 40 题：在 Vue 中，子组件为何不可以修改父组件传递的 Prop

如果修改了，Vue 是如何监控到属性的修改并给出警告的。

1、子组件为何不可以修改父组件传递的 Prop 单向数据流，易于监测数据的流动，出现了错误可以更加迅速的定位到错误发生的位置。
2、如果修改了，Vue 是如何监控到属性的修改并给出警告的。

```
if (process.env.NODE_ENV !== 'production') {
      var hyphenatedKey = hyphenate(key);
      if (isReservedAttribute(hyphenatedKey) ||
config.isReservedAttr(hyphenatedKey)) {
        warn(
          ("\"" + hyphenatedKey + "\" is a reserved attribute and cannot
be used as component prop."),
          Vm
        );
      }
      defineReactive$$1(props, key, value, function () {
        if (!isRoot && !isUpdatingChildComponent)
 {
          warn(
            "Avoid mutating a prop directly since the value will be " +
"overwritten whenever the parent component re-renders. " +
"Instead, use a data or computed property based on the prop's " +
"value. Prop being mutated: \"" + key + "\"",
            Vm
          );
        }
      });
    }
```

在 initProps 的时候，在 defineReactive 时通过判断是否在开发环境，如果是开发环境，会在触发 set 的时候判断是否此 key 是否处于 updatingChildren 中被修改，如果不是，说明此修改来自子组件，触发 warning 提示。
需要特别注意的是，当你从子组件修改的 prop 属于基础类型时会触发提示。这种情况下，你是无法修改父组件的数据源的， 因为基础类型赋值时是值拷贝。你直接将另一个非基础类型（Object, array）赋值到此 key 时也会触发提示(但实际上不会影响父组件的数据源)， 当你修改 object 的属性时不会触发提示，并且会修改父组件数据源的数据。

第 41 题：下面代码输出什么

```
var a = 10;(function () {
    console.log(a)
        a = 5
        console.log(window.a)
        var a = 20;
        console.log(a)})()
```

分别为 undefined　10　20，原因是作用域问题，在内部声名 var a = 20;相当于先声明 var a;然后再执行赋值操作，这是在ＩＩＦＥ内形成的独立作用域，如果把 var a=20 注释掉，那么 a 只有在外部有声明，显示的就是外部的 A 变量的值了。结果 A 会是 10　5　5

第 42 题：实现一个 sleep 函数

比如 sleep(1000) 意味着等待 1000 毫秒，可从 Promise、Generator、Async/Await 等角度实现

```
const sleep = (time) => {
  return new Promise(resolve => setTimeout(resolve,
 time))}sleep(1000).then(() => {
    // 这里写你的骚操作})
```

第 43 题：使用 sort() 对数组 [3, 15, 8, 29, 102, 22] 进行排序，输出结果

输出：[102, 15, 22, 29, 3, 8]
解析：根据 MDN 上对 Array.sort()的解释，默认的排序方法会将数组元素转换为字符串，然后比较字符串中字符的 UTF-16 编码顺序来进行排序。所以'102' 会排在 '15' 前面。

第 44 题：介绍 HTTPS 握手过程

1、clientHello
2、SeverHello
3、客户端回应

4、服务器的最后回应

第 45 题：HTTPS 握手过程中，客户端如何验证证书的合法性

1. 校验证书的颁发机构是否受客户端信任。
2. 通过 CRL 或 OCSP 的方式校验证书是否被吊销。
3. 3 对比系统时间，校验证书是否在有效期内。
4. 通过校验对方是否存在证书的私钥，判断证书的网站域名是否与证书颁发的域名一致。

第 46 题：输出以下代码执行的结果并解释为什么

```
var obj = {
   '2': 3,
    '3': 4,
       'length': 2,
       'splice': Array.prototype.splice,
       'push':
    Array.prototype.push}obj.push(1)obj.push(2)console.log(obj)
```

结果：[,,1,2], length 为 4
伪数组（ArrayLike）

第 47 题：双向绑定和 vuex 是否冲突

在严格模式下直接使用确实会有问题。解决方案：

```
<input v-model="message" />computed: {
    message: {
           set (value)
     {
               this.$store.dispatch('updateMessage', value);
           },
```

```
        get () {
            Return
 this.$store.state.obj.message
        }
    }}mutations: {
    UPDATE_MESSAGE (state, v) {
        state.obj.message = v;
    }}actions: {
    update_message ({ commit }, v) {
        commit('UPDATE_MESSAGE', v);
    }
}
```

第 48 题：call 和 apply 的区别是什么，哪个性能更好一些

1. Function.prototype.apply 和 Function.prototype.call 的作用是一样的，区别在于传入参数的不同；
2. 第一个参数都是，指定函数体内 this 的指向；
3. 3 第二个参数开始不同，apply 是传入带下标的集合，数组或者类数组，apply 把它传给函数作为参数，call 从第二个开始传入的参数是不固定的，都会传给函数作为参数。
4. call 比 apply 的性能要好，平常可以多用 call, call 传入参数的格式正是内部所需要的格式

第 49 题:为什么通常在发送数据埋点请求的时候使用的是 1x1 像素的透明 gif 图片？

1. 没有跨域问题，一般这种上报数据，代码要写通用的；（排除 ajax）
2. 不会阻塞页面加载，影响用户的体验，只要 new Image 对象就好了；（排除 JS/CSS 文件资源方式上报）
3. 在所有图片中，体积最小；（比较 PNG/JPG）

第 50 题：（百度）实现 (5).add(3).minus(2) 功能。

例： 5 + 3 - 2，结果为 6
答：

```
Number.prototype.add = function(n)
 {  return this.valueOf() + n;
};Number.prototype.minus = function(n) {
  return this.valueOf() - n;
};
```

第 51 题：Vue 的响应式原理中 Object.defineProperty 有什么缺陷？

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

- Object.defineProperty 无法监控到数组下标的变化，导致通过数组下标添加元素，不能实时响应；
- Object.defineProperty 只能劫持对象的属性，从而需要对每个对象，每个属性进行遍历，如果，属性值是对象，还需要深度遍历。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。

第 52 题：怎么让一个 div 水平垂直居中

```
<div class="parent">
  <div class="child"></div></div>
```

一、

```
div.parent {
    display: flex;
        justify-content: center;
        align-items: center;
    }
```

