**ECMAScript(ES5\_ES6\_ES7)**

# 第1章：**ECMAScript简介**

## 1.1 什么是ECMAScript

1. 它是一种由ECMA组织（前身为欧洲计算机制造商协会）制定和发布的脚本语言规范

2. 而我们学的JavaScript是ECMA的实现, 但术语ECMAScript和JavaScript平时表达同一个意思

3. JS包含三个部分：

1). ECMAScript（核心）

2). 浏览器端扩展

1). DOM（文档对象模型）

2). BOM（浏览器对象模型）

3). 服务器端扩展

1). Node

## 1.2 ES (ECMAScript)几个重要的版本

1) ES5 : 09年发布

2) ES6(ES2015) : 15年发布, 也称为ECMA2015

3) ES7(ES2016) : 16年发布, 也称为ECMA2016 (变化不大，大都是草案)

4) ES8(ES2017):17年6月底发布,目前都是草案

## 1.3学习地址

|  |
| --- |
| 1). ES5 :  http://www.zhangxinxu.com/wordpress/2012/01/introducing-ecmascript-5-1/  http://www.ibm.com/developerworks/cn/web/wa-ecma26  2). ES6  http://es6.ruanyifeng.com/  3). ES7  http://www.w3ctech.com/topic/1614 |

# 第2章：**ES5知识点**

## 2.1 严格模式

### 2.1.1 什么是严格模式？

1) 除了正常运行模式(混杂模式)，ES5添加了第二种运行模式："严格模式"（strict mode）。

2) 顾名思义，这种模式使得Javascript在更严格的语法条件下运行

### 2.1.2 严格模式的目的/作用？

1) 消除Javascript语法的一些不合理、不严谨之处，减少一些怪异行为

2) 消除代码运行的一些不安全之处，保证代码运行的安全

3) 为未来新版本的Javascript做好铺垫

### 2.1.3 如何使用严格模式？

1) 在全局或函数的第一条语句定义为: 'use strict';

2) 如果浏览器不支持, 只解析为一条简单的语句, 没有任何副作用

### 2.1.4 语法和行为改变

1) 必须用var声明变量

2) 创建eval作用域

3) 禁止this指向window

4) 对象不能有重名的属性

5) 函数不能有重名的形参

## 2.2 json对象

### 2.2.1 什么是json字符串

1. ‘abc’和‘{“username”: “kobe”}’ 这两个都是json字符串对吗？
2. Json字符串是一种数据格式，通常用来前后端进行数据传递

### 2.2.2 json字符串同原生JS对象/数组相互转换

1. JSON.stringify(obj/arr)

js对象(数组)转换为json对象(数组)

2. JSON.parse(json)

json对象(数组)转换为js对象(数组)

### 2.2.3 思考：json字符串中为什么只有json对象/json数组？

1) json对象/数组 作为数据进行传递后通常需要转换为原生JS对象/数组才能去使用

2) json对象/数组就是为了与原生的JS对象/数组相互转换的

原生的字符串无法转换为JS对象/数组，所以不能算为json字符串，毫无意义

## 2.3 Object扩展

### 2.3.1 Object.create(prototype, [descriptors])

1) 以指定对象为原型创建新的对象

2) 为新的对象指定新的属性, 并对属性进行描述

*- value : 指定值  
- writable : 标识当前属性值是否是可修改的, 默认为false  
- configurable: 标识当前属性是否可以被删除 默认为false  
- enumerable： 标识当前属性是否能用for in 枚举 默认为false*

### 2.3.2 Object.defineProperties(object, descriptors)

1) 为指定对象定义扩展多个属性

2) get: 用来获取当前属性值得回调函数

3) set: 修改当前属性值得触发的回调函数，并且实参即为修改后的值

4) 存取器属性：setter,getter一个用来存值，一个用来取值

### 2.3.3 对象本身的get和set方法

|  |
| --- |
| **var *obj*** = {  **firstName** : **'kobe'**,  **lastName** : **'bryant'**,  **get** fullName(){  **return this**.**firstName** + **' '** + **this**.**lastName** },  **set** fullName(data){  **var** names = data.split(**' '**);  **this**.**firstName** = names[0];  **this**.**lastName** = names[1];  } }; **console**.log(***obj***.fullName); ***obj***.fullName = **'curry stephen'**; **console**.log(***obj***.fullName); |

