

ES6

// 面试问题 // ☆注意点 // eg例子 //

一、简介

- JavaScript是一种基于对象（object--based）的语言。
- JS是一个弱类型的语言
 - html 布局、内容
 - css 样式
 - JS 交互
- JS 分为两种数据类型：基本数据类型（number, string, boolean, undefined, symbol
（受保护的数据类型）），引用数据类型（数组，对象）。
- F12 控制台
 - Elements
 - 左 html代码，右 css代码
 - Console 控制台
 - 检查JS代码
 - Sources
 - 打断点
 - 所有源代码
 - Network
 - 整个网页加载的东西
- JS是一种（运行在客户端的交汇式）脚本语言，ES是一种标准。JS语言遵循了ES的标准。
- 新语法通过Babel转化为ES5的语法再在浏览器上运行



- 两个环境：开发环境与生产环境

二、语法

1. let (定义变量)

- 定义变量时，写var与不写var的区别？

不写var（未声明变量），即未出现新变量（将覆盖该元素的值），相当于给全局对象定义一个属性（相当于定义一个全局的window的属性），全局对象过多即window属性过多会污染全局对象；

写var，在当前作用域内写一个新的变量。

```
1 //写var
2 var a = 5 ;
3 function foo(){
4     var a = 6;
5     console.log('1==' + a);
6 }
7 foo(); //调用函数
8 console.log('2==' + a);
9
10 //输出结果：
11 //1==6 2==5
12
13 //不写var
14 var a = 5 ;
15 function foo(){
16     a = 6;
17     console.log('1==' + a);
18 }
19 foo(); //调用函数
20 console.log('2==' + a);
21
22 //输出结果：
23 //1==6 2==6
```

1.1 let与var的区别 (var的特点)

- let不属于顶层对象window (var属于)
 - 用let优点：不污染顶层对象window
- let不允许重复声明 (var允许)
 - var的重复声明及覆盖，let在重复声明时会报错
 - 用let优点：防止使用到同一个名字造成数据更改

```
1 var a = 5 ;
2 var a = 6 ;
3 console.log(a);
4
5 //输出结果：6(覆盖)
```

- let不存在变量提升 (var存在)
 - 用let优点：确保先定义再使用

```
1 console.log(a);
```

```
2 var a = 5 ;
3 //相当于
4 //var a ;
5 //console.log(a);
6 //a = 5 ;
7 //故输出结果为：undefined (未定义)
8
9 //若用let替换var，则会报错
```

- let存在暂时性死区
 - 暂时性死区：在声明变量前的区域（在声明变量之前是不能够使用该变量的）
 - 先定义再调用
- let存在块级作用域（var不存在）
 - 即在作用域内才生效

```
1 if(false){
2     var a = 5;
3 }
4 console.log(a);
5 //输出：undefined
6 //将let替换掉var，则报错
```

2.const (定义常量)

2.1 ES5定义常量的方法

```
1 object.defineProperty(window, PI, {
2     value:3.14159,
3     writable:false, //定义为不可写即为常量
4 });
5 console.log(PI);
```

2.2 ES6定义常量的方法 const

```
1 const a = 5;
2 a = 6; //无效
3 console.log(a);
4 //输出结果：a=5
```

2.3 关于const 1 (同let)

- const不属于顶层对象window
- const不允许重复声明
- const不存在变量提升
- const存在暂时性死区
- const存在块级作用域

2.4 关于const 2

eg1:

```
1 const a = 5;
2 a = 6;
3 console.log(a);
4 //输出结果：5
```

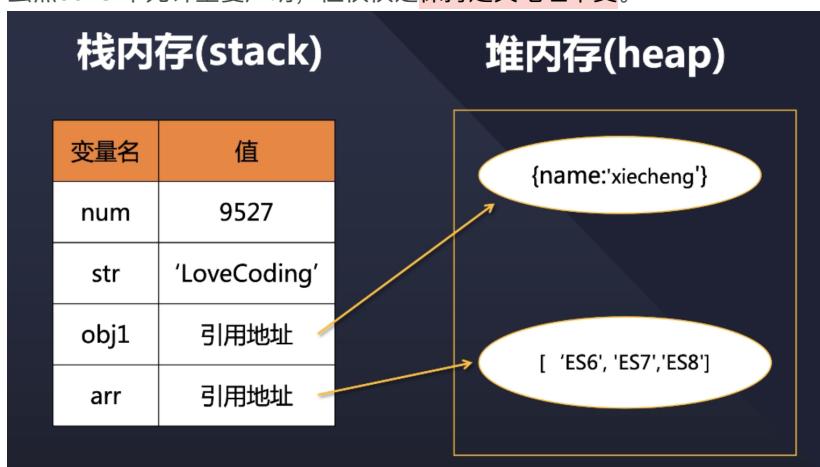
eg2:

```
1 const ly = {
2   name:'liuyu',
3   age:21,
4 };
5 console.log(ly);
6 //输出结果：
7 //{"name:'liuyu',age:21}
8
9 const ly = {
10   name:'liuyu',
11   age:21,
12 };
13 ly.age = 3;
14 console.log(ly);
15 //输出结果：
16 //{"name:'liuyu',age:3}
```

eg3:

```
1 const arr = [1, 2, 3];
2 arr[0] = 824;
3 console.log(arr);
4 //输出结果：
5 // [824, 2, 3]
```

