目录

[声明变量 5](#_Toc4942373)

[声明变量的关键字 5](#_Toc4942374)

[let特性 5](#_Toc4942375)

[const特性 6](#_Toc4942376)

[变量声明总结 6](#_Toc4942377)

[解构赋值 7](#_Toc4942378)

[对象的解构赋值 7](#_Toc4942379)

[默认值 7](#_Toc4942380)

[别名 7](#_Toc4942381)

[数组的解构赋值 8](#_Toc4942382)

[字符串的解构赋值 8](#_Toc4942383)

[函数参数的解构赋值 8](#_Toc4942384)

[不完全解构 9](#_Toc4942385)

[如果等号的右边是不可遍历的结构 9](#_Toc4942386)

[解构用途 9](#_Toc4942387)

[模板字符串 10](#_Toc4942388)

[模板字符串中插入变量和表达式${} 10](#_Toc4942389)

[箭头函数 11](#_Toc4942390)

[语法形式 11](#_Toc4942391)

[箭头函数的特性 11](#_Toc4942392)

[扩展运算符（…） 13](#_Toc4942393)

[数组的扩展 13](#_Toc4942394)

[对象的扩展 13](#_Toc4942395)

[应用：取代apply方法 13](#_Toc4942396)

[应用：求最大值Math.max() 13](#_Toc4942397)

[应用：与解构赋值结合，生成新数组 14](#_Toc4942398)

[应用：将字符串转为真正的数组 14](#_Toc4942399)

[将实现了 Iterator（迭代） 接口的对象转为数组 14](#_Toc4942400)

[总结：扩展运算符的作用 14](#_Toc4942401)

[函数的扩展 15](#_Toc4942402)

[参数设置默认值 15](#_Toc4942403)

[rest参数 15](#_Toc4942404)

[对象的扩展 16](#_Toc4942405)

[对象简洁表示法 16](#_Toc4942406)

[对象的新增方法 17](#_Toc4942407)

[Object.keys() 17](#_Toc4942408)

[Object.values() 17](#_Toc4942409)

[Object.entries() 17](#_Toc4942410)

[Object.fromEntries() 17](#_Toc4942411)

[Object.assign() 17](#_Toc4942412)

[数组的迭代方法 18](#_Toc4942413)

[ES5中的迭代方法 18](#_Toc4942414)

[ES5中的归并方法 19](#_Toc4942415)

[ES6中的迭代方法 20](#_Toc4942416)

[Set 和 Map定义新的数据结构 21](#_Toc4942417)

[Set数据结构 21](#_Toc4942418)

[利用set数据结构给数组去重 21](#_Toc4942419)

[set实例的方法 21](#_Toc4942420)

[Map数据结构 24](#_Toc4942421)

[创建map数据 24](#_Toc4942422)

[Map实例下的方法-set/get 24](#_Toc4942423)

[Map实例下的方法-delete 24](#_Toc4942424)

[遍历接口iterator 25](#_Toc4942425)

[弱引用WeakMap/WeakSet 26](#_Toc4942426)

[WeakMap 27](#_Toc4942427)

[Symbol 28](#_Toc4942428)

[理解 28](#_Toc4942429)

[应用场景 28](#_Toc4942430)

[带来的问题 28](#_Toc4942431)

[面向对象 29](#_Toc4942432)

[ES5中面向对象的写法 29](#_Toc4942433)

[原型链 29](#_Toc4942434)

[原型对象下的属性和方法 30](#_Toc4942435)

[constructor属性 30](#_Toc4942436)

[不经意修改了constructor 30](#_Toc4942437)

[hasOwnProperty（）方法 30](#_Toc4942438)

[独特的toString()方法 31](#_Toc4942439)

[toString()方法的作用 31](#_Toc4942440)

[对象的继承 32](#_Toc4942441)

[拷贝继承 32](#_Toc4942442)

[组合继承 33](#_Toc4942443)

[原型继承 34](#_Toc4942444)

[类class 35](#_Toc4942445)

[class的静态方法和属性 36](#_Toc4942446)

[静态方法 36](#_Toc4942447)

[静态属性 36](#_Toc4942448)

[class的继承 37](#_Toc4942449)

[Promise 38](#_Toc4942450)

[比较同步和异步 38](#_Toc4942451)

[resolve/reject/then 38](#_Toc4942452)

[简单使用 38](#_Toc4942453)

[解决地狱回调-Promise.all().then() 39](#_Toc4942454)

[封装promise 40](#_Toc4942455)

[利用jquery的ajax返回promise 40](#_Toc4942456)

[竞速Promise.race() 40](#_Toc4942457)

[生成器函数generator 41](#_Toc4942458)

# 声明变量

## 声明变量的关键字

var let const

var声明变量带来的问题

1. 变量提升
2. 同一个作用下可以声明多次
3. 声明的变量是window对象上的属性，如果window对象下的属性和变量同名，变量将覆盖原有属性

console.log(a);//undefined

console.log(a++);//NaN

var a=10;

var a=20;

var alert=30;

## let特性

1. 不允许重复声明
2. 不存在变量提升
3. 块级作用域 （从块的开始到声明的这段区域） 暂存死区

{

var web=10;

let js=20;

}

console.log(web);

console.log(js);