二、

```
div.parent {
    position: relative;
```

```
}div.child {
    position: absolute;
     top: 50%;
    left: 50%;
    transform: translate(-50%, -50%);
  }/* 或者 */div.child {
    width: 50px;
    height: 10px;
    position: absolute;
    top: 50%;
    left: 50%;
    margin-left: -25px;
    margin-top: -5px;}/* 或 */div.child {
    width: 50px;
    height: 10px;
    position: absolute;
    left: 0;
    top: 0;
    right: 0;
    bottom: 0;
    margin: auto;
}
```

三、

```
div.parent {
    display: grid;
    }div.child {
        justify-self: center;
        align-self: center;
    }
```

四、

```
div.parent {
    font-size: 0;
        text-align: center;
        &::before {
            content: "";
           display: inline-block;
            width: 0;
            height: 100%;
            vertical-align: middle;
        }}div.child{
    display: inline-block;
    vertical-align: middle;
```

```
}
```

第 53 题：输出以下代码的执行结果并解释为什么

```
var a = {n: 1};
var b = a;a.x = a = {n: 2};
console.log(a.x)
    console.log(b.x)
```

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

第 54 题：冒泡排序如何实现，时间复杂度是多少， 还可以如何改进？

冒泡算法的原理：
升序冒泡： 两次循环，相邻元素两两比较，如果前面的大于后面的就交换位置
降序冒泡： 两次循环，相邻元素两两比较，如果前面的小于后面的就交换位置

js 实现：

```
// 升序冒泡 function maopao(arr){
  const array = [...arr]  for(let i = 0, len = array.length; i < len - 1; i++){
    for(let j = i + 1; j < len; j++) {
          if (array[i] > array[j]) {
            let temp = array[i]
            array[i] = array[j]
            array[j] = temp
```

```
        }
      }
    }
    return array
}
```

top | Filter

```
> function maopao(arr){
    const array = [...arr]
    for(let i = 0, len = array.length;
      for(let j = i + 1; j < len; j++)
        if (array[i] > array[j]) {
          let temp = array[i]
          array[i] = array[j]
          array[j] = temp
        }
      }
    }
    return array
  }
< undefined
> maopao([1,2,49,23,45,11,14])
< ▶(7) [1, 2, 11, 14, 23, 45, 49]
  maopao([1,6,4,5,3,2])
  ▶(6) [1, 2, 3, 4, 5, 6]
  maopao([1,2,3,4,5,6])
  ▶(6) [1, 2, 3, 4, 5, 6]
>
```

看起来没问题，不过一般生产环境都不用这个，原因是效率低下，冒泡排序在平均和最坏情况下的时间复杂度都是 O(n^2)，最好情况下都是 O(n)，空间复杂度是 O(1)。因为就算你给一个已经排好序的数组，如[1,2,3,4,5,6] 它也会走一遍流程，白白浪费资源。所以有没有什么好的解决方法呢？
答案是肯定有的：加个标识，如果已经排好序了就直接跳出循环。

优化版：

```
function maopao(arr){
  const array = [...arr]
  let isOk = true  for(let i = 0, len = array.length;
 i < len - 1; i++){
    for(let j = i + 1; j < len; j++) {
          if (array[i] > array[j]) {
            let temp = array[i]
            array[i] = array[j]
            array[j] = temp
            isOk = false
          }
        }
        if(isOk){
          Break
        }
      }
      return array}
```

测试： 数组：[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]

从测试结果来看： 普通冒泡排序

top

```
> function maopao(arr){
    const array = [...arr]
    console.time('time1')
    for(let i = 0, len = array.lengt
      for(let j = i + 1; j < len; j+
        if (array[i] > array[j]) {
          let temp = array[i]
          array[i] = array[j]
          array[j] = temp
        }
      }
    }
    console.timeEnd('time1')
    return array
  }

  // 优化版
  function maopao1(arr){
    const array = [...arr]
    let isOk = true
    console.time('time2')
    for(let i = 0, len = array.lengt
      for(let j = i + 1; j < len; j+
        if (array[i] > array[j]) {
```

时间：0.044ms 优化后冒泡排序时间：0.018ms

第 55 题：某公司 1 到 12 月份的销售额存在一个对象里面

如下：{1:222, 2:123, 5:888}，请把数据处理为如下结构：[222, 123, null, null, 888, null, null, null, null, null, null, null]。

```
let obj = {1:222, 2:123, 5:888};
const result = Array.from({ length: 12 }).map((_, index) => obj[index + 1] || null);
console.log(result)
```

第 56 题：要求设计 LazyMan 类，实现以下功能。

```
LazyMan('Tony');
// Hi I am Tony
LazyMan('Tony').sleep(10).eat('lunch');
// Hi I am Tony
// 等待了 10 秒...
// I am eating
lunchLazyMan('Tony').eat('lunch').sleep(10).eat('dinner');
// Hi I am Tony
// I am eating lunch// 等待了 10 秒...
// I am eating
dinerLazyMan('Tony').eat('lunch').eat('dinner').sleepFirst(5).sleep(10).eat('junk food');
// Hi I am Tony// 等待了 5 秒...
// I am eating lunch
// I am eating dinner
// 等待了 10 秒...
// I am eating junk food
```

答：

```
class LazyManClass {
  constructor(name) {
    this.name = name
        this.queue = []
        console.log(`Hi I am ${name}`)
        setTimeout(() => {
```

```
      this.next()
    },0)
  }
  sleepFirst(time) {
    const fn = () => {
      setTimeout(() => {
        console.log(`等待了${time}秒...`)
        this.next()
      }, time)
    }
    this.queue.unshift(fn)
    return this
  }
  sleep(time) {
    const fn = () => {
      setTimeout(() => {
        console.log(`等待了${time}秒...`)
        this.next()
      },time)
    }
    this.queue.push(fn)
    return this
  }
  eat(food) {
    const fn = () => {
      console.log(`I am eating ${food}`)
      this.next()
    }
    this.queue.push(fn)
    return this
  }
  next() {
    const fn = this.queue.shift()
    fn && fn()  }}function LazyMan(name) {
  return new LazyManClass(name)}
```

第 57 题:分析比较 opacity:0、visibility:hidden、display:none 优劣和适用场景。

- display: none (不占空间，不能点击)（场景，显示出原来这里不存在的结构）
- visibility: hidden（占据空间，不能点击）（场景：显示不会导致页面结构发生变动，不会撑开）
- opacity: 0（占据空间，可以点击）（场景：可以跟 transition 搭配）