虽然const不允许重复声明，但仅仅是保持定义地址不变。



其中，基本变量存入栈内存（stack），引用变量存入堆内存（heap）（类似于obj、arr等，他们存放在栈内存当中的只是一个引用地址，而真正的数据存放在堆内存当中，即引用地址不变，数据可改变。）

eg4:

```
1 //Object.freeze 可以浅层冻结
2 const ly = {
3   name:'liuyu',
4   age:21,
5 };
6 Object.freeze(ly); //冻结
7 ly.age = 3; //无效
8 console.log(ly);
9 //输出结果：
10 //{"name": "liuyu", "age": 21}
```

☆ let和const，优先使用const。

3.解构赋值

- 按照一定模式，从数组和对象中提取值，对变量进行赋值。
- ☆ 等号左右两边要相等！

3.1 数组解构

eg1:

```
1 const arr = [1, 2, 3];
2 const [a, b, c] = arr; //等号左右两边相同
3 console.log(a, b, c);
4 //输出结果：1 2 3
```

eg2:

```
1 const [a, b, c] = [1, 2, 3, 4];
2 console.log(a, b, c);
3 //输出结果：1 2 3(不报错)
```

eg3:

```
1 const [a, b, c, d] = [1, 2, 3];
2 console.log(a, b, c, d);
3 //输出结果：1 2 3 undefined
```

3.2 对象解构

eg1:

```
1 const obj{
2   name:'liuyu',
3   age:21,
4 };
5 const {name, age} = obj;
6 //即等于
```

```
7 //const name = obj.name;
8 //const age = obj.age;
9 console.log(name,age);
10 //输出结果：liuyu 21
```

☆ tips: const { name,age } 和 const { age, name }无差别, name 和 age 是一组键值对。

eg2:

```
1 const obj{
2   name:'liuyu',
3   age:21,
4 };
5 const name = 'yuxiaobo';
6 const {name,age} = obj;
7 console.log(name,age);
8 //输出结果：报错，name已被声明
9
10 const obj{
11   name:'liuyu',
12   age:21,
13 };
14 const name = 'yuxiaobo';
15 const {
16   name:userName,//起别名
17   age,
18 } = obj;
19 console.log(name,age);
20 //输出结果：yuxiaobo 21
21 //若console.log(userName,age);则输出：liuyu 21
```

特别的：

```
1 const obj{
2   name:'liuyu',
3   age:21,
4 };
5 const {
6   name:userName,//obj.name此刻已经等于userName
7   age:userAge,
8 } = obj;
9 console.log(name);
10 //结果不报错
11
12 const obj{
13   name:'liuyu',
14   age:21,
15 };
16 const {
17   name:userName,
18   age:userAge,
19 } = obj;
20 console.log(age);
21 //结果报错
```

☆ window下有 name 属性，因此 console.log(name) 不报错。 (尽量不以 name 起名)

3.3 字符串解构 (同数组解构)

eg1: (遍历)

```
1 const str = 'liuyu';
2 for(let i = 0; i < str.length; i++){
3     console.log(str[i]);
4 }
5 //输出结果：l i u y u
```

eg2:

```
1 const str = 'liuyu';
2 const [a, b, c, d, e] = str;
3 console.log(a,b,c,d,e);
4 //输出结果：l i u y u
```

3.4 应用

3.4.1 函数参数的解构赋值

eg1:

```
1 function sum(x, y){
2     return x + y;
3 }
4 const res = sum(5, 6);
5 console.log(res);
6 //输出结果：11
```

eg2: 对象

```
1 function foo(o){ // o 形参
2     console.log(o.name,o.age);
3 }
4 const obj = {
5     name:'liuyu',
6     age:21,
7 }
8 foo(obj); // obj 实参
9 //输出结果：liuyu 21
10
11 //解构赋值：
12 function foo({name,age}){
13     console.log(name,age);
14 }
15 foo({name:'liuyu', age:21});
16 //输出结果：liuyu 21
```

eg3: 数组

```
1 function bar(arr){
```

```
2   console.log(arr[0], arr[1], arr[2]);
3 }
4 bar([1, 2, 3]);
5 //输出结果：1 2 3
6
7 //解构赋值
8 function bar([a, b, c]){
9   console.log(a, b, c);
10 }
11 bar([1, 2, 3]);
```

eg4: 函数有返回值

```
1 function foo(){
2   //业务逻辑操作
3   return{
4     name:'liuyu',
5     age:21,
6   }
7 }
8 const res = foo();
9 console.log(res.name, res.age);
10 //输出结果:liuyu 21
11
12 //解构赋值
13 function foo(){
14   //业务逻辑操作
15   return{
16     name:'liuyu',
17     age:21,
18   }
19 }
20 const {name,age} = foo(); //等号两边要相同
21 console.log(name,age)
22 //输出结果:liuyu 21
```

eg5: 默认值①

```
1 const arr = ['liuyu', 'ly', 'yuxiaobo'];
2 const [a, b, c, d = 'tianC'] = arr;
3 console.log(a, b, c, d);
4 //输出结果：liuyu ly yuxiaobo tianC
5
6 const arr = ['liuyu', 'ly', 'yuxiaobo', 'yuzetian'];
7 const [a, b, c, d = 'tianC'] = arr;
8 console.log(a, b, c, d);
9 //输出结果：liuyu ly yuxiaobo yuzetian
```