//let解决闭包的问题

for(let i=0; i<lis.length; i++){

lis[i].onclick=function(){

alert(i);

}

}

## const特性

1. 同let特性(不能重复声明，不提升)
2. 不同于let的特性有，声明时必须赋值，赋值后不能修改
3. 但是如果声明的常量是一个对象或数组，那么对象或数组里面的数据是可以再次修改的。

const o=[1,2,3];

o[0]='m';

console.log(o);

## 变量声明总结

var声明的变量由函数划分作用域

let const声明的变量由代码块{}划分作用域

# 解构赋值

概念：ES6允许按照一定的模式，从对象或数组或字符串中提取值，对变量进行赋值，这种行为被称作解构

## 对象的解构赋值

let tools={

add:function(){},

isFunction:function(paras){

console.log(paras);

},

isArray:function(){}

}

/\* var add=tools.add;

var isFunction=tools.isFunction;

var isArray=tools.isArray; \*/

let {add,isFunction,isArray}=tools;//解构赋值

isFunction(1);

如果

let obj={a:1}

let a=””;

{a}=obj;//这样写是不行的

({a}=obj;)//要在外围整体加一个括号才行

## 默认值

let {a=”zhagngsan”,add,isFunction,isArray}=tools;//解构赋值

## 别名

let {a:name,add,isFunction,isArray}=tools;//解构赋值

## 数组的解构赋值

var [a,b,c]=[1,2,3];

console.log(a,b,c);//1 2 3

var [a,b,c=100]=[1,2];

console.log(a,b,c);//1 2 100

本质上这种写法属于‘模式匹配‘，只要等号两边的模式相同，左边的变量就会被赋予对应的值

## 字符串的解构赋值

var [a,b,c]="devWebServer";

console.log(a,b,c);//d e v

let { length : len} = 'yahooa';

console.log(len);//6

## 函数参数的解构赋值

function move({x = 0,y = 0}={}){

return [x,y];

}

console.log(move({x : 3,y : 4})); //[3,4]

console.log(move({x : 3})); //[3,0]

console.log(move({})); //[0,0]

console.log(move()); //[0,0]

//move()的参数是一个对象，通过对这个对象进行解构，得到变量x、y的值，如果解构失败，x和y 等于默认值

function move2({x,y} = {x : 1, y : 2 }){

return [x,y];

}

console.log(move2({x : 6,y : 8})); //[6,8]

console.log(move2({})); //[undefined,undefined]

console.log(move2()); //[1,2]

//move2() 是为函数move2的参数指定默认值，而不是为变量x和y指定默认值，所以与前一种写法的结果不太一样，undefined 就会触发函数的默认值

## 不完全解构

let [x,y] = [1,2,3];

console.log(x,y); //1 2

let [a,[b],d] = [1,[2,3],4];

console.log(a,b,d); //1 2 4

## 如果等号的右边是不可遍历的结构

let [a] = 1;

let [b] = false;

let [c] = NaN;

let [d] = undefined;

let [e] = null;

let [f] = {};

//上面的语句都会报错：Uncaught TypeError: 1 is not iterable

iterable(遍历)

## 解构用途

交换变量的值

从函数返回多个值

函数参数的定义

提取json数据

函数参数的默认值

输入模块的指定用法

# 模板字符串

模板字符串，是增强版的字符串，用反引号``表示，可以放入换行字符串（功能）

<script>

let str=`

<ul>

<li></li>

<li></li>

</ul> `

</script>

## 模板字符串中插入变量和表达式${}

<script>

var a="hello";

var str=`

<ul>

<li>${a}</li>

<li>${[1,2,3].push(4)}</li>

</ul>

`;

console.log(str);

</script>

# 箭头函数

## 语法形式

()=>{}

()=>123 // 123为箭头函数的返回值

a=>123+a // a表示形参，只有一个参数可以不写括号，123+a作为返回值

(a,b)=>a+b // a,b两个都是形参，必须加括号

(a,b)=>( { num:a+b } ) // 返回一个对象数据，这个时候{}外面必须放一个()

<script>

/\*let f=()=>{

console.log(123);

}

f();\*/

/\*let f=()=>123;

console.log(f());\*/

/\*let f=a=>123+a;

console.log(f(10));\*/

/\*let f=(a,b)=>a+b;

console.log(f(10,20));\*/

/\*let f=(a,b)=>({num:a+b});

console.log(f(10,20));\*/

let f=(a,b)=>{

if(a<10)

{

a=100;

}

return a+b;

};

console.log(f(1,20));

</script>

## 箭头函数的特性

1. **函数体内的this指向的是定义函数时作用域的this 箭头函数绑定了父上下文**
2. 不可以当做构造函数
3. 不可以使用arguments对象

document.onclick=function(){

console.log(this);//document

setTimeout(()=>{

console.log(this);//document

},1000);

}

document.onclick=()=>{

console.log(this);//window

setTimeout(()=>{

console.log(this);//window

},1000);

}

# 扩展运算符（…）

## 数组的扩展

将一个数组转换为用逗号隔开的参数序列

<script>

var arr1=[1,2,3];

var arr2=[4,5,6];

var arr=[...arr1]; //复制了一份

console.log(arr);

var arr=[...arr1,...arr2]; //合并两份

console.log(arr);