第 58 题：箭头函数与普通函数（function）的区别是什么？构造函数（function）可以使用 new 生成实例，那么箭头函数可以吗？为什么？

箭头函数是普通函数的简写，可以更优雅的定义一个函数，和普通函数相比，有以下几点差异：

- 函数体内的 this 对象，就是定义时所在的对象，而不是使用时所在的对象。
- 不可以使用 arguments 对象，该对象在函数体内不存在。如果要用，可以用 rest 参数代替。
- 不可以使用 yield 命令，因此箭头函数不能用作 Generator 函数。
- 不可以使用 new 命令，因为：
  - 1.没有自己的 this，无法调用 call，apply。
  - 2.没有 prototype 属性 ，而 new 命令在执行时需要将构造函数的 prototype 赋值给新的对象的 __proto__

new 过程大致是这样的：

```
function newFunc(father, ...rest) {
  var result = {};
  result.__proto__ = father.prototype;
  var result2 = father.apply(result, rest);
  if (
    (typeof result2 === 'object' || typeof result2 === 'function') &&
result2 !== null
    ) {
      return result2;
    }
    return result;
  }
```

第 59 题：给定两个数组，写一个方法来计算它们的交集。

例如：给定 nums1 = [1, 2, 2, 1]，nums2 = [2, 2]，返回 [2, 2]。

```
var nums1 = [1, 2, 2, 1], nums2 = [2, 2, 3, 4];
// 1.
// 有个问题，
 [NaN].indexOf(NaN) === -1var newArr1 = nums1.filter(function(item) {
    return nums2.indexOf(item) > -1;
});
console.log(newArr1);
// 2.
var newArr2 = nums1.filter((item) => {
    return nums2.includes(item);
});
console.log(newArr2);
```

第 60 题：已知如下代码，如何修改才能让图片宽度为 300px ？

注意下面代码不可修改。

```
<img src="1.jpg" style="width:480px!important;" >
```

答：

```
max-width: 300pxtransform: scale(0.625,0.625)
```

第 61 题：介绍下如何实现 token 加密

jwt 举例：

1. 需要一个 secret（随机数）
2. 后端利用 secret 和加密算法(如：HMAC-SHA256)对 payload(如账号密码)生成一个字符串(token)，返回前端
3. 前端每次 request 在 header 中带上 token
4. 后端用同样的算法解密

第 62 题：redux 为什么要把 reducer 设计成纯函数

redux 的设计思想就是不产生副作用，数据更改的状态可回溯，所以 redux 中处处都是纯函数

第 63 题：如何设计实现无缝轮播

简单来说，无缝轮播的核心是制造一个连续的效果。最简单的方法就是复制一个轮播的元素，当复制元素将要滚到目标位置后，把原来的元素进行归位的操作，以达到无缝的轮播效果。

贴一段轮播的核心代码：

```
// scroll the notice
  useEffect(() => {
    const requestAnimationFrame =
          window.requestAnimationFrame ||
     window.webkitRequestAnimationFrame ||
     window.mozRequestAnimationFrame
        const cancelAnimationFrame =
          window.cancelAnimationFrame ||
     window.webkitCancelAnimationFrame ||
window.mozCancelAnimationFrame
        const scrollNode = noticeContentEl.current
        const distance = scrollNode.clientWidth / 2
        scrollNode.style.left = scrollNode.style.left || 0
        window.__offset = window.__offset || 0
        let requestId = null
        const scrollLeft = () => {
          const speed = 0.5
          window.__offset = window.__offset + speed
          scrollNode.style.left = -window.__offset + 'px'
          // 关键行：当距离小于偏移量时，重置偏移量
          if (distance <= window.__offset) window.__offset = 0
          requestId = requestAnimationFrame(scrollLeft)
        }    requestId = requestAnimationFrame(scrollLeft)
        if (pause) cancelAnimationFrame(requestId)
        return () => cancelAnimationFrame(requestId)
      }, [notice, pause])
```

第 64 题：模拟实现一个 Promise.finally

```
Promise.prototype.finally = function (callback) {
  let P = this.constructor;
  return this.then(
    value  => P.resolve(callback()).then(() => value),
       reason => P.resolve(callback()).then(()
 => { throw reason })
     );
    };
```

第 65 题： a.b.c.d 和 a['b']['c']['d']，哪个性能更高？

应该是 a.b.c.d 比 a['b']['c']['d'] 性能高点，后者还要考虑 [] 中是变量的情况，再者，从两种形式的结构来看，显然编译器解析前者要比后者容易些，自然也就快一点。

变量

```
▲ ast1: Object {program: Script, name: null, loc: O
    comments: null
  ▶ loc: Object {start: Object, end: Object, lines:
    name: null
  ▶ original: Object {program: Script, name: null, l
  ▶ program: Script {type: "Program", body: Array(1)
  ▲ tokens: Array(7) [Object, Object, Object, …]
      length: 7
    ▶ __proto__: Array(0) [, …]
    ▶ 0: Object {type: "Identifier", value: "a", loc:
    ▶ 1: Object {type: "Punctuator", value: ".", loc:
    ▶ 2: Object {type: "Identifier", value: "b", loc:
    ▶ 3: Object {type: "Punctuator", value: ".", loc:
    ▶ 4: Object {type: "Identifier", value: "c", loc:
    ▶ 5: Object {type: "Punctuator", value: ".", loc:
    ▶ 6: Object {type: "Identifier", value: "d", loc:
    type: "File"
  ▶ __proto__: Object {}
▲ ast2: Object {program: Script, name: null, loc: O
    comments: null
  ▶ loc: Object {start: Object, end: Object, lines:
    name: null
  ▶ original: Object {program: Script, name: null, l
  ▶ program: Script {type: "Program", body: Array(1)
```

第 66 题：ES6 代码转成 ES5 代码的实现思路是什么

ES6 转 ES5 目前行业标配是用 Babel，转换的大致流程如下：

1. 解析：解析代码字符串，生成 AST；
2. 转换：按一定的规则转换、修改 AST；
3. 生成：将修改后的 AST 转换成普通代码。

如果不用工具，纯人工的话，就是使用或自己写各种 polyfill 了。

第 67 题：数组编程题

随机生成一个长度为 10 的整数类型的数组，例如 [2, 10, 3, 4, 5, 11, 10, 11, 20]，将其排列成一个新数组，要求新数组形式如下，例如 [[2, 3, 4, 5], [10, 11], [20]]。

```
function formArray(arr: any[]) {
  const sortedArr = Array.from(new Set(arr)).sort((a, b) => a - b);
const map = new Map();
  sortedArr.forEach((v) => {
    const key = Math.floor(v / 10);
      const group = map.get(key) || [];
      group.push(v);
      map.set(key, group);
    });  return [...map.values()];}// 求连续的版本 function
formArray1(arr: any[]) {
    const sortedArr = Array.from(new Set(arr)).sort((a, b) => a - b);
    return sortedArr.reduce((acc, cur) => {
      const lastArr = acc.slice().pop() || [];
      const lastVal = lastArr.slice().pop();
      if (lastVal!=null && cur-lastVal === 1)
    {
        lastArr.push(cur);
      } else {
        acc.push([cur]);
      }
      return acc;
```

```
  }, []);}function genNumArray(num: number, base = 100) {
  return Array.from({length: num}, () =>
 Math.floor(Math.random()*base));
}const arr = genNumArray(10, 20);
 //[2, 10, 3, 4, 5, 11, 10, 11, 20];
const res = formArray(arr);console.log(`res
${JSON.stringify(res)}`);
```