可设置默认值，但为浅层默认（惰性的）。

eg6: 默认值2

```
1 function foo(){
2   console.log(123);
```

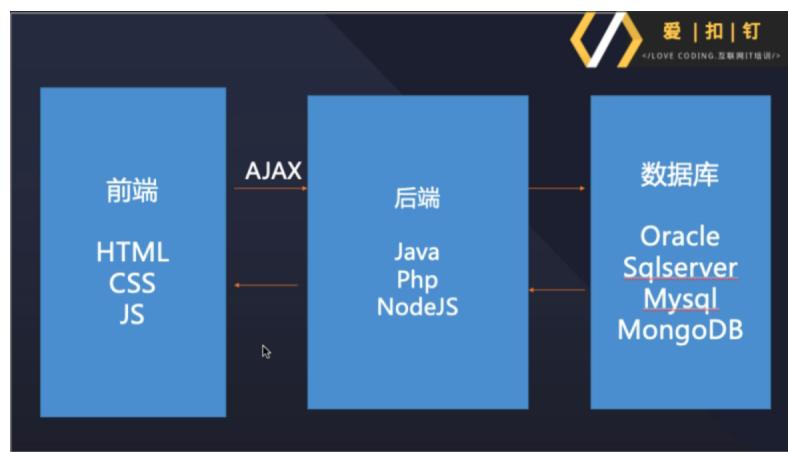
```
3 }
4 const [a = foo()] = [];
5 //输出结果：123
6 //“a = foo()”为默认值
```

3.4.2 变量互换

eg:

```
1 //①
2 let a = 5;
3 let b = 6;
4 let temp = a;//temp = 5
5 a = b;//a = 6
6 b = temp;//b = 5
7 console.log(a, b);
8
9 //②
10 let a = 5;
11 let b = 6;
12 a = a + b;//a = 11
13 b = a - b;//b = 5
14 a = a - b;//a = 6
15 console.log(a, b);
16
17 //③
18 let a = 5;
19 let b = 6;
20 [a, b] = [b, a]
21 console.log(a, b);
```

3.4.3 JSON



*AJAX 不刷新页面请求

*CDN (Content Delivery Network) 内容分发网络

3.5 关于解构赋值是深拷贝还是浅拷贝

解构赋值，如果所解构的原对象是一维数组或对象，其本质就是对基本数据类型进行等号赋值，那它就是深拷贝；

如果是多维数组或对象，其本质就是对引用类型数据进项等号赋值，那它就是浅拷贝；

4.ES5中的数组遍历

4.1 for

```
1 const arr = [1, 2, 3, 4, 5];
2 for(let i = 0; i < arr.length; i++){
3     if(i === 2){
4         continue;
5     }
6     console.log(arr[i]);
7 }
8 //输出结果：1 2 4 5
9 //break替换continue，输出结果：1 2
```

☆ continue 结束本次循环，开始下次循环

☆ break 跳出循环

4.2 forEach()

- 没有返回值，只针对每个元素调用function，不能使用continue和break

```
1 const arr = [1, 2, 3, 4, 5];
2 const res = arr.forEach(function(elem){
3     console.log(elem);
4     return elem;
5 })
6 console.log(res);
7 //输出结果：1 2 3 4 5 undefined ( 没有返回值 )
```

4.3 map()

- 有返回值，且返回值长度与原数组长度一致

```
1 const arr = [1, 2, 3, 4, 5];
2 const res = arr.map(function(elem, index, array){
3     if( elem > 2){
4         return elem;
5     }
6 })
7 console.log(res);
8 //输出结果： (5)[undefined,undefined, 3, 4, 5 ]
```

4.4 filter()

- filter:过滤；只返回符合function条件的元素数组

```
1 const arr = [1, 2, 3, 4, 5];
```

```
2 const res = arr.filter(function(elem, index, array){  
3     if( elem > 2){  
4         return elem;  
5     }  
6 })  
7 console.log(res);  
8 //输出结果：(3)[3, 4, 5]
```

4.5 some() / every()

- some () : 返回boolean, 判断是否有元素符合function条件
- every () : 返回boolean, 判断每个元素是否符合function条件

```
1 //some()  
2 const arr = [1, 2, 3, 4, 5];  
3 const res = arr.some(function(elem, index, array){  
4     if( elem > 2){  
5         return elem;  
6     }  
7 })  
8 console.log(res);  
9 //输出结果：true  
10  
11 //every()  
12 const arr = [1, 2, 3, 4, 5];  
13 const res = arr.filter(function(elem, index, array){  
14     if( elem > 2){  
15         return elem;  
16     }  
17 })  
18 console.log(res);  
19 //输出结果：false
```

4.6 reduce()

- 接收一个函数作为累加器, (prev, cur, index, array)

```
1 const arr = [1, 2, 3, 4, 5];  
2 const res = arr.reduce(function(prev, cur, index, array){  
3     console.log(prev, cur);  
4     return prev += cur;  
5 },0);  
6 console.log(res);
```

输出结果:

0 1	5 数组遍历.html:75
1 2	5 数组遍历.html:75
3 3	5 数组遍历.html:75
6 4	5 数组遍历.html:75
10 5	5 数组遍历.html:75
15	5 数组遍历.html:78
>	

- prev 累加器初始值（第一次0 (no次数 yes结果)）
- cur 当前循环时的第一个值

4.7 for in ?