## 2.4 Array扩展

1. Array.prototype.indexOf(value) : 得到值在数组中的第一个下标  
2. Array.prototype.lastIndexOf(value) : 得到值在数组中的最后一个下标  
3. Array.prototype.forEach(function(item, index){}) : 遍历数组  
4. Array.prototype.map(function(item, index){}) : 遍历数组返回一个新的数组，返回加工之后的值  
5. Array.prototype.filter(function(item, index){}) : 遍历过滤出一个新的子数组， 返回条件为true的值

## 2.5 Function扩展

1. Function.prototype.bind(obj) :  
作用: 将函数内的this绑定为obj, 并将函数返回  
2. 面试题: 区别bind()与call()和apply()?  
 \* 都能指定函数中的this  
 \* call()/apply()是立即调用函数  
 \* bind()是将函数返回

\* call，bind传参方式一样，从第二个参数开始直接传入，apply传参需第二个参数是数组，所传参数放在数组里

# 第3章：**ES6知识点**

## 3.1 ES6必备

### 3.1.1 定义变量/常量

1) let定义变量

*1. 作用:  
 \* 与var类似, 用于声明一个变量  
2. 特点:  
 \* 在块作用域内有效  
 \* 不能重复声明  
 \* 不会预处理, 不存在提升  
3. 应用:  
 \* 循环遍历加监听  
 \* 使用let取代var是趋势*

2） const定义常量

*1、理解：什么是常量？ 常量通常被认为长时间保持一个状态，不应该被修改*

*2、在ES5中定义常量也是用var但没有真正意义上实现常量的意义，可以被修改*

*3、定义常量const，定义的常量不能被修改*

### 3.1.2 变量的解构赋值

1. 理解(解构赋值四字一分为二)：

- 解构：解析目标对象的结构

- 赋值：解析结构后获取数据

1. 对象的解构赋值

|  |
| --- |
| *let {n, a} = {n:'tom', a:12}* |

1. 数组的解构赋值

|  |
| --- |
| *let [a,b] = [1, 'atguigu'];* |

1. 用途

多用在给多个形参赋值

### 3.1.3 模板字符串

1) 作用：简化字符串的拼接

2) 模板字符串必须用 `` 包含

3) 变化的部分使用${xxx}定义

### 3.1.4 简化的对象写法

1) 省略同名的属性值

2) 省略方法的function

|  |
| --- |
| let ***x*** = 3;  let ***y*** = 5;  *//普通额写法 // let obj = { // x : x, // y : y, // getPoint : function () { // return this.x + this.y // } // };  //简化的写法* let ***obj*** = {  ***x***,  ***y***,  getPoint(){  **return this**.**x** }  };  **console**.log(***obj***, ***obj***.getPoint()); |

### 3.1.5 箭头函数

1. 作用：定义匿名函数

2. 场景：多用来定义回调函数

3. 特点：

\* 语法简洁

\* 没有自己的this，定义时候所处的对象就是它的this

\* 扩展理解：看外层有没有函数，如果有则当前箭头函数的this同外层函数的this一样，如果没有则this指向window

4.语法：

*\* 没有参数: () => console.log('xxxx')  
\* 一个参数: i => i+2  
\* 大于一个参数: (i,j) => i+j  
\* 函数体不用大括号: 默认返回结果  
\* 函数体如果有多个语句, 需要用{}包围，若有需要返回的内容，需要手动返回*

### 3.1.6 点点点运算符

1) rest(可变)参数,用来取代arguments 但比arguments灵活,只能是最后部分形参参数

|  |
| --- |
| *function add(...values) {  let sum = 0;  for(value of values) {  sum += value;  }  return sum; }* |

2) 扩展运算符

let arr1 = [1,3,5];  
let arr2 = [2,...arr1,6];  
arr2.push(...arr1);