第 68 题： 如何解决移动端 Retina 屏 1px 像素问题

1. 伪元素 + transform scaleY(.5)
2. border-image
3. background-image
4. box-shadow

第 69 题： 如何把一个字符串的大小写取反（大写变小写小写变大写），例如 ’ AbC’ 变成 ’aBc’ 。

```
function processString (s) {
    var arr = s.split('');
        var new_arr = arr.map((item) => {
            return item === item.toUpperCase() ? item.toLowerCase() :
item.toUpperCase();
        });
        Return
 new_arr.join('');}console.log(processString('AbC'));function
 swapString(str) {
  var result = ''
  for (var i = 0; i < str.length; i++) {
    var c = str[i]
    if (c === c.toUpperCase()) {
      result += c.toLowerCase()
    } else {
      result += c.toUpperCase()
    }
```

```
    }
    return result}swapString('ADasfads123!@$!@#') // =>
'adASFADS123!@$!@#'
```

第 70 题： 介绍下 webpack 热更新原理，是如何做到在不刷新浏览器的前提下更新页面的

1. 当修改了一个或多个文件；
2. 文件系统接收更改并通知 webpack；
3. webpack 重新编译构建一个或多个模块，并通知 HMR 服务器进行更新；
4. HMR Server 使用 webSocket 通知 HMR runtime 需要更新，HMR 运行时通过 HTTP 请求更新 jsonp；
5. HMR 运行时替换更新中的模块，如果确定这些模块无法更新，则触发整个页面刷新。

第 71 题： 实现一个字符串匹配算法，从长度为 n 的字符串 S 中，查找是否存在字符串 T，T 的长度是 m，若存在返回所在位置。

```
const find = (S, T) => {
  if (S.length < T.length) return -1
  for (let i = 0; i < S.length; i++) {
    if (S.slice(i, i + T.length) === T) return i
    }
    return -1
```

第 72 题： 为什么普通 for 循环的性能远远高于 forEach 的性能，请解释其中的原因。

Preparation code

```
<script>
let arrs = new Array(1000000);
</script>
```

Test runner

Done. Ready to run again.

Testing in Chrome	
Tes	

for	for (let i = 0; i < arrs.length;
	};

forEach	arrs.forEach((arr) => { });

for 循环没有任何额外的函数调用栈和上下文；
forEach 函数签名实际上是

```
array.forEach(function(currentValue, index, arr), thisValue)
```

它不是普通的 for 循环的语法糖，还有诸多参数和上下文需要在执行的时候考虑进来，这里可能拖慢性能；

第 73 题： 介绍下 BFC、IFC、GFC 和 FFC

BFC（Block formatting contexts）：
块级格式上下文页面上的一个隔离的渲染区域，那么他是如何产生的呢？可以触发 BFC 的元素有 float、position、overflow、display：table-cell/ inline-block/table-caption ； BFC 有什么作用呢？比如说实现多栏布局’

IFC（Inline formatting contexts）：
内联格式上下文 IFC 的 line box（线框）高度由其包含行内元素中最高的实际高度计算而来(不受到竖直方向的 padding/margin 影响)IFC 中的 line box 一般左右都贴紧整个 IFC,但是会因为 float 元素而扰乱。float 元素会位于 IFC 与与 line box 之间，使得 line box 宽度缩短。 同个 ifc 下的多个 line box 高度会不同 IFC 中时不可能有块级元素的，当插入块级元素时（如 p 中插入 div）会产生两个匿名块与 div 分隔开，即产生两个 IFC，每个 IFC 对外表现为块级元素，与 div 垂直排列。那么 IFC 一般有什么用呢？水平居中：当一个块要在环境中水平居中时，设置其为 inline-block 则会在外层产生 IFC,通过 text-align 则可以使其水平居中。垂直居中：创建一个 IFC，用其中一个元素撑开父元素的高度，然后设置其 vertical-align:middle，其他行内元素则可以在此父元素下垂直居中。

GFC（GrideLayout formatting contexts）：
网格布局格式化上下文当为一个元素设置 display 值为 grid 的时候，此元素将会获得一个独立的渲染区域，我们可以通过在网格容器（grid container）上定义网格定义行（grid definition rows）和网格定义列（grid definition columns）属性各在网格项目（grid item）上定义网格行（grid row）和网格列（grid columns）为每一个网格项目（grid item）定义位置和空间。那么 GFC 有什么用呢，和 table 又有什么区别呢？首先同样是一个二维的表格，但 GridLayout 会有更加丰富的属性来控制行列，控制对齐以及更为精细的渲染语义和控制。

FFC（Flex formatting contexts）：
自适应格式上下文 display 值为 flex 或者 inline-flex 的元素将会生成自适应容器（flex container），可惜这个牛逼的属性只有谷歌和火狐支持，不过在移动端也足够了，至少 safari 和 chrome 还是 OK 的，毕竟这俩在移动端才是王道。Flex Box 由伸缩容器和伸缩项目组成。通过设置元素的 display 属性为 flex 或

inline-flex 可以得到一个伸缩容器。设置为 flex 的容器被渲染为一个块级元素，而设置为 inline-flex 的容器则渲染为一个行内元素。伸缩容器中的每一个子元素都是一个伸缩项目。伸缩项目可以是任意数量的。伸缩容器外和伸缩项目内的一切元素都不受影响。简单地说，Flexbox 定义了伸缩容器内伸缩项目该如何布局。