- `for in`用于遍历对象，不能用于遍历数组。因为`for in`会循环遍历出自定义方法。

eg: 遍历对象

```

1 const obj = {
2   name:'liuyu',
3   age:21,
4 };
5 for (let prop in obj){
6   console.log(obj[prop])
7 }
8 //输出结果：liuyu 21
9 //console.log(obj.prop)输出结果为：undefined undefined

```

4.8 其他问题

- 如何判断一个值是否为数组？
 - `console.log(typeof arr)? (X)` `typeof`只能判断基本类型，不能判断引用类型。
 - 使用 `console.log(arr instanceof Array)` (arr为数组名称)

5.ES6中的数组遍历

5.1 find()

- `find()`: 返回第一个通过测试的元素

```

1 const arr = [1, 2, 3, 4, 5, 3];
2 const res = arr.find(elem, index, array){
3   return res === 3;
4 }
5 console.log(res);
6 //输出结果：3

```

5.2 findIndex()

- `findIndex()`: 返回的值为该通过第一个元素的索引

```

1 find():返回第一个通过测试的元素
2 const arr = [1, 2, 3, 4, 5, 3];

```

```
3 const res = arr.findIndex(elem, index, array){  
4     return res === 3;  
5 }  
6 console.log(res);  
7 //输出结果：2
```

5.3 for of

```
1 //arr : 输出值  
2 const arr = ['es6', 1, 2, 3];  
3 for(let elem of arr){  
4     console.log(elem)  
5 };  
6 //输出结果：es6 1 2 3  
7  
8 //values : 输出值  
9 const arr = ['es6', 1, 2, 3];  
10 for(let elem of arr.values()){  
11     console.log(elem)  
12 };  
13 //输出结果：es6 1 2 3  
14  
15 //keys:输出下标  
16 const arr = ['es6', 1, 2, 3];  
17 for(let index of arr.keys()){  
18     console.log(index)  
19 };  
20 //输出结果：0 1 2 3  
21  
22 //entries():[index, elem]  
23 const arr = ['es6', 1, 2, 3];  
24 for(let [index, elem] of arr.entries()){  
25     console.log(elem)  
26 };  
27 //输出结果： 0 1 2 3 (index)  
28 //输出结果：es6 1 2 3 (elem )
```

6.数组的扩展

6.1 类数组 / 伪数组

- 拥有length属性
- 不具有数组所具有的方法

eg: (getElementsByTagName.....)HTMLCollection, (querySelectorAll) NodeList.....

6.2 Array.from(): 类数组转化为数组

```
1 <body>
```

```
2 <div id="parent">
3     <div class="elem">1</div>
4     <div class="elem">2</div>
5     <div class="elem">3</div>
6 </div>
7 <script>
8     const elem = document.querySelectorAll('#parent .elem');
9     const arr = Array.from(elem);
10    arr.push(123);
11    console.log(arr);
12 </script>
13 </body>
```

输出结果：

```
Array(4)
  0: div.elem
  1: div.elem
  2: div.elem
  3: 123
  length: 4
  [[Prototype]]: Array(0)
```

6.3 Array.of(): 将一组值转化为数组

```
1 const arr = Array.of(3, 'es6', true, [1,2,3], {age: 11});
2 console.log(arr);
```

输出结果：

```
Array(5)
  0: 3
  1: "es6"
  2: true
  3: (3) [1, 2, 3]
  4: {age: 11}
  length: 5
  [[Prototype]]: Array(0)
```

6.4 fill(values, start, end): 使用给定值填充数组，改变原数组

```
1 const arr = Array.of(3, 'es6', true, [1,2,3], {age: 11});
2 arr.fill('xxx', 2, 4)
3 console.log(arr);
```

输出结果：



```
▼ (5) [3, "es6", "xxx", "xxx", {...}] ⓘ
  0: 3
  1: "es6"
  2: "xxx"
  3: "xxx"
  ▶ 4: {age: 11}
  length: 5
  ► [[Prototype]]: Array(0)
```

6.5 includes(): 数组是否包含给定的值, boolean

- 检测是否包含某串字符

```
1 const str = 'LoveCoding';
2 console.log(str.indexOf('Cod'));
3 //输出结果：4 ( 找得到 : 输出该字符串第一个字母的下标 ; 找不到 : 输出-1 )
4
5 const str = 'LoveCoding';
6 console.log(str.includes('Cod'));
7 //输出结果 : true ( 找得到 : 输出true ; 找不到 : 输出false )
```

7. 扩展运算符和剩余 (rest) 运算符

- 表达方式: ...
- 扩展运算符: 把数组或者类数组展开成用逗号隔开的值
- rest参数: 把逗号隔开的值组合成一个数组 (剩余运算符必须是最后一个参数)

7.1 实例

eg1: 数组 -> 值

```
1 const arr = [1, 2, 3];
2 console.log(...arr);
3 //输出结果 : 1 2 3
```

eg2: 不确定参数

```
1 function foo(a, b, ...arg){
2   console.log(a);
3   console.log(b);
4   console.log(arg);
5 }
6 foo(1, 2, 3, 4);
7 //输出结果 : 1 2 [3,4]
```

eg3: 数组复制

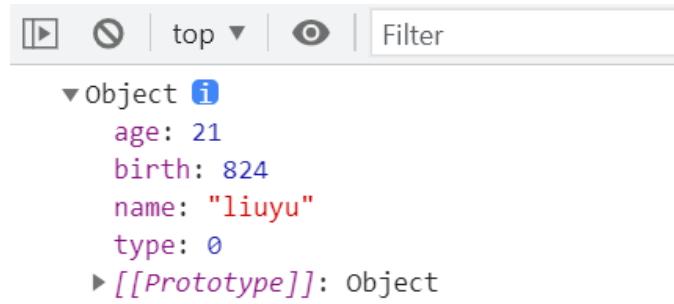
```
1 const arr1 = [1, 2, 3];
```

```
2 const arr2 = [...arr1];
```

eg4: 合并对象

```
1 const info = {  
2     name:'liuyu',  
3     age:21,  
4     birth:0824  
5 }  
6 const obj = {  
7     ...info,  
8     type:0  
9 }  
10 console.log(obj);
```