### 3.1.7 形参默认值

*\* 当不传入参数的时候默认使用形参里的默认值*

|  |
| --- |
| *function Point(x = 1,y = 2) { this.x = x; this.y = y; }* |

### 3.1.8 Promise对象

1. 理解Promise对象

\* Promise对象: 代表了未来某个将要发生的事件(通常是一个异步操作)  
\* 有了promise对象, 可以将异步操作以同步的流程表达出来, 避免了层层嵌套的回调函数(俗称'回调地狱')  
\* ES6的Promise是一个构造函数, 用来生成promise实例

1. 使用Promise对象

|  |
| --- |
| *\* 创建promise对象  let promise = new Promise((resolve, reject) => {  //初始化promise状态为 pending  //执行异步操作  if(异步操作成功) {  resolve(value);//修改promise的状态fullfilled  } else {  reject(errMsg);//修改promise的状态为rejected  }  }) \* 调用promise的then()  promise.then(function(  result => console.log(result),  errorMsg => alert(errorMsg)  ))* |

1. Promise对象的三个状态

*\* pending: 初始化状态  
\* fullfilled: 成功状态  
\* rejected: 失败状态*

1. 应用

\* 使用Promise对象实现超时处理

|  |
| --- |
| **function** *getNews*(url) {  *//创建一个promise对象* let promise = **new** Promise((resolve, reject) => {  *//初始化promise状态为pending  //启动异步任务* let request = **new** XMLHttpRequest();  request.**onreadystatechange** = **function** () {  **if**(request.**readyState** === 4){  **if**(request.**status** === 200){  let news = request.**response**;  resolve(news);  }**else**{  reject(**'请求失败了。。。'**);  }  }  };  request.**responseType** = **'json'**;*//设置返回的数据类型* request.open(**"GET"**, url);*//规定请求的方法，创建链接* request.send();*//发送* })  **return** promise; }  *getNews*(**'http://localhost:3000/news?id=2'**)  .then((news) => {  **console**.log(news);  **document**.write(**JSON**.stringify(news));  **console**.log(**'http://localhost:3000'** + news.commentsUrl);  **return** *getNews*(**'http://localhost:3000'** + news.commentsUrl);  }, (error) => {  alert(error);  })  .then((comments) => {  **console**.log(comments);  **document**.write(**'<br><br><br><br><br>'** + **JSON**.stringify(comments));  }, (error) => {  alert(error);  }) |

### 3.1.9 Symbol属性

*前言：ES5中对象的属性名都是字符串，容易造成重名，污染环境*

1. 什么是Symbol属性

*ES6中的添加了一种原始数据类型symbol(已有的原始数据类型：String, Number, boolean, null, undefined, 对象)*

1. Symbol特点

*1、Symbol属性对应的值是唯一的，解决命名冲突问题  
2、Symbol值不能与其他数据进行计算，包括同字符串拼串  
3、for in, for of遍历时不会遍历symbol属性。*

3) Symbol应用

1. 调用Symbol函数得到symbol值

|  |
| --- |
| *let symbol = Symbol(); let obj = {}; obj[symbol] = 'hello';* |

1. 传参标识

|  |
| --- |
| *let symbol = Symbol('one'); let symbol2 = Symbol('two'); console.log(symbol);// Symbol('one') console.log(symbol2);// Symbol('two')* |

1. 内置Symbol值

*\* 除了定义自己使用的Symbol值以外，ES6还提供了11个内置的Symbol值，指向语言内部使用的方法。*

*\* Symbol.iterator，对象的Symbol.iterator属性，指向该对象的默认遍历器方法(后边讲)*

### 3.1.10 iterator遍历器

#### 1. 什么是iterator？

*iterator是一种接口机制，为各种不同的数据结构提供统一的访问机制*

#### Iterator接口作用

*1、为各种数据结构，提供一个统一的、简便的访问接口；  
2、使得数据结构的成员能够按某种次序排列  
3、ES6创造了一种新的遍历命令for...of循环，Iterator接口主要供for...of消费。*