第 74 题： 使用 JavaScript Proxy 实现简单的数据绑定

```
<div id="app">
    <input type="text" id="input">
        <div>
          TODO:
          <span id="text"></span>
        </div>    <div id="btn">Add To Todo List</div>
        <ul id="list"></ul>  </div>
        const input = document.getElementById('input')
        const text = document.getElementById('text')
        const list = document.getElementById('list')
        const btn = document.getElementById('btn')
        let render
        const inputObj = new Proxy({}, {
          get (target, key, receiver) {
            return Reflect.get(target, key, receiver)
          },
          set (target, key, value, receiver) {
            if (key === 'text') {
              input.value = value
              text.innerHTML = value
            }
            return Reflect.set(target, key, value,
     receiver)
          }
        })
        class Render {
          constructor (arr) {
            this.arr = arr
          }
          init () {
           const fragment = document.createDocumentFragment()
            for (let i = 0; i < this.arr.length; i++) {
              const li = document.createElement('li')
```

```
          li.textContent = this.arr[i]
          fragment.appendChild(li)
        }
        list.appendChild(fragment)
      }
      addList (val) {
        const li = document.createElement('li')
        li.textContent = val
        list.appendChild(li)
      }
    }
    const todoList = new Proxy([], {
      get (target, key, receiver) {
        return Reflect.get(target, key, receiver)
      },
      set (target, key, value, receiver) {
        if (key !== 'length')
{
          render.addList(value)
        }
        return Reflect.set(target, key, value,
receiver)
      }
    })
    window.onload = () => {
      render = new Render([])
      render.init()
    }
    input.addEventListener('keyup', e => {
      inputObj.text = e.target.value
    })
    btn.addEventListener('click', () =>
{
      todoList.push(inputObj.text)
      inputObj.text = ''
    })
```

第 75 题：数组里面有 10 万个数据，取第一个元素和第 10 万个元素的时间相差多少

数组可以直接根据索引取的对应的元素，所以不管取哪个位置的元素的时间复杂度都是 O(1)
得出结论：消耗时间几乎一致，差异可以忽略不计

第 76 题：输出以下代码运行结果

```
// example 1
var a={}, b='123', c=123;
  a[b]='b';a[c]='c';
  console.log(a[b]);
---------------------
// example 2var a={}, b=Symbol('123'), c=Symbol('123');
  a[b]='b';
a[c]='c';
  console.log(a[b]);
---------------------
// example 3var a={}, b={key:'123'}, c={key:'456'};
  a[b]='b';a[c]='c';
  console.log(a[b]);
```

答：

1. 对象的键名只能是字符串和 Symbol 类型。
2. 其他类型的键名会被转换成字符串类型。
3. 对象转字符串默认会调用 toString 方法。

```
// example 1
var a={}, b='123', c=123;a[b]='b';
// c 的键名会被转换成字符串'123'，这里会把 b 覆盖掉。a[c]='c';
  // 输出 cconsole.log(a[b]);
// example 2var a={}, b=Symbol('123'), c=Symbol('123');
  // b 是 Symbol 类型，不需要转换。a[b]='b';
// c 是 Symbol 类型，不需要转换。任何一个 Symbol 类型的值都是不相等的，所以不会覆盖掉 b。a[c]='c';
// 输出 bconsole.log(a[b]);
// example 3var a={}, b={key:'123'}, c={key:'456'};
  // b 不是字符串也不是 Symbol 类型，需要转换成字符串。
// 对象类型会调用 toString 方法转换成字符串 [object Object]。a[b]='b';
// c 不是字符串也不是 Symbol 类型，需要转换成字符串。
```

```
// 对象类型会调用 toString 方法转换成字符串 [object Object]。这里会把 b 覆盖掉。a[c]='c';
  // 输出 cconsole.log(a[b]);
```

第 77 题：算法题「旋转数组」

给定一个数组，将数组中的元素向右移动 k 个位置，其中 k 是非负数。
示例 1：
输入: [1, 2, 3, 4, 5, 6, 7] 和 k = 3 输出: [5, 6, 7, 1, 2, 3, 4] 解释: 向右旋转 1 步: [7, 1, 2, 3, 4, 5, 6] 向右旋转 2 步: [6, 7, 1, 2, 3, 4, 5] 向右旋转 3 步: [5, 6, 7, 1, 2, 3, 4]

示例 2：
输入: [-1, -100, 3, 99] 和 k = 2 输出: [3, 99, -1, -100] 解释: 向右旋转 1 步: [99, -1, -100, 3] 向右旋转 2 步: [3, 99, -1, -100]

答：

```
function rotate(arr, k) {
  const len = arr.length  const step = k % len  return arr.slice(-step).concat(arr.slice(0, len - step))}// rotate([1, 2, 3, 4, 5, 6], 7) => [6, 1, 2, 3, 4, 5]
```

第 78 题：Vue 的父组件和子组件生命周期钩子执行顺序是什么

1. 父组建： beforeCreate -> created -> beforeMount
2. 子组件： -> beforeCreate -> created -> beforeMount -> mounted
3. 父组件： -> mounted
4. 总结：从外到内，再从内到外

第 79 题：input 搜索如何防抖，如何处理中文输入

```
<div>
    <input type="text" id="ipt">
```

```
</div>  <script>
  let ipt = document.getElementById('ipt');
  let dbFun = debounce()
  ipt.addEventListener('keyup', function (e) {
    dbFun(e.target.value);
  })
  function debounce() {
    let timer;
    return function (value) {
      clearTimeout(timer);
      timer = setTimeout(() =>
{
        console.log(value)
      }, 500);
    }
  }
</script>
```

第 80 题：介绍下 Promise.all 使用、原理实现及错误处理

```
const p = Promise.all([p1, p2, p3]);
```

Promise.all 方法接受一个数组作为参数，p1、p2、p3 都是 Promise 实例，如果不是，就会先调用下面讲到的 Promise.resolve 方法，将参数转为 Promise 实例，再进一步处理。（Promise.all 方法的参数可以不是数组，但必须具有 Iterator 接口，且返回的每个成员都是 Promise 实例。）

第 81 题：打印出 1 - 10000 之间的所有对称数

例如：121、1331 等

```
[...Array(10000).keys()].filter((x) => {
   return x.toString().length > 1 && x ===
Number(x.toString().split('').reverse().join(''))
 })
```

第 82 题：周一算法题之「移动零」

给定一个数组 nums，编写一个函数将所有 0 移动到数组的末尾，同时保持非零元素的相对顺序。

示例:
输入: [0,1,0,3,12] 输出: [1,3,12,0,0]
复制代码说明: 必须在原数组上操作，不能拷贝额外的数组。 尽量减少操作次数
答：

```
function zeroMove(array) {
        let len = array.length;
        let j = 0;
        for(let
 i=0;i<len-j;i++){
            if(array[i]===0){
                array.push(0);
                array.splice(i,1);
                i --;
                j ++;
            }
        }
        return array;
    }
```