输出结果:



```
Object {  
    age: 21  
    birth: 824  
    name: "liuyu"  
    type: 0  
} [prototype]: Object
```

8.箭头函数

- this指向定义时所在对象，而非调用时所在对象。 (注意this指向)
- 箭头函数不可以当作构造函数
- 不可以使用arguments对象

```
1 const obj = function(a){  
2     console.log(a);  
3 }  
4 obj(1);  
5 //箭头函数  
6 const obj = a => console.log(a);  
7 obj.(1);
```

箭头函数be like: 去掉function, 在 () 和 {} 之间添加箭头=>, 若参数只有一个, () 可去掉, 若 {} 内只有一行可去掉 {} (只有一行相当于return了)。

9.对象扩展

9.1 属性简明表示法

- 如果方法中, key与value相等, 可以省略value。 (方法可省略“: function”)

```
1 const name = 'liuyu';
```

```

2 const age = 21;
3 const obj = {
4   name:name,
5   age:age,
6   showName:function(){
7     console.log(this.name);
8   }
9 };
10 console.log(obj);
11 obj.showName();
12
13 //可简明为：
14 const name = 'liuyu';
15 const age = 21;
16 const obj = {
17   name,
18   age,
19   showName(){
20     console.log(this.name);
21   }
22 };
23 console.log(obj);
24 obj.showName();

```

9.2 属性名表达式

- 当key值为变量时，可为其添加“[]”

```

1 const n = 'name';
2 const obj = {
3   n:'liuyu';
4 };
5 console.log(obj);
6 //输出结果：{n："liuyu"}
7
8 const n = 'name';
9 const obj = {
10   [n]:'liuyu';
11 };
12 console.log(obj);
13 //输出结果：{name:"liuyu"}

```

9.3 Object.is()

- 判断两个值是否严格相等 (====)

```

1 const obj1 = {
2   name:'liuyu',
3   age:21
4 };
5 const obj2 = {

```

```
6   name:'liuyu',
7   age:21
8 };
9 console.log(Object.is(obj1, obj2));
10 //输出结果：false
11 //obj1和obj2的引用地址不一样
```

特别的：

```
1 //NaN:Not a Number
2 console.log(NaN == NaN);
3 //输出结果：false
4 console.log(Object.is(NaN, NaN));
5 //输出结果：true
```

9.4 Object.assign ()

- 对象合并（数组be like: ...）,第一个参数是目标对象，后面的参数都是源对象。☆注意：
Object.assign() 为浅拷贝。

合并方法一：...

```
1 const obj1 = {
2   name:'liuyu',
3   age:21
4 };
5 const obj2 = {
6   birth:824
7 };
8 const obj3 = {
9   ...obj1,
10  ...obj2
11 };
12 console.log(obj3);
```

合并方法二：Object.assign()

```
1 const obj1 = {
2   name:'liuyu',
3   age:21
4 };
5 const obj2 = {
6   birth:824
7 };
8 Object.assign(obj1, obj2); //把obj2的东西合并到obj1
9 console.log(obj1);
```

9.5 in：判断对象是否包含某个属性

- 用于数组和对象

```
1 const obj = {
2   name:'liuyu',
3   age:21
```

```
4 };
5 console.log('age1' in obj);
6 //输出结果：false
```

9.6 对象的遍历方式

方法一：for ... in

```
1 const obj = {
2   name:'liuyu',
3   age:21
4 };
5 for(let prop in obj){
6   console.log(prop, obj[prop]);
7 }
```

输出结果：

The screenshot shows a browser's developer tools console interface. At the top, there are several icons: a play button, a stop button, a 'top' dropdown, a refresh eye icon, and a 'Filter' input field. Below the toolbar, the console output is displayed in two lines:
name liuyu
age 21

方法二：Object.keys()

```
1 const obj = {
2   name:'liuyu',
3   age:21
4 };
5 Object.keys(obj).forEach(function(prop){
6   console.log(prop, obj[prop]);
7 });
8 //可简化为：
9 const obj = {
10   name:'liuyu',
11   age:21
12 };
13 Object.keys(obj).forEach(prop => console.log(prop, obj[prop]));
14 //输出结果同上
```

10.类与继承

- 类 (class) 是对象 (object) 的模板，定义了同一种对象共有的属性和方法。

10.1 面向过程与面向对象



1. 打开冰箱门
2. 把大象装进去 面向过程
3. 关上冰箱门

1. 大象
2. 冰箱 面向对象
3. 隐藏对象

10.2 改变this指向

- 改变方法里this的指向: call, apply, bind

eg1:

```

1 const obj = {name:'es6'};
2 function foo(a, b){
3     console.log(this, a, b);
4 }
5 foo.call(obj, 1, 2);
6 //foo.apply(obj, [1, 2]);
7 //foo.bind(obj, 1, 2)();或者foo.bind(obj)(1, 2);