</script>

## 对象的扩展

用于取出参数对象的所有可遍历属性，拷贝到当前对象中

<script>

var o1={a:1};

var o2={b:2};

var o={...o1};//复制了一份

console.log(o);

var o={...o1,...o2};//合并两份

console.log(o);

</script>

## 应用：取代apply方法

function fn(x,y,z) {

console.log(x,y,z);

}

var arrs=[1,2,3];

//fn.apply(null,arrs);

fn(...arrs);

## 应用：求最大值Math.max()

// ES5 的写法

Math.max.apply(null, [14, 3, 77])

// ES6 的写法

Math.max(...[14, 3, 77])

// 等同于

Math.max(14, 3, 77);

## 应用：与解构赋值结合，生成新数组

const [first, ...rest] = [1, 2, 3, 4, 5];

first // 1

rest // [2, 3, 4, 5]

const [first, ...rest] = [];

first // undefined

rest // []:

const [first, ...rest] = ["foo"];

first // "foo"

rest // []

## 应用：将字符串转为真正的数组

[...'hello']

// [ "h", "e", "l", "l", "o" ]

## 将实现了 Iterator（迭代） 接口的对象转为数组

var nodeList = document.querySelectorAll('div');

var array = [...nodeList];

## 总结：扩展运算符的作用

1. 用来遍历数组、对象、字符串，把他们变成逗号隔开的数据。
2. 把逗号隔开的数据变成真正的数组。
3. 把类数组转换成真正的数组。

# 函数的扩展

## 参数设置默认值

函数调用时有值就取调用的值，否则用默认值看看下面两种操作

function fn(a) {

a=a||10;

console.log(a);

}

fn(1);//1

fn(0);//10//这里没取我们传递的值

fn('');//10//这里没取我们传递的值

function fn2(a=10) {//这里是给形参直接赋一个默认值

//let a=5;//会报错 Uncaught SyntaxError: Identifier 'a' has already been declared

console.log(a);

}

fn2(1);//1 fn2(0);//0 fn2('');//

**函数内部不能再用let或const再声明同样的形参**

## rest参数

function fn(...arr){//此处...arr叫rest参数，arr得到的是一个真数组

console.log(arr);

}

fn(1);//[1]

fn(1,2);[1,2]

fn(1,2,3);[1,2,3]

确定几个实参

function fn(a,b,...arr){//此处a,b会找前面两个实参,剩余的都给arr

console.log(a); console.log(b); console.log(arr);

}

fn(1);

fn(1,2);

fn(1,2,3);

fn(1,2,3,4,5,6,7,8,9);

# 对象的扩展

## 对象简洁表示法

属性名和变量名相同，可以直接写上变量名

方法可以省略function关键字

var web="hello";

var o={

web, //相当于 web:web

fn(){ //相当于 fn:function(){alert(1)}

alert(1);

}

}

console.log(o.web); o.fn();

var util=(function(){

function add(){};

function isFunction(){};

/\* return{

add:add,

isFunction:isFunction

}; //无需再这样写了 \*/

return{

add,

isFunction

}

})()

# 对象的新增方法

## Object.keys()

ES5 引入了Object.keys方法，返回一个数组，成员是参数对象自身的（不含继承的）所有可遍历（enumerable）属性的键名。

var obj = { foo: 'bar', baz: 42 };

Object.keys(obj)

// ["foo", "baz"]

## Object.values()

Object.values方法返回一个数组，成员是参数对象自身的（不含继承的）所有可遍历（enumerable）属性的键值。

const obj = { foo: 'bar', baz: 42 };

Object.values(obj)

// ["bar", 42]

## Object.entries()

Object.entries()方法返回一个数组，成员是参数对象自身的（不含继承的）所有可遍历（enumerable）属性的键值对数组。

const obj = { foo: 'bar', baz: 42 };

Object.entries(obj)

// [ ["foo", "bar"], ["baz", 42] ]

除了返回值不一样，该方法的行为与Object.values基本一致。

## Object.fromEntries()

该方法是Object.entries()的逆操作，用于将一个键值对数组转为对象。

Object.fromEntries([

['foo', 'bar'],

['baz', 42]

])

// { foo: "bar", baz: 42 }

## Object.assign()

合并对象：Object.assign(目标对象，源对象1，源对象2…)

var o1={a:1,b:4,c:5,d:function(){}}

var o2={b:2}

var o3={c:3}

Object.assign(o1,o2,o3);

console.log(o1);// {a: 1, b: 2, c: 3, d: ƒ}

# 数组的迭代方法

## ES5中的迭代方法

ECMAScript 5 为数组定义了5 个迭代方法。每个方法都接收两个参数：

1. 要在每一项上运行的函数
2. （可选的）运行该函数的作用域对象——影响this的值。

传入这些方法中的函数会接收三个参数：

1. 数组项的值
2. 该项在数组中的位置
3. 和数组对象本身。

根据使用的方法不同，这个函数执行后的返回值可能会也可能不会影响方法的返回值。

以下是这5 个迭代方法的作用。

every()：对数组中的每一项运行给定函数，如果该函数对**每一项都返回true，则返回true**。

filter()：对数组中的每一项运行给定函数，**返回**该函数会返回true的项组成的**数组**。 forEach()：对数组中的每一项运行给定函数。这个方法**没有返回值**。