第 83 题：var、let 和 const 区别的实现原理是什么

三者的区别：

- var 和 let 用以声明变量，const 用于声明只读的常量；
- var 和 let 用以声明变量，const 用于声明只读的常量；
- var 声明的变量，不存在块级作用域，在全局范围内都有效，let 和 const 声明的，只在它所在的代码块内有效；
- let 和 const 不存在像 var 那样的“变量提升”现象，所以 var 定义变量可以先使用，后声明，而 let 和 const 只可先声明，后使用；
- let 声明的变量存在暂时性死区，即只要块级作用域中存在 let，那么它所声明的变量就绑定了这个区域，不再受外部的影响。
- let 不允许在相同作用域内，重复声明同一个变量；
- const 在声明时必须初始化赋值，一旦声明，其声明的值就不允许改变，更不允许重复声明；如 const 声明了一个复合类型的常量，其存储的是一个引用地址，不允许改变的是这个地址，而对象本身是可变的。

变量与内存之间的关系，主要由三个部分组成：

- 变量名
- 内存地址
- 内存空间

JS 引擎在读取变量时，先找到变量绑定的内存地址，然后找到地址所指向的内存空间，最后读取其中的内容。当变量改变时，JS 引擎不会用新值覆盖之前旧值的内存空间（虽然从写代码的角度来看，确实像是被覆盖掉了），而是重新分配一个新的内存空间来存储新值，并将新的内存地址与变量进行绑定，JS 引擎会在合适的时机进行 GC，回收旧的内存空间。
const 定义变量（常量）后，变量名与内存地址之间建立了一种不可变的绑定关系，阻隔变量地址被改变，当 const 定义的变量进行重新赋值时，根据前面的论述，JS 引擎会尝试重新分配新的内存空间，所以会被拒绝，便会抛出异常。

第 84 题：请实现一个 add 函数，满足以下功能。

```
add(1);
 // 1add(1)(2);
 // 3add(1)(2)(3);
// 6add(1)(2, 3);
// 6add(1, 2)(3);
 // 6add(1, 2, 3);
 // 6
```

答：
实现 1：

```
function currying(fn, length) {
  length = length || fn.length;     // 注释 1
  return function (...args) {           // 注释 2    return
args.length >= length    // 注释 3
        ? fn.apply(this, args)          // 注释 4
      : currying(fn.bind(this, ...args), length - args.length) // 注释
5  }}
```

实现 2：

```
const currying = fn =>
    judge = (...args) =>
            args.length >= fn.length
                ? fn(...args)
                : (...arg) => judge(...args, ...arg)
```

其中注释部分
注释 1：第一次调用获取函数 fn 参数的长度，后续调用获取 fn 剩余参数的长度
注释 2：currying 包裹之后返回一个新函数，接收参数为 ...args
注释 3：新函数接收的参数长度是否大于等于 fn 剩余参数需要接收的长度
注释 4：满足要求，执行 fn 函数，传入新函数的参数
注释 5：不满足要求，递归 currying 函数，新的 fn 为 bind 返回的新函数（bind 绑定了 ...args 参数，未执行），新的 length 为 fn 剩余参数的长度

第 85 题：react-router 里的 <Link> 标签和 <a> 标签有什么区别

如何禁掉 标签默认事件，禁掉之后如何实现跳转。
答：
Link 点击事件 handleClick 部分源码：

```
if (_this.props.onClick) _this.props.onClick(event);
if (
  !event.defaultPrevented && // onClick prevented default
  event.button === 0 && // ignore everything but left
clicks  !_this.props.target && // let browser handle "target=_blank"
etc.  !isModifiedEvent(event) // ignore clicks with modifier keys
) {
  event.preventDefault();
  var history = _this.context.router.history;
  var _this$props = _this.props,
    replace = _this$props.replace,
        to = _this$props.to;
      if (replace) {
        history.replace(to);
      } else {
        history.push(to);
      }
    }
```

Link 做了 3 件事情：

- 有 onclick 那就执行 onclick
- click 的时候阻止 a 标签默认事件（这样子点击123就不会跳转和刷新页面）
- 再取得跳转 href（即是 to），用 history（前端路由两种方式之一，history & hash）跳转，此时只是链接变了，并没有刷新页面

第 86 题：周一算法题之「两数之和」

给定一个整数数组和一个目标值，找出数组中和为目标值的两个数。
你可以假设每个输入只对应一种答案，且同样的元素不能被重复利用。

示例：
给定 nums = [2, 7, 11, 15], target = 9 因为 nums[0] + nums[1] = 2 + 7 = 9 所以返回 [0, 1]
答：

```
function anwser (arr, target) {
  let map = {}  for (let i = 0; i < arr.length; i++) {
   map[arr[i]] = i  }
  for (let i = 0; i < arr.length; i++) {
    var d = target - arr[i]
        if (map[d]) {
          return [i, map[d]]
        }
      }
      return new Error('404 not found')}
```

第 87 题：在输入框中如何判断输入的是一个正确的网址。

```
function isUrl(url) {
  const a = document.createElement("a");
  a.href = url;
  return (
    [
        /^(http|https):$/.test(a.protocol),
        a.host,
        a.pathname !== url,
        a.pathname !== `/${url}`
      ].find(x => !x) === undefined
    );
  }
```

第 88 题：实现 convert 方法，把原始 list 转换成树形结构，要求尽可能降低时间复杂度

以下数据结构中，id 代表部门编号，name 是部门名称，parentId 是父部门编号，为 0 代表一级部门，现在要求实现一个 convert 方法，把原始 list 转换成树形结构，parentId 为多少就挂载在该 id 的属性 children 数组下，结构如下：

```
// 原始 list 如下 let list =[
    {id:1,name:'部门A',parentId:0},
        {id:2,name:'部门B',parentId:0},
        {id:3,name:'部门C',parentId:1},
        {id:4,name:'部门D',parentId:1},
        {id:5,name:'部门E',parentId:2},
        {id:6,name:'部门F',parentId:3},
        {id:7,name:'部门G',parentId:2},
        {id:8,name:'部门H',parentId:4}];
    const result = convert(list, ...);// 转换后的结果如下 let result =
[
    {
      id: 1,
      name: '部门A',
      parentId: 0,
      children: [
        {
          id: 3,
          name: '部门C',
          parentId: 1,
          children: [
            {
              id: 6,
              name: '部门F',
              parentId: 3
            }, {
              id: 16,
              name: '部门L',
              parentId: 3
            }
          ]
        },
        {
          id: 4,
          name: '部门D',
          parentId: 1,
          children: [
           {
             id: 8,
              name: '部门H',
              parentId: 4
            }
          ]
        }
```

```
        ]
      },
    • • •];
```