```

- call, apply, bind区别?
 - call: (this指向, 1, 2,(多个参数)) 。改变指向+调用方法。
 - apply: (this指向, [数组]) (两个参数)。改变指向+调用方法。
 - bind: 只能改变指向不会调用方法。

eg2:

```

1 <body>
2     <button id = "btn">按钮</button>
3     <script>
4         const oBtn = document.querySelector('#btn');
5         oBtn.addEventListener('click',function(){
6             setTimeout(function(){
7                 this.innerHTML = "我被点击啦";
8             }.bind(this), 2000)
9         })
10    </script>
11 </body>

```

只能用bind, call和apply会立即被调用。

10.3 ES5的类与继承

10.3.1 类

```

1 function Animal(age, color){
2     this.age = age,
3     this.color = color
4 };
5 //实例方法 ( 实例化.方法 ) 跟实例化对象有关
6 Animal.prototype.showAge = function(){
7     console.log('我的年龄是：' + this.age);
8 };
9 //静态方法 跟对象无关
10 Animal.run = function(){
11     console.log("我是动物，我会跑")
12 };
13
14 const dog = new Animal(2, 'black');
15 console.log(dog);
16 dog.showAge();
17 Animal.run();

```

输出结果：

The screenshot shows a browser's developer tools console interface. At the top, there are buttons for play/pause, stop, and filter, followed by a dropdown menu set to 'top'. Below the toolbar, the console output is displayed:

```

▼ Animal ⓘ
  age: 2
  color: "black"
  ► [[Prototype]]: object
我的年龄是: 2
我是动物，我会跑
Live reload enabled.
>

```

10.3.2 继承

```

1 function Animal(age, color){
2     this.age = age,
3     this.color = color
4 };
5 //实例方法 ( 实例化.方法 ) 跟实例化对象有关
6 Animal.prototype.showAge = function(){
7     console.log('我的年龄是：' + this.age);
8 };
9 //静态方法 跟对象无关
10 Animal.run = function(){
11     console.log("我是动物，我会跑")
12 };
13
14 function Cat(name, age, color){
15     //继承父类的属性
16     Animal.call(this, age, color);
17     this.name = name;

```

```
18 } ;
19 //继承父类的方法
20 Cat.prototype = new Animal();
21 Cat.prototype.constructor = Cat;
22
23 const c1 = new Cat('小黑', 2, 'white' );
24 console.log(c1);
25 c1.showAge();
```

输出结果：

The screenshot shows a browser's developer tools console interface. At the top, there are buttons for play/pause, stop, and filter, followed by a dropdown set to 'top'. A 'Filter' input field is present. Below the toolbar, the console output is displayed in a list format. The first item is a expanded object: '▶ Cat {age: 2, color: "white", name: "小花"}'. The second item is a string: '我的年龄是: 2'. There is also a blue '▶' button at the bottom left of the list.

```
>
```

10.4 ES6的类与继承

10.4.1 类

```
1 class Person{
2     constructor(name, age){
3         this.name = name;
4         this.age = age;
5     }
6     showInfo(){
7         console.log(this.name, this.age);
8     }
9 }
10 let student = new Person("liuyu", 21);
11 student.showInfo();
12 //输出结果：liuyu 21
```

10.4.2 继承

```
1 class Person{
2     constructor(name, age){
3         this.name = name;
4         this.age = age;
5     }
6     showInfo(){
7         console.log(this.name, this.age);
8     }
9 }
10 class Teacher extends Person{
11     constructor(name, age, school){
12         super(name, age);
```

```
13     this.school = school;
14 }
15     showSchool(){
16         console.log(this.school);
17     }
18 }
19 let teacher1 = new Teacher(name, age, school);
20 teacher1.showInfo();
21 teacher1.showSchool();
```

// class 可以称为ES5的语法糖（指计算机语言中添加的某种语法，这种语法对语言的功能并没有影响，但是更方便程序员使用）。

11.Symbol

- Symbol（受保护的数据类型）是基本数据类型，表示独一无二的值，可以保证不会与其他属性名产生冲突。
- 格式：

```
1 let name = Symbol();
2 obj[name] = 'xx';
```

☆ 该属性不会出现在for...in、for...of循环中。

```
1 let obj = {
2     a:1,
3     b:2
4 }
5 let c = Symbol();
6 obj[c] = 3;
7 console.log(obj);
8 //输出结果：{a: 1, b: 2, Symbol(): 3}
9 console.log(obj[c]);
10 //输出结果：3
```

12.Set

12.1 Set

- Set是类数组，但是成员的值都是唯一的，没有重复的值。函数接受数组或类数组作为参数。
- new Set();

```
1 let arr = [1, 2, 3, 4, 5, 6, 7, 1, 2, 3];
2 console.log(arr);
3 //输出结果：( 10 ) [1, 2, 3, 4, 5, 6, 7, 1, 2, 3]
4 let setOne = new Set(arr);
5 //输出结果：Set(7) {1, 2, 3, 4, 5, 6, 7}
```

数组去重eg：

```
1 let arr = [1, 2, 3, 4, 5, 6, 7, 1, 2, 3];
2 let setOne = new Set(arr);
3 console.log(Array.from(new Set(arr)));
4 //输出结果：(7) [1, 2, 3, 4, 5, 6, 7]
```

12.2 一些方法

- add (value) : 添加某个值，返回Set结构本身。
- delete (value) : 删除某个值，返回一个布尔值，表示删除是否成功。
- has (value) : 返回一个布尔值，表示该值是否是Set成员。
- clear () : 清除所有成员，没有返回值。

12.3 遍历

- keys () : 表示值。
- entries () : 表示[值, 索引]。 (但在Set里key和value值一样)
- forEach ()
- for...of