#### Iterator接口工作原理

*1. 创建一个指针对象，指向数据结构的起始位置。  
2. 第一次调用next方法，指针自动指向数据结构的第一个成员  
3. 接下来不断调用next方法，指针会一直往后移动，直到指向最后一个成员  
4.每调用next方法返回的是一个包含value和done的对象，{value: 当前成员的值,done: 布尔值}  
 4.1 value表示当前成员的值，done对应的布尔值表示当前的数据的结构是否遍历结束。  
 4.2 当遍历结束的时候返回的value值是undefined，done值为false*

#### 原生具备iterator接口的数据类型

*1、Array  
2、arguments  
3、set容器  
4、map容器  
5、String*

#### ES6方法和iterator接口的关系

1. *使用解构赋值以及...三点运算符时会调用iterator接口*
2. *Generator函数的yeild语句会调用iterator接口(后边讲)*

#### 自定义iterator接口(遍历器对象)

|  |
| --- |
| **function** *mockIterator*(arr) {  let nextIndex = 0; // 记录访问数据结构的位置  **return** {  next: **function** () {// 遍历器对象  **return** nextIndex<arr.**length**?{**value**: arr[nextIndex++], **done**: **false**}:{**value**: **undefined**, **done**: **true**} // 动态返回遍历的结果数据  }  } }  let arr = [1,2,3,4,5]; let iteratorObj = *mockIterator*(arr); **console**.log(iteratorObj.next()); **console**.log(iteratorObj.next()); **console**.log(iteratorObj.next()); |

### 3.1.11 Generator函数

#### 1、 什么是Generator函数？

*1、ES6提供的解决异步编程的方案之一  
2、Generator函数是一个状态机，内部封装了不同状态的数据，  
3、用来生成遍历器对象(iterator接口)  
4、可暂停函数(惰性求值), yield可暂停，next方法可启动。每次返回的是yield后的表达式结果*

#### Generator函数特点

*1、function 与函数名之间有一个星号  
2、内部用yield表达式来定义不同的状态  
例如：  
 function\* generatorExample(){  
 let result = yield 'hello'; // 状态值为hello  
 yield 'generator'; // 状态值为generator  
 }  
3、generator函数返回的是指针对象(接3.1.10章节里iterator)，而不会执行函数内部逻辑  
4、调用next方法函数内部逻辑开始执行，遇到yield表达式停止，返回{value: yield后的表达式结果/undefined, done: false/true}  
5、再次调用next方法会从上一次停止时的yield处开始，直到最后  
6、yield语句返回结果通常为undefined， 当调用next方法时传参内容会作为启动时yield语句的返回值。*

|  |
| --- |
| **function**\* *generatorTest*() {  **console**.log(**'函数开始执行'**);  yield **'hello'**;  **console**.log(**'函数暂停后再次启动'**);  yield **'generator'**; } *// 生成遍历器对象* let ***Gt*** = *generatorTest*(); *// 执行函数，遇到yield后即暂停* **console**.log(***Gt***); *// 遍历器对象* let ***result*** = ***Gt***.next(); *// 函数执行,遇到yield暂停* **console**.log(***result***); *// {value: "hello", done: false}* ***result*** = ***Gt***.next(); *// 函数再次启动* **console**.log(***result***); *// {value: 'generator', done: false}* ***result*** = ***Gt***.next(); **console**.log(***result***); *// {value: undefined, done: true}表示函数内部状态已经遍历完毕* |

#### 3、人为给对象添加iterator接口(可用for of遍历)

|  |
| --- |
| let ***myIterable*** = {}; ***myIterable***[Symbol.**iterator**] = **function**\* () {  yield 1;  yield 2;  yield 4; }; **for**(let ***i* of *myIterable***){  **console**.log(***i***); } let ***obj*** = [...***myIterable***]; **console**.log(***obj***); |

#### 4、案例练习

|  |
| --- |
| *\* 需求： \* 1、发送ajax请求获取新闻内容 \* 2、新闻内容获取成功后再次发送请求，获取对应的新闻评论内容 \* 3、新闻内容获取失败则不需要再次发送请求。*  **function**\* *sendXml*() {  *// url为next传参进来的数据* let url = yield *getNews*(**'http://localhost:3000/news?newsId=2'**);  yield *getNews*(url); } **function** *getNews*(url) {  $.get(url, **function** (data) {  **console**.log(data);  let commentsUrl = data.commentsUrl;  let url = **'http://localhost:3000'** + commentsUrl;  *// 当获取新闻内容成功，发送请求获取对应的评论内容  // 调用next传参会作为上次暂停是yield的返回值* ***sx***.next(url);  }) }   let ***sx*** = *sendXml*(); *// 发送请求获取新闻内容* ***sx***.next(); |