答：

```
function convert(list) {
   const res = []
    const map = list.reduce((res, v) => (res[v.id] = v, res), {})
        for (const item of list) {
            if (item.parentId === 0) {
                res.push(item)
                Continue
            }
            if (item.parentId in map) {
                const parent = map[item.parentId]
                parent.children = parent.children || []
                parent.children.push(item)
            }
        }
        return res}
```

第 89 题：设计并实现 Promise.race()

```
Promise._race = promises => new Promise((resolve, reject) => {
    promises.forEach(promise => {
            promise.then(resolve, reject)
        })})Promise.myrace = function(iterator) {
        return new Promise ((resolve,reject) => {
            try {
                let it = iterator[Symbol.iterator]();
                while(true) {
                    let res = it.next();
                    console.log(res);
                    if(res.done) break;
                    if(res.value instanceof Promise){
                    res.value.then(resolve,reject);
                } else{
                    resolve(res.value)
                }
                                }
        } catch (error) {
            reject(error)
        }
```

```
})
 }
```

第 90 题：实现模糊搜索结果的关键词高亮显示

```
<!DOCTYPE html><html lang="en"><head>  <meta charset="UTF-8">  <meta
name="viewport" content="width=device-width, initial-scale=1.0">
<meta http-equiv="X-UA-Compatible" content="ie=edge">  <title>auto
complete</title>  <style>
    bdi {
        color: rgb(0, 136, 255);
      }
      li {
        list-style: none;
      }
  </style></head><body>
  <input class="inp" type="text">
  <section>
    <ul class="container"></ul>
  </section></body><script>
  function debounce(fn, timeout = 300) {
    let t;    return (...args) => {
      if (t) {
        clearTimeout(t);
      }
      t = setTimeout(() => {
        fn.apply(fn, args);
      }, timeout);
    }
  }
  function memorize(fn) {
    const cache = new Map();
    return (name) => {
      if (!name) {
        container.innerHTML = '';
        return;
      }
      if (cache.get(name)) {
        container.innerHTML = cache.get(name);
        return;
      }
      const res = fn.call(fn, name).join('');
```

```
        cache.set(name, res);
        container.innerHTML = res;
      }
    }
    function handleInput(value) {
      const reg = new RegExp(`\(${value}\)`);
      const search = data.reduce((res, cur) => {
        if (reg.test(cur)) {
          const match = RegExp.$1;
          res.push(`<li>${cur.replace(match,
'<bdi>$&</bdi>')}</li>`);
        }
        return res;
      }, []);
      return search;
    }
      const data = ["上海野生动物园", "上饶野生动物园", "北京巷子", "
上海中心", "上海黄埔江", "迪士尼上海", "陆家嘴上海中心"]
    const container = document.querySelector('.container');
    const memorizeInput = memorize(handleInput);
    document.querySelector('.inp').addEventListener('input',
debounce(e => {
      memorizeInput(e.target.value);
    }))</script></html>
```

第 91 题：介绍下 HTTPS 中间人攻击

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

中间人攻击过程如下：

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

防范方法：

服务端在发送浏览器的公钥中加入 CA 证书，浏览器可以验证 CA 证书的有效性

第 92 题：已知数据格式，实现一个函数 fn 找出链条中所有的父级 idconst value = ’112’

```
const fn = (value) => {...}fn(value) // 输出 [1,  11,  112]
答：const data = [
      {
        id: "1",
        name: "test1",
        children: [
          {
            id: "11",
            name: "test11",
            children: [
              {
                id: "111",
                name: "test111"
              },
              {
                id: "112",
                name: "test112"
              }
            ]
          },
          {
            id: "12",
            name: "test12",
            children: [
              {
                id: "121",
                name: "test121"
              },
              {
                id: "122",
                name:
 "test122"
              }
            ]
          }
        ]
      }
```

```
];
    const find = value => {
      let result = [];
      let findArr = data;
      let skey = "";
      for (let i = 0, l = value.length; i < l; i++) {
        skey += value[i];
        let item = findArr.find(item => {
          return item.id == skey;
        });
        if (!item) {
          return [];
        }
        result.push(item.id);
        if (item.children) {
          findArr = item.children;
        } else {
         if (i < l - 1) return [];
          return result;
        }
      }
    };
    //调用看结果
    function testFun() {
      console.log("1,11,111:", find("111"));
      console.log("1,11,112:", find("112"));
      console.log("1,12,121:", find("121"));
console.log("1,12,122:", find("122"));
      console.log("[]:", find("113"));
      console.log("[]:", find("1114"));
    }
```

第 93 题：给定两个大小为 m 和 n 的有序数组 nums1 和 nums2。请找出这两个有序数组的中位数。要求算法的时间复杂度为 O(log(m+n))。

示例 1：

nums1 = [1, 3] nums2 = [2]

中位数是 2.0

示例 2：
nums1 = [1, 2] nums2 = [3, 4]
中位数是(2 + 3) / 2 = 2.5

答：

```
const findMedianSortedArrays = function(
    nums1: number[],
      nums2: number[]
    ) {
      const lenN1 = nums1.length;
      const lenN2 = nums2.length;
      const median = Math.ceil((lenN1 + lenN2 + 1) / 2);
      const isOddLen = (lenN1 + lenN2) % 2 === 0;
      const result = new Array<number>(median);
      let i = 0; // pointer for nums1
      let j = 0; // pointer for nums2
      for (let k = 0; k < median; k++) {
        if (i < lenN1 && j < lenN2) {
          // tslint:disable-next-line:prefer-conditional-expression
if (nums1[i] < nums2[j]) {
            result[i + j] = nums1[i++];
          } else {
            result[i + j] = nums2[j++];
          }
        } else if (i < lenN1) {
          result[i + j] = nums1[i++];
        } else if (j < lenN2) {
          result[i + j] = nums2[j++];
        }
      }
      if (isOddLen) {
        return (result[median - 1] + result[median - 2]) / 2;
      } else {
        return result[median - 1];
      }
    };
```