12.4 长度

- size ()

12.5 WeakSeat

- 成员只能是对象，不能是其他值。
- add () : 增加值。

13.Map

- 类似于对象，也是键值对的集合，但“键”的范围不限于字符串，各种类型的值（包括对象）都可以当作键。
- **Map不允许key值重复。**
- new Map ()
- 一些方法：
 - .size 属性返回Map结构的成员总数
 - .set(key , value): 添加
 - .get(key)
 - .has(key)
 - .delete(key)
 - .clear()
- 遍历
- WeakMap: 只接受对象作为键名（null除外），不接受其他类型的值作为键名。（**不允许key值重复**）

14.数值的扩展

- 二进制0B 八进制0O

- Number.isFinite() (检测是否是个数字，返回布尔值) , Number.isNaN() (检测是否是数值，返回布尔值)
- Number.parseInt() (检查是否是整形，返回布尔值) , Number.parseFloat()
- Number.isInteger()
- Number.MAX_SAFE_INTEGER, Number.MIN_SAFE_INTEGER (最大/小安全范围)
- 一些方法移植到Number对象上面，行为完全保持不变。这样做的目的，是逐步减少全局性方法，使得语言逐步模块化。

15.Proxy

- 代理
- 常用拦截方法

Proxy

拦截器	作用
get	拦截对象属性的读取，比如proxy.foo和proxy['foo']
set	拦截对象属性的设置，返回一个布尔值，比如proxy.foo = v或proxy['foo'] = v
has	拦截propKey in proxy的操作，返回一个布尔值
ownKeys	拦截Object.getOwnPropertyNames(proxy)、Object.getOwnPropertySymbols(proxy)、Object.keys(proxy)、for...in循环，返回一个数组。

Proxy

拦截器	作用
deleteProperty	拦截delete proxy[propKey]的操作，返回一个布尔值
apply	拦截函数的调用、call和apply操作
construct	拦截new命令，返回一个对象

16.Reflect

Reflect

- 将Object属于语言内部的方法放到Reflect上
- 修改某些Object方法的返回结果，让其变得更合理
- 让Object操作变成函数行为
- Reflect对象的方法与Proxy对象的方法一一对应

17.异步操作前置知识

- JS是单线程的
- 同步任务（逐行执行）与异步任务（eg定时器）
- Ajax原理
 - AJAX(异步通讯手段)=异步JavaScript+XML
 - AJAX = Asynchronous JavaScript And XML.
 - AJAX不是编程语言，是一种用于创建快速动态网页的技术。
 - 特别的，如果请求本地文件，出于安全得是http请求方式，而非file协议。

```
1 // 第一步创建xmlhttprequest对象
2 var xmlhttp;
3 if (window.XMLHttpRequest) { // code for IE7+, Firefox, Chrome, Opera, Safari
4     xmlhttp = new XMLHttpRequest();
5 } else { // code for IE6, IE5
6     xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
7 }
8 // 第二步发送请求
9 xmlhttp.open("GET", "http://47.92.82.13:4000/showMessage", true);
10 xmlhttp.send();
11 // 第三回调处理
12 xmlhttp.onreadystatechange = function () {
13     if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
14         console.log(JSON.parse(xmlhttp.responseText));
15     }
16 }
```

- 关于readyState和status

XMLHttpRequest 对象属性

属性	描述
onreadystatechange	定义当 readyState 属性发生变化时被调用的函数
readyState	保存 XMLHttpRequest 的状态。 <ul style="list-style-type: none">◦ 0: 请求未初始化◦ 1: 服务器连接已建立◦ 2: 请求已收到◦ 3: 正在处理请求◦ 4: 请求已完成且响应已就绪
responseText	以字符串返回响应数据
responseXML	以 XML 数据返回响应数据
status	返回请求的状态号 <ul style="list-style-type: none">◦ 200: "OK"◦ 403: "Forbidden"◦ 404: "Not Found" <p>如需完整列表请访问 Http 消息参考手册</p>
statusText	返回状态文本 (比如 "OK" 或 "Not Found")

- 关于JSON

- 字符串 -> 对象: JSON.parse(对象)
- 对象 -> 字符串: JSON.stringify(对象)

```
1 window.onload = function(){
```

```

2 }
3 let obj = {
4   a:"abc"
5 }
6 let str = JSON.stringify(obj);
7 console.log(str);
8 console.log(JSON.parse(str));

```

- Callback Hell

18.Promise

- Promise是一个构造函数。 (异步优化解决方案，本身并不做请求)
 - promise的三种状态: pending, resolve, reject
 - new Promise(function (resolve , reject) { })
 - resolve 表示成功, 与 then 连用
 - reject 表示失败, 与 catch 连用

```

1 //成功  resolve ( then )
2 new Promise(function (resolve, reject) {
3   setTimeout(function () {
4     console.log("1");
5     resolve();
6   }, 1000)
7 }).then(function () {
8   setTimeout(function () {
9     console.log("2");
10  }, 1000)
11 })
12
13 //失败  reject ( catch )
14 new Promise(function (resolve, reject) {
15   setTimeout(function () {
16     console.log("1");
17     reject();
18   }, 1000)
19 }).then(function () {
20   setTimeout(function () {
21     console.log("2");
22   }, 1000)
23 }).catch(function () {
24   console.log("erro");
25 })

```

- 多层简化封装

```

1 function proFun(val){
2   return new Promise(function(resolve,reject){
3     setTimeout(function(){
4       console.log(val);
5       resolve();