map()：对数组中的每一项运行给定函数，**返回**每次函数调用的结果组成的**数组**。

some()：对数组中的每一项运行给定函数，如果该函数对**任一项返回true，则返回true**。

以上方法都**不会修改数组中的包含的值**。

var numbers = [1,2,3,4,5,4,3,2,1];

var filterResult = numbers.filter(function(item, index, array){

**return (item > 2);**

});

alert(filterResult); //[3,4,5,4,3]

var numbers = [1,2,3,4,5,4,3,2,1];

var mapResult = numbers.map(function(item, index, array){

**return item \* 2;**

});

alert(mapResult); //[2,4,6,8,10,8,6,4,2]

## ES5中的归并方法

ECMAScript 5 还新增了两个归并数组的方法：reduce()和reduceRight()。

这两个方法**都会迭代数组的所有项**，然后构建**一个最终返回的值**。

reduce()方法从数组的第一项开始，逐个遍历到最后。

reduceRight()则从数组的最后一项开始，向前遍历到第一项。

这两个方法都接收两个参数：一个在每一项上调用的函数和（可选的）作为归并基础的初始值。

reduce()和reduceRight()的函数接收4 个参数：前一个值、当前值、项的索引和数组对象

这个函数返回的任何值都会作为第一个参数自动传给下一项。第一次迭代发生在数组的第二项上，因此第一个参数是数组的第一项，第二个参数就是数组的第二项。

var values = [1,2,3,4,5];

var sum = values.reduce(function(prev, cur, index, array){

**return prev + cur;**

});

alert(sum); //15

var values = [1,2,3,4,5];

var sum = values.reduceRight(function(prev, cur, index, array){

**return prev + cur;**

});

alert(sum); //15

## ES6中的迭代方法

**找到第一个符合条件的数组元素（find）**

var arr3=[1,2,3,4,5,4,3,2,1];

var value=arr3.find(function(item,index,arr){

if(index>3) {

return item>3;

}

})

console.log(value);//5

**找到第一个符合条件的数组元素的索引值（findIndex）**

let arr = [1,2,3,4,5]

let arr1 = arr.findIndex((value, index, array) => value > 3)

console.log(arr1) // 3

**使用给定的值，填充一个数组arr.fill(target, start, end)**

**ps:填充完后会改变原数组**

　　参数： target -- 待填充的元素

　　　　　 start -- 开始填充的位置-索引

　　　　    end -- 终止填充的位置-索引（不包括该位置)

let arr = [1,2,3,4,5]

let arr3 = arr.fill(5,1,3)

console.log(arr3)// [1, 5, 5, 4, 5]

**判断数中是否包含给定的值arr.includes()**

let arr = [1,2,3,4,5]

let arr1 = arr.includes(2)

console.log(arr1) // ture

let arr2 = arr.includes(9)

console.log(arr2) // false

let arr3 = [1,2,3,NaN].includes(NaN)

console.log(arr3) // true

ps:与indexOf()的区别：

1 indexOf()返回的是数值，而includes()返回的是布尔值

2 indexOf() 不能判断NaN，返回为-1 ，includes()则可以判断

# Set 和 Map定义新的数据结构

## Set数据结构

它类似于数组，但是**成员的值都是唯一的，没有重复的值**。

Set**本身是一个构造函数**，用来生成 Set 数据结构

var s = new Set(["a","b","c"]);

console.log(s);// Set(3) **{"a", "b", "c"}**

console.log(s.size);//3

## 利用set数据结构给数组去重

var arr = [1,2,3,2,3, 4, 5,"1",NaN,NaN];

var s = new Set(arr);

console.log( s )// Set(7) {1, 2, 3, 4, 5, "1",NaN }

利用…扩展运算符把set数据结构变回数组

console.log( [...s] );

## set实例的方法

add():增加数据到set结构，返回新的set数据，**不会增加相同的数据**。

delete():删除set结构中的数据，返回true或false

var s = new Set(["a","b","c"]);

s.add( 1 ).add( 2 ).add( 1 ).add( NaN ).add( NaN );

var res1 = s.delete( "b" );

var res2 = s.delete( "z" );

console.log(res1,res2,s);//true false Set(5) {"a", "c", 1, 2, NaN}

has():测试set结构中是否包含某个数据，返回true或false；

clear():清除set中的所有数据，无返回值，会修改原对象；

var s = new Set(["a","b","c"]);

console.log(s.has("a"),s.has("x"))//true false

s.clear();

console.log(s);//Set(0) {}

forEach():遍历set数据结构，回调函数的参数item，index，set分别表示：数据、数据的索引、set数据

s.forEach( function(item,index,set){

  console.log( item,index,set )

} )

//a a Set(3) {"a", "b", "c"}

//b b Set(3) {"a", "b", "c"}

//c c Set(3) {"a", "b", "c"}

keys() values() 遍历set数据结构

var s = new Set(["a","b","c"]);

var keys = s.keys();

console.log( keys );

console.log( keys.next() )

console.log( keys.next() )

console.log( keys.next() )

console.log( keys.next() )

var values = s.values();

console.log( values )

console.log( values.next() )

console.log( values.next() )

console.log( values.next() )

console.log( values.next() )