第 94 题：vue 在 v-for 时给每项元素绑定事件需要用事件代理吗？为什么？

在 vue 中 vue 做了处理

如果我们自己在非 vue 中需要对很多元素添加事件的时候，可以通过将事件添加到它们的父节点而将事件委托给父节点来触发处理函数

第 95 题：模拟实现一个深拷贝，并考虑对象相互引用以及 Symbol 拷贝的情况

一个不考虑其他数据类型的公共方法，基本满足大部分场景

```
function deepCopy(target, cache = new Set()) {
  if (typeof target !== 'object' || cache.has(target)) {
    return target
      }
      if (Array.isArray(target)) {
        target.map(t => {
          cache.add(t)
          return t
        })
      } else {
        return
[...Object.keys(target), ...Object.getOwnPropertySymbols(target)].red
uce((res, key) => {
    cache.add(target[key])
          res[key] = deepCopy(target[key], cache)
          return res
        }, target.constructor !== Object ?
Object.create(target.constructor.prototype) : {})
      }
    }
```

主要问题是

- symbol 作为 key，不会被遍历到，所以 stringify 和 parse 是不行的
- 有环引用，stringify 和 parse 也会报错

我们另外用 getOwnPropertySymbols 可以获取 symbol key 可以解决问题 1，用集合记忆曾经遍历过的对象可以解决问题 2。当然，还有很多数据类型要独立去拷贝。比如拷贝一个 RegExp，lodash 是最全的数据类型拷贝了，有空可以研究一下

另外，如果不考虑用 symbol 做 key，还有两种黑科技深拷贝，可以解决环引用的问题，比 stringify 和 parse 优雅强一些

```
function deepCopyByHistory(target) {
```

```
  const prev = history.state
  history.replaceState(target, document.title)
  const res = history.state
  history.replaceState(prev, document.title)
  return res
}
async function deepCopyByMessageChannel(target) {
  return new Promise(resolve => {
    const channel = new MessageChannel()
      channel.port2.onmessage = ev => resolve(ev.data)
channel.port1.postMessage(target)
    }).then(data => data)}
```

无论哪种方法，它们都有一个共性：失去了继承关系，所以剩下的需要我们手动补上去了，故有 Object.create(target.constructor.prototype)的操作

第 96 题：介绍下前端加密的常见场景和方法

首先，加密的目的，简而言之就是将明文转换为密文、甚至转换为其他的东西，用来隐藏明文内容本身，防止其他人直接获取到敏感明文信息、或者提高其他人获取到明文信息的难度。通常我们提到加密会想到密码加密、HTTPS 等关键词，这里从场景和方法分别提一些我的个人见解。

场景-密码传输

前端密码传输过程中如果不加密，在日志中就可以拿到用户的明文密码，对用户安全不太负责。这种加密其实相对比较简单，可以使用 PlanA-前端加密、后端解密后计算密码字符串的 MD5/MD6 存入数据库；也可以 PlanB-直接前端使用一种稳定算法加密成唯一值、后端直接将加密结果进行 MD5/MD6，全程密码明文不出现在程序中。

PlanA 使用 Base64 / Unicode+1 等方式加密成非明文，后端解开之后再存它的 MD5/MD6 。

PlanB 直接使用 MD5/MD6 之类的方式取 Hash ，让后端存 Hash 的 Hash 。

场景-数据包加密

应该大家有遇到过：打开一个正经网站，网站底下蹦出个不正经广告——比如 X 通的流量浮层，X 信的插入式广告……（我没有针对谁）但是这几年，我们会发现这种广告逐渐变少了，其原因就是大家都开始采用 HTTPS 了。被人插入这种广告的方法其实很好理解：你的网页数据包被抓取->在数据包到达你手机之前被篡改->你得到了带网页广告的数据包->渲染到你手机屏幕。而 HTTPS 进行了包加密，就解决了这个问题。严格来说我认为从手段上来看，它不算是一种前端加密场景；但是从解决问题的角度来看，这确实是前端需要知道的事情。

Plan 全面采用 HTTPS

场景-展示成果加密

经常有人开发网页爬虫爬取大家辛辛苦苦一点一点发布的数据成果，有些会影响你的竞争力，有些会降低你的知名度，甚至有些出于恶意爬取你的公开数据后进行全量公开……比如有些食谱网站被爬掉所有食谱，站点被克隆；有些求职网站被爬掉所有职位，被拿去卖信息；甚至有些小说漫画网站赖以生存的内容也很容易被爬取。
Plan 将文本内容进行展示层加密，利用字体的引用特点，把拿给爬虫的数据变成“乱码”。举个栗子：正常来讲，当我们拥有一串数字“12345”并将其放在网站页面上的时候，其实网站页面上显示的并不是简单的数字，而是数字对应的字体的“12345”。这时我们打乱一下字体中图形和字码的对应关系，比如我们搞成这样：

图形：1 2 3 4 5 字码：2 3 1 5 4
这时，如果你想让用户看到“12345”，你在页面中渲染的数字就应该是“23154”。这种手段也可以算作一种加密。

第 97 题：React 和 Vue 的 diff 时间复杂度从 O(n^3) 优化到 O(n) ，那么 O(n^3) 和 O(n) 是如何计算出来的？

三种优化来降低复杂度：

1. 如果父节点不同，放弃对子节点的比较，直接删除旧节点然后添加新的节点重新渲染；
2. 如果子节点有变化，Virtual DOM 不会计算变化的是什么，而是重新渲染，
3. 通过唯一的 key 策略

第 98 题：写出如下代码的打印结果

```
function changeObjProperty(o) {
  o.siteUrl = "http://www.baidu.com"
  o = new Object()
  o.siteUrl = "http://www.google.com"
}
 let webSite = new Object();
changeObjProperty(webSite);
console.log(webSite.siteUrl);
```

答：
webSite 属于复合数据类型，函数参数中以地址传递，修改值会影响到原始值，但如果将其完全替换成另一个值，则原来的值不会受到影响

第 99 题：编程算法题

用 JavaScript 写一个函数，输入 int 型，返回整数逆序后的字符串。如：输入整型 1234，返回字符串“4321”。要求必须使用递归函数调用，不能用全局变量，输入函数必须只有一个参数传入，必须返回字符串。

```
function fun(num) {
  let num1 = num / 10;
  let num2 = num % 10;
  if (num1 < 1) {
    return num;
      } else {
        num1 = Math.floor(num1);
        return `${num2}${fun(num1)}`;
      }
    }
    var a = fun(12345);
    console.log(a);
    console.log(typeof a);
```

第 100 题：请写出如下代码的打印结果

```
function Foo() {
  Foo.a = function() {
    console.log(1);
    };
      this.a = function() {
        console.log(2);
      };
    }
    Foo.prototype.a = function() {
      console.log(3);
    };
    Foo.a = function() {
      console.log(4);
```

```
    };
    Foo.a();
    let obj = new Foo();
    obj.a();
    Foo.a();
```

答：

4 2 1