```

```

6     },1000)
7   })
8 }
9 proFun(1).then(function(){
10   return proFun(2);
11 }).then(function(){
12   return proFun(3);
13 }).then(function(){
14   return proFun(4);
15 })

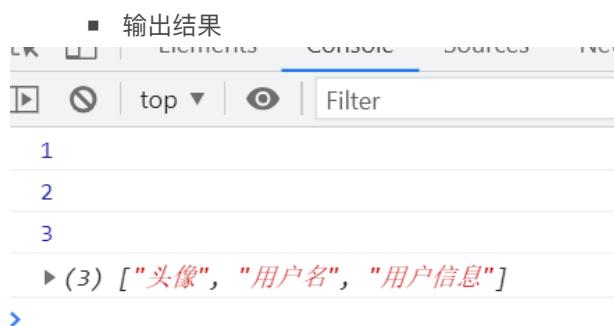
```

- Promise.all()

```

1 let p1 = new Promise(function(resolve, reject){
2   setTimeout(function(){
3     console.log(1);
4     resolve();
5   },1000)
6 })
7 let p2 = new Promise(function(resolve, reject){
8   setTimeout(function(){
9     console.log(2);
10    resolve();
11   },2000)
12 })
13 let p3 = new Promise(function(resolve, reject){
14   setTimeout(function(){
15     console.log(3);
16     resolve();
17   },3000)
18 })
19 let arr = [p1, p2, p3];
20 Promise.all(arr).then(function(res){
21   console.log(res);
22 }).catch(function(){
23   console.log("error")
24 })

```



- Promise.race()

```

1 let p1 = new Promise(function(resolve, reject){

```

```

2   setTimeout(function(){
3     console.log(1);
4     resolve();
5   },1000)
6 })
7 let p2 = new Promise(function(resolve, reject){
8   setTimeout(function(){
9     console.log(2);
10    resolve();
11  },2000)
12 })
13 let p3 = new Promise(function(resolve, reject){
14   setTimeout(function(){
15     console.log(3);
16     resolve();
17  },3000)
18 })
19 let arr = [p1, p2, p3];
20 Promise.race(arr).then(function(res){
21   console.log(res);
22 }).catch(function(){
23   console.log("error")
24 })

```

■ 输出结果

1
头像
2
3
▶

- Promise.all()和Promise.race()的区别:
 - all: 一个失败, 认定全部失败
 - race: 第一条请求成功即成功

19.Generator

- 说一步走一步。 (wjjw—戳一蹦跶同理)
- 一个 yield 一个 next
- 执行完了console.log(g.next())返回“done: true”
- next传参的时候是传给上一个yield
- yield 传出的数据为 NaN

```
1 function* foo(x){
```

```

2 var y = 2 * (yield (x + 1));
3 console.log(y); //结果4  ?
4 var z = yield (y / 3);
5 return (x + y + z );
6 }
7 var a = foo(5);
8 console.log(a.next());
9 console.log(a.next(2));
10 console.log(a.next(1));

```

20.async_await (ES2017)

- 是Promise的语法糖。

```

1 async function foo(){
2     await new Promise(function(resolve, reject){
3         setTimeout(function(){
4             console.log(1);
5             resolve();
6         },1000)
7     })
8     await new Promise(function(resolve, reject){
9         setTimeout(function(){
10            console.log(2);
11            resolve();
12        },1000)
13    })
14    console.log("3");
15 }
16 foo();

```

21.babel

21.1 环境安装

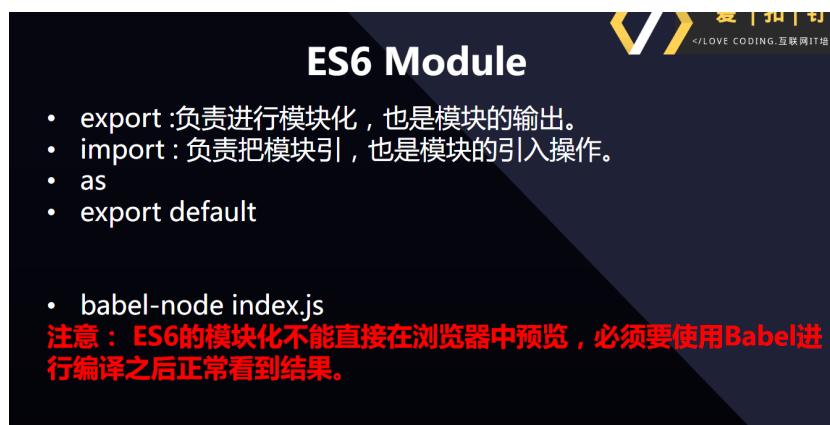


21.2 文件转化(ES6 → ES5)

- 文件: babel src/index1.js -o dist/index1.js
- 文件夹: babel src -d dist

3. 实时监控: babel src -w -d dist

22.ES6 Module



ES6 Module

- `export` :负责进行模块化，也是模块的输出。
- `import` : 负责把模块引，也是模块的引入操作。
- `as`
- `export default`

• `babel-node index.js`

注意：ES6的模块化不能直接在浏览器中预览，必须要使用Babel进行编译之后正常看到结果。

23.深/浅拷贝

- 深拷贝：修改新变量的值不会影响原有变量的值。默认情况下基本数据类型（number, string, null, undefined, boolean）都是深拷贝。
- 浅拷贝：修改新变量的值会影响原有的变量的值。默认情况下引用类型（object）都是浅拷贝。