控制台显示的结果：

/\*

SetIterator {"a", "b", "c"}

{value: "a", done: false}

{value: "b", done: false}

{value: "c", done: false}

{value: undefined, done: true}

SetIterator {"a", "b", "c"}

{value: "a", done: false}

{value: "b", done: false}

{value: "c", done: false}

{value: undefined, done: true}

\*/

entries方法返回的遍历器，同时包括键名和键值，所以每次输出一个数组，它的两个成员完全相等

var s = new Set(["a","b","c"]);

var entries = s.entries();

console.log( entries )

console.log( entries.next() )

console.log( entries.next() )

console.log( entries.next() )

console.log( entries.next() )

控制台显示结果：

/\*

SetIterator {"a", "b", "c"}

{value: Array(2), done: false}

{value: Array(2), done: false}

{value: Array(2), done: false}

{value: undefined, done: true}

\*/

## Map数据结构

Map数据结构出现的背景：JavaScript传统上只能用字符串当作键。这给它的使用带来了很大的限制。例如：用一个对象作为键名

let obj={clsn:'person'};

let person={name:'lc'};

obj[person]="aaa";

console.log(obj);// {clsn: "person", [object Object]: "aaa"}

### 创建map数据

var m=new Map([['name','lc'],['age',30]]);

console.log(m);//Map(2) {"name" => "lc", "age" => 30}

console.log(m.size);//2

### Map实例下的方法-set/get

var m=new Map([['name','lc'],['age',30]]);

m.set('sex','man');

var s=m.get('sex');

console.log(m,s);//Map(3) {"name" => "lc", "age" => 30, "sex" => "man"} "man"

不同于set的是，set是用add方法添加，map是用set添加，用get获取键值

### Map实例下的方法-delete

var m=new Map([['name','lc'],['age',30]]);

m.set('sex','man');

var d=m.delete('sex');

var a=m.delete('age');

console.log(m,d,a);//Map(1) {"name" => "lc"} true true

不同与set数据结构的是，map是通过键名，删除键值对

Map的其他方法：has() forEach() keys() values() entryies() 与get相同 只是多了一个键值的控制。

# 遍历接口iterator

检查一个对象是否部署了iterator，以数组为例。

var arr = [10,20,30];

console.log( arr[Symbol.iterator] );

// ƒ values() { [native code] }

如果输出一个向上面的函数，说明这个对象上有iterator接口

检查对象和Map数据是否有iterator接口

var obj={a:1,b:2,c:3};

console.log(obj[Symbol.iterator]);//undefined

var m=new Map([["a",1],["b",2],["c",3]]);

console.log(m[Symbol.iterator]);//ƒ entries() { [native code] }

如果该对象部署了遍历接口，就可以使用ES6提供的（keys、values、entries方法遍历）

var arr = [10,20,30];

console.log( arr[Symbol.iterator] );

var k = arr.keys();

console.log( k.next() );

console.log( k.next() );

console.log( k.next() );

console.log( k.next() );

如果该对象部署了遍历接口，也可以用ES6提供的for(value of obj){}遍历

var arr = [10,20,30];

for( var attr of arr ){

    console.log( attr );// 10 20 30

}

for( var attr in arr ){

    console.log( attr );// 0 1 2

}

总结： 数组，类数组及Set和Map数据 上是部署了iterator接口的。

# 弱引用WeakMap/WeakSet

理解下强引用

var m=new Map();

var personA={

name:'zs',

age:30

}

var personB={

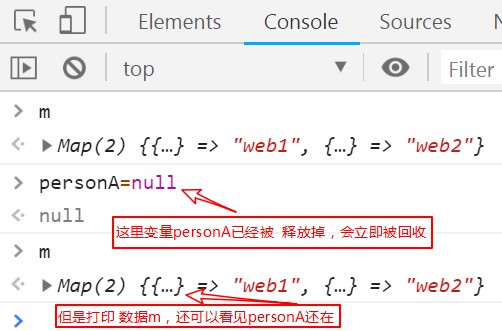
name:'ls',

age:20

}

m.set(personA,'web1');

m.set(personB,'web2');





也就是说垃圾回收机制虽然把personA给回收了，但是 Map数据中是强引用，所以打印m还能看见personA的数据。

## WeakMap

var wm=new WeakMap();

var personA={

name:'zs',

age:30

}

var personB={

name:'ls',

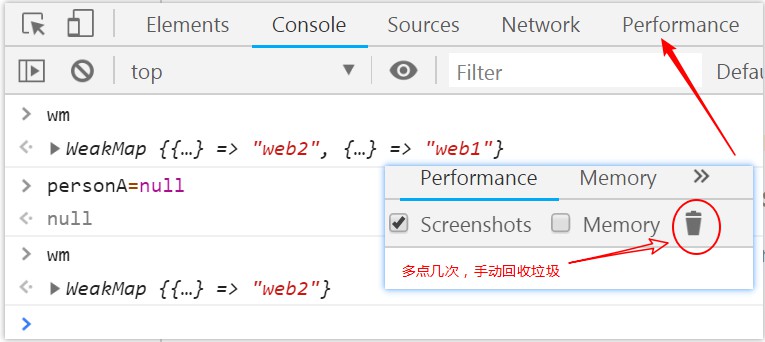
age:20

}

wm.set(personA,'web1');

wm.set(personB,'web2');

