js 中 this 指向问题:

```
1var a = 1;
1var b = { c: function ()
1{ console.log(this.a); },
1d: () => { console.log(this.a); } };
1b.d.bind({ a: 2 });
1var fun = b.c;
1fun();
1b.c();
1b.d();
```

第一个 func 应该是隐式类型绑定，this 指向 window，
window.func() =>1
第三个箭头函数本身不含有 this，绑定的是定义时候的上下文，=> 1;
第二个显式绑定，this 指向 b，b 没有 a 属性打印 undefined

2. 事件轮询机制：

```
1console.log(1)
1setTimeout(() => {
1console.log(2)
1Promise.resolve().then(() => <font color="#999999">{</font>
1console.log(3) }) })
1new Promise((resolve) => {
1console.log(4)
1setTimeout(() => {
1console.log(5) resolve(); }, 2);
1}).then(res => {
1console.log(res) });
// 1 4 2 3 5 undefined
```

3. 手写 promsie.all

```
1function all(promises){
1const values = [];
1return new Promise((resolve, reject)=>
1{ promises.forEach(
1(promise, index)=>
1{ promise.then((value)=>{
1//values.push(value);
1values[index] = value;
1if(values.length == promises.length){
1resolve(values); } },

1reason => { reject(reason); })
1})
1})
```

13.// 环形链表判断：
说了快慢指针

// 对象是否有循环引用
// 没搞清楚这个题目：
1//line=readline()
1//print(line)
1var a = { b: { c: { d: a } } }
1function refCycle(obj){
1for(let key in obj){
1if(obj[key] == obj){
1return true; } else{
1refCycle(obj[key]); }
1} }
1console.log(refCycle(a))
1function fn(object) {

1// 首先判断 object 是否存在于 map.keys 中
1if (Array.from(map.keys()).includes(object))
1{ // 如果存在则取出值并返回 return map.get(object); }
1var cloneObj = {};
1// 设置 object 为 key，cloneObj 为值
1map.set(object, cloneObj);
1for (var key in object) {
1// 赋予新对象相应的 property
1// 通过递归调用来拷贝 property 的值
1cloneObj[key] = fn(object[key]);
1}
1// 返回新对象 return cloneObj;

1}
1var obj = {};
1obj.a = obj;
1var map = new Map();
1fn(obj);
手写一个 flat 函数
刚开始写了一个版本：
1Array.prototype.flat1 =
1function (arr, n){
1let newArr = new Array();
1for(let i = 0; i < arr.length; i++)(
1if(typeof(arr[i]) !== "object" && n-- > 0){

1newArr = newArr.concat(this.flat1(arr[i]));

```
1} else {
1newArr.push(arr[i]); } )
1return newArr;
1}
1var flat = function(arr, depth){
1let res = [], depthArg = depth || 1,
1depthNum = 1,
1flatMap = (arr) => {
1arr.map((element, index, array) =>
1{ if(Object.prototype.toString(element).slice(8,-1) === "Array")
1{ if(depthNum < depthArg) {
1depthNum++;
1flatMap(element); } else
1{ res.push(element);
1if(index === array.length - 1)
1depthNum = 0; } } }) };
1flatMap(arr); return res; };
1let arr = [[1], [[2]], [3]];
1console.log(flat(arr));
1// [1,[2], [3]]
```

15.hash 模式和 history 模式的实现原理

监听 hash 的改变:

```
1<!DOCTYPE html> <html lang="en">
1<head> <meta charset="UTF-8">
1<meta name="viewport" content="width=device-width, initial-scale=1.0"> <title>Document</title>
1</head> <body> <div id="app">
1<a href="#/home">首页</a>
1<a href="#/about">关于</a>
1</div> <div class="router-view">
1</div> <script>
1// 获取 router-view 的 dom const routerViewEl =
1document.getElementsByClassName("router-view")[0];
1// 监听 url 的改变 window.addEventListener("hashchange", () =>
1{ switch (location.hash) {
1case "#/home":
1routerViewEl.innerHTML = "首页";
1break;
1case "#/about": routerViewEl.innerHTML = "关于";
1break;
 default: routerViewEl.innerHTML = ""; } }); // html5 中的 history // history 接口是 HTML5 新增加的，
1它有六种模式改变 url 而不刷新页面 // replaceState: 替换原来的路径 // pushState: 使用新的路径
 // popState: 路径的回退 // go: 向前或向后 forward: 向 1. 获取 router-view 的 dom
1const routerViewEl =
1document.getElementsByClassName("router-view")[0];
```

12.histort 新增加 api

```
// 获取所有的 a 元素，自己来监听 a 元素的改变
const aEls = document.getElementsByTagName("a");
for (let el of aEls)
{ el.addEventListener("click",
e => { e.preventDefault();
const href = el.getAttribute("href");
history.pushState({}, "", href);
urlChange(); // history.go(-1)
// history.back();
// urlChange(); }) }
// 执行返回操作时候，依然来到 urlChange

window.addEventListener('popstate', urlChange);


// window.addEventListener("pushState", urlChange);


// 监听 URL 的改变 function urlChange() {
console.log(location.pathname);
switch (location.pathname) {
case "/home": routerViewEl.innerHTML = "首页";
break;
case "/about":
routerViewEl.innerHTML = "关于";
break;
default: routerViewEl.innerHTML = "";
} };
```

实现数组的 slice 方法：

```
Array.prototype.slice =
function(start, end){
let len = this.length;
let l = start === undefined ? 0 :
start < 0 ? Math.max(start + len, 0) :
Math.min(start, len);
let r = end === undefined ? len : end < 0 ?
Math.max(end + len, 0) :
Math.min(end, len);
const res = []; while(l < r)
{ res.push(this[l++]);
}
return res; }
```

判断数据类型 3 种方法，实现 instaneOf

```
const instance_of = (left, rigth) =>
{ const baseType = ['number', 'string', 'boolean', 'undefined', 'symbol'];
const RP = right.prototype;
while(true){ if(left == null) {
return false; } else if
(left == RP) { return true; }
left = left.__proto__;
} }
```