弱引用，在把变量设为空值(null)之后，我们手动回收垃圾，发现wm里面少了一组键值对。这样有利于释放内存，防止内存泄漏！



WeakMap 与 Map 在 API 上的区别主要是两个:

一是没有遍历操作（即没有keys()、values()和entries()方法），也没有size属性。

二是无法清空，即不支持clear方法。

因此，WeakMap只有四个方法可用：get()、set()、has()、delete()。

WeakSet的使用同上，这里就不在过多讲解。

# Symbol

ES6 引入了一种新的原始数据类型Symbol，表示独一无二的值。它是 JavaScript 语言的第七种数据类型。

## 理解

var a=Symbol();

var b=Symbol();

console.log(typeof a)//symbol

console.log(a,b);//Symbol() Symbol()

console.log(a===b);//false//可以把a,b理解为两个随机数

var m=Symbol('m');//参数m是做一个标识，方便使用

var n=Symbol('n');

console.log(m,n)//Symbol(m) Symbol(n)

## 应用场景

//后台同事给我们的数据

var obj={ a:1,b:2 }

//拿到数据后我们给obj也添加了一个属性a

//obj.a=2;

//console.log(obj);//{ a:2,b:2 }//这样会修改掉原来的数据

var a=Symbol('k');

obj[a]=2;

console.log(obj);//{ a:1,b:2,Symbol(k):2 }

console.log(obj[a]);//2

console.log(obj['a']);//1

## 带来的问题

使用for…in语句找不到键名为symbol数据的键值

var obj={a:1,b:2};

//使用另一种方法，添加键值对

var a=Symbol('a');

obj={ [a]:"abc",a:1,b:2 };

var b=Symbol('b');

obj[b]='123';

for(attr in obj){

  console.log(obj[attr]);//1 2 找不到'abc' 123

}

var key=Object.getOwnPropertySymbols(obj); //找obj中symbol类型的键名

console.log(key instanceof Array);//true key是一个数组

console.log(key);

//会按照添加键值对的顺序找

console.log(obj[key[0]]);//abc

console.log(obj[key[1]]);//123

# 面向对象

面向对象的特征：抽象、封装、继承、多态、重载

## ES5中面向对象的写法

(推理过程课堂演示)

function Person(name,age){

  this.name = name;

  this.age = age;

}

Person.prototype.say = function(){

  console.log( "我会说话..." )

}

var p1 = new Person( "web",30 );

缺陷：代码分成两块，不便于代码的逻辑管理

## 原型链

实例对象与原型之间的链接，叫做原型链（也叫隐式原型\_\_proto\_\_）

原型链的最外层 : Object.prototype

function Fn(){}

Fn.prototype.num = 10;

var f = new Fn();

alert(f.num);//这里就是通过原型链，拿到到num的 10

function Fn(){  this.num = 20; }

Fn.prototype.num = 10;

var f = new Fn();

alert(f.num);//这里都定义了num, 那么构造函数中的num优先级高 20

function Fn(){  //this.num = 20; }

//Fn.prototype.num = 10;

Object.prototype.num = 30;

var f = new Fn();

alert(f.num);//30



总结：原型对象上有的属性和方法，实例对象都可以根据原型链找到，如果过冲突就看谁先出现，谁先出现就用谁的。（有指定的用指定的，无指定继承最近的）

## 原型对象下的属性和方法

### constructor属性

每个构造函数都有一个原型对象，该对象下有一个默认属性constructor指向该构造函数。那么实例对象可以通过原型链也找到该属性。

function Fn(){}

var f=new Fn();

console.log(Fn.prototype.constructor);//Fn

console.log(f.constructor)//Fn

var obj={name:'lc'}

console.log(obj.constructor);//?

var arr=[];

console.log(arr.constructor);//?

利用该属性可以检测实例对象是不是由该构造函数实现的。（检测对象数据类型）

**注意：constructor属性不能被for…in遍历的到**

### 不经意修改了constructor

function Fn(){}

// Fn.prototype.name = '小明';

// Fn.prototype.age = 20;

Fn.prototype = {

  //constructor : Fn,

  name : '小明',

  age : 20

};

var f = new Fn();

console.log( f.constructor );//?

### hasOwnProperty（）方法

每个构造函数的原型对象下都有一个继承自Object对象下的hasOwnProperty()方法，该方法是用来测试自己身上是不是包含该属性。如果包含则返回true，不包含则返回false。参数是字符串形式的属性。

var obj={name:'lc'}

Object.prototype.name="abc";

console.log(obj.hasOwnProperty==Object.prototype.hasOwnProperty);//?

console.log(obj.hasOwnProperty('name'));//?

console.log(Object.prototype.hasOwnProperty('name'));//?

## 独特的toString()方法

本地对象下面都是自带的 , 自己写的对象都是通过原型链找object下面的

var arr = [];

alert( arr.toString == Object.prototype.toString ); //false

//这个arr.toString其实是原型对象Array.prototype.toString

function Fn(){ }

var f = new Fn();

alert( f.toString == Object.prototype.toString ); //true

### toString()方法的作用

1、把对象转成字符串

var arr = [1,2,3];

//改写本地对象下的toString方法

Array.prototype.toString = function(){

  return this.join('+');

};

alert( arr.toString() ); //'1+2+3'

2、进制转换

var num = 255;

alert( num.toString(16) ); //'ff'

3、判断对象的数据类型

var arr = [];

// alert( Object.prototype.toString.call(arr) )

alert( Object.prototype.toString.call(arr) == '[object Array]' ); //'[object Array]'

检测对象的数据类型的三种方法：

arr.constructor==Array

arr instanceof Array

Object.prototype.toString.call(arr) == '[object Array]'

## 对象的继承

**【什么是继承】**

在原有对象的基础上，略作修改，得到一个新的对象 ， 不影响原有对象的功能

**【为什么要学继承】**继承的作用：代码复用

**【如何实现继承】** 属性的继承、方法继承

### 拷贝继承

function CreatePerson(name,sex){ //父类

  this.name = name;

  this.sex = sex;

}

CreatePerson.prototype.showName = function(){

  alert( this.name );

};

var p1 = new CreatePerson('小明','男');

//p1.showName();

function CreateStar(name,sex,job){ //子类

  CreatePerson.call(this,name,sex);

  this.job = job;

}

//CreateStar.prototype = CreatePerson.prototype;//浅拷贝

extend( CreateStar.prototype , CreatePerson.prototype );

CreateStar.prototype.showJob = function(){

};

var p2 = new CreateStar('黄晓明','男','演员');

p2.showName();

function extend(obj1,obj2){

  for(var attr in obj2){

    obj1[attr] = obj2[attr]; //深拷贝

  }

}

总结：拷贝继承call修改this指向，for…in深拷贝。

### 组合继承

利用原型链继承

function A(){

this.name="a";

}

A.prototype.sayName=function(){

console.log(this.name);

}

function B(){}

B.prototype=new A();//这里主要是通过原型链实现继承

var b1=new B();

console.log(b1.name);//继承属性

b1.sayName();//继承方法



但是这样继承会带来问题：

1、b1实例对象的constructor会变成了A

2、如果再new一个b2,那么b2的属性和b1的属性继承的值如果是对象，那么他们之间将存在引用关系，耦合度比较大。

function A(){ this.arr=[1,2,3];}

function B(){}

B.prototype=new A();

var b1=new B();

b1.arr.push(4);

console.log(b1.constructor) //function A(){}

console.log(b1.arr);//[1,2,3,4]

var b2=new B();

console.log(b2.arr);//[1,2,3,4]

解决：

function A(){ this.arr=[1,2,3]; }

function B(){ A.call(this) }

B.prototype=new A();

B.prototype.constructor=B;

var b1=new B();

b1.arr.push(4);

console.log(b1.arr);//[1,2,3,4]

var b2=new B();

console.log(b2.arr);//[1,2,3]

### 原型继承

要求你必须有一个对象可以作为另一个对象的基础。

let person={

  name:'aa',

  loves:['打球','游泳']

}

function object(obj){

  function Fn(){};

  Fn.prototype=obj; //对传入其中的对象执行了一次浅复制

  return new Fn();

}

let A=object(person);

A.loves.push('旅游');

console.log(A.name);

console.log(A.loves);

let B=object(person);

B.name='bbb';

console.log(B.name);

console.log(B.loves);

#### Object.create()

ECMAScript 5 通过新增Object.create()方法规范化了原型式继承。这个方法接收两个参数：一个用作新对象原型的对象和（可选的）一个为新对象定义额外属性的对象。在传入一个参数的情况下，Object.create()与object()方法的行为相同。

let person={  name:'aa',  loves:['打球','游泳'] }

let A=Object.create(person);

console.log(A.name);//aa

console.log(A.loves);// ["打球", "游泳"]

let B=Object.create(person);

console.log(B.name);//aa

console.log(B.loves);// ["打球", "游泳"]

let C=Object.create(person,{

  name:{value:'cc'},

  sayLove:{

    value:function(){

      console.log(this.loves)

      }

    }

  });

console.log(C.name);

C.sayLove();

# 类class

ES6的写法

class Person{

  constructor(name,age){

    this.name = name;

    this.age = age;

  }

  say(){

    console.log( "say...." )

  }

}

var p1 = new Person( "web",27 );

console.log( typeof Person )//function

//say方法还是在构造函数Person的原型上

console.log( Person.prototype )// {constructor: ƒ, say: ƒ}

Person("web",27) //如果你这样调用它会报错，一定要new 调用

在构造函数Person的原型方法中使用this，this是指向new出来的实例的

class Person{

  constructor(name,age){

    this.name = name;

    this.age = age;

  }

  say(){

    console.log( "我叫"+this.name+"今年,"+this.age )

  }

}

var p1 = new Person( "web",27 );

注意：构造函数的原型~~不可以~~用Object.keys(Person.prototype)遍历*得到一个空数组*，但是理论上讲他上面应该有一个say方法

# class的静态方法和属性

构造函数（类）身上的方法叫静态方法

<script>

let str=new String('网');

console.log( str.charCodeAt(str) );//32593

console.log( String.fromCharCode(32593) );

</script>

## 静态方法

class Person{

constructor(){}//此处如果不定义实例的属性可以不写，会隐式创建

**static aa**(){

alert('静态方法')

}

bb(){

alert('实例对象上的方法');

}

**static cc(obj)**{

return obj instanceof Person;

}

}

Person.aa();

let b=new Person();

b.bb();

console.log(Person.cc(1));//false

console.log(Person.cc(b));//true;

## 静态属性

class Person{};//这里会隐式创建一个constructor方法

Person.props="abc";

console.log(Person.props);//abc

# class的继承

这里的super和ES5的call方法调用一样的意思，用来修改this指向；

# Promise

Promise:承诺（允诺），将来要执行的动作； **异步编程的一个解决方案**

## 比较同步和异步

同步阻塞模式，异步多任务同时进行，非阻塞模式；

同步代码写起来简单易懂，异步代码复杂（回调地狱）

**promise用同步写代码的模式来解决异步编程**

## resolve/reject/then

let p=new Promise(function(resolve,reject){

//异步程序的代码写这里

});

//参数是一个必填的函数，不填会报错；

//resolve:表示异步成功后的解决函数；

//reject：表示异步处理错误的拒绝函数;

p.then(function(res){},function(err){});

//这里的两个函数分别代表resolve和reject

## 简单使用

let p=new Promise(function(resolve,reject){

$.ajax({

url:'a.txt',

dataType:'json',

success(res){

resolve(res)

},

error(err){

reject(err);

}

})

});

p.then(function(data){

console.log(data);

}).catch(function(err){

console.log('失败',err)

});

建立一个a.txt文件，引入jQuery进来。

## 解决地狱回调-Promise.all().then()

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

$.ajax({

url:'a.txt',

dataType:'json',

success(res){

resolve(res)

},

error(err){

reject(err);

}

})

});

let p2=new Promise(function(resolve,reject){

$.ajax({

url:'b.txt',

dataType:'json',

success(res){

resolve(res)

},

error(err){

reject(err);

}

})

});

Promise.all([p1,p2]).then(function(res){

let [arr,json]=res;//拿回来的数据也是一个数组

console.log(arr);

console.log(json);

},function(err){

console.log(err)

});

**all表示全部成功返回数据，错误只要有一个错就停止；**

## 封装promise

function CreatePromise(url){

return new Promise(function(resolve,reject){

$.ajax({

url,

dataType:'json',

success(res){

resolve(res)

},

error(err){

reject(err);

}

})

});

}

Promise.all([

CreatePromise('a.txt'),

CreatePromise('b.txt')

]).then(function(res){

let [arr,json]=res;

console.log(arr);

console.log(json);

},function(err){

console.log(err)

});

## 利用jquery的ajax返回promise

Promise.all([

$.ajax({url:'a.txt',dataType:'json'}),

$.ajax({url:'b.txt',dataType:'json'})

]).then(function(res){

let [arr,json]= res;

console.log(arr);

console.log(json);

},function(err){

console.log('失败');

});

## 竞速Promise.race()

Promise.race([a,b,c,d,e]).then()

这里是如果有一个资源由多个负载点提供能拿回数据 ，那么谁的速度快，就先拿谁的数据过来使用。（这样其实也建立了多个http请求，影响性能）；

# 生成器函数generator

普通函数调用时会一次执行完毕，但是生成器函数碰到yield运算符会暂停住；

function \*show(){

alert('a');

yield;//放弃执行，等待你next()调用才会继续

alert('b');

}

let g=show();

console.log(g);

g.next();

调用生成器函数得用函数返回的生成器对象调next()方法才能一步一步执行；

本质上生成器函数，是将函数show分成了几个小段函数比如上面的是分成了

show\_1()和show\_2()

应用：可以解决我们在show函数中间请求异步数据等数据回来，我们再next()；

## 模拟数据请求

//普通函数

function fn(){

业务代码...

ajax(url,get,function(){

等数据回来执行业务代码

ajax(xxxx);

})

}

//生成器函数

function\* fn(){

业务代码...

yield ajax(xxxx);

yield ajax(xxxx);

yield ajax(xxxx);

}

## 理解yield

**利用yield传递参数**

function\* show(){

alert('a');

let n=yield;

alert('b');

alert(n);//3

}

let gen=show();

gen.next(1);

gen.next(3);

总结：第一次调用时是没法给yield传参的，要想第一次调用传参，可以通过gen=show(99)这样传参；

**利用yield返回数据**

function\* show(){

console.log('a');

yield 666;

console.log('b');

return 888;

}

let gen=show();

let res1=gen.next();

//let res2=gen.next();

let res2=gen.next();

console.log(res1);//{value: 666, done: false}

//console.log(res2);//{value: undefined, done: true}

console.log(res2);//{value: 888, done: true}

第一次调用next()方法拿到的结果是res1对象

第二次调用next()方法拿到的结果是res2对象，但是如果函数没有返回值，那么就是undefined,有返回值就是返回值

**传参和返回一起**

function\* show(){

console.log('a');

let s=yield 666;

console.log('b');

console.log(s);

return 888;

}

let gen=show();

let res1=gen.next();

let res2=gen.next(res1.value);

把上一次的结果给下一次用