### 3.1.12 async函数

贴心小提示：async函数来自ES7

#### 什么是async函数

1. Generator函数的语法糖
2. 真正意义上去解决异步回调的问题，同步流程表达异步操作

#### async语法

|  |
| --- |
| *async function foo(){  await 异步操作;  await 异步操作； }* |

#### async函数特点

1、不需要像Generator去调用next方法，遇到await等待，当前的异步操作完成就往下执行  
2、返回的总是Promise对象，可以用then方法进行下一步操作  
3、async取代Generator函数的星号\*，await取代Generator的yield  
4、语意上更为明确，使用简单，经临床验证，暂时没有任何副作用

#### 4、案例练习

|  |
| --- |
| async **function** *sendXml*(url) {  **return new** Promise((resolve, reject) => {  $.ajax({  url,  **type**: **'GET'**,  success: data => resolve(data),  error: error => reject(error)  })  }) }  **async function** *getNews*(url) {  let result = **await** *sendXml*(url);  let result2 = **await** *sendXml*(url);  **console**.log(result, result2); } *getNews*(**'http://localhost:3000/news?id=2'**) |

### 3.1.13 class类

*1. 通过class定义类/实现类的继承  
2. 在类中通过constructor定义构造方法  
3. 通过new来创建类的实例  
4. 通过extends来实现类的继承  
5. 通过super调用父类的构造方法  
6. 重写从父类中继承的一般方法*

|  |
| --- |
| **class** Person {  *//调用类的构造方法* constructor(name, age){  **this**.**name** = name;  **this**.**age** = age;   }  *//定义一般的方法* showName(){  **console**.log(**this**.**name**, **this**.**age**);  } } let ***person*** = **new** Person(**'kobe'**, 39); **console**.log(***person***, ***person***.showName());  *//定义一个子类* **class** StrPerson **extends** Person{  constructor(name, age, salary){  **super**(name, age);*//调用父类的构造方法* **this**.**salary** = salary;  }  showName(){*//在子类自身定义方法* **console**.log(**this**.**name**, **this**.**age**, **this**.**salary**);  } } let ***str*** = **new** StrPerson(**'weide'**, 38, 1000000000); **console**.log(***str***); ***str***.showName(); |

### 3.1.14 Module模块化

在JS模块化课程里单独讲解

## 3.2 ES6其他语法

### 3.2.1 字符串扩展

*1. includes(str) : 判断是否包含指定的字符串  
2. startsWith(str) : 判断是否以指定字符串开头  
3. endsWith(str) : 判断是否以指定字符串结尾  
4. repeat(count) : 重复指定次数*

|  |
| --- |
| *//startsWith(str) : 判断是否以指定字符串开头* **console**.log(***str***.startsWith(**'a'**));*//true* **console**.log(***str***.startsWith(**'d'**));*//false //endsWith(str) : 判断是否以指定字符串结尾* **console**.log(***str***.endsWith(**'g'**));*//true* **console**.log(***str***.endsWith(**'d'**));*//false //repeat(count) : 重复指定次数a* **console**.log(***str***.repeat(5)); |

### 3.2.2 数值扩展

*1. 二进制与八进制数值表示法: 二进制用0b, 八进制用0o  
2. Number.isFinite(i) : 判断是否是有限大的数  
3. Number.isNaN(i) : 判断是否是NaN  
4. Number.isInteger(i) : 判断是否是整数  
5. Number.parseInt(str) : 将字符串转换为对应的数值  
6. Math.trunc(i) : 直接去除小数部分*

|  |
| --- |
| *//Number.isFinite(i) : 判断是否是有限大的数* **console**.log(Number.isFinite(**NaN**));*//false* **console**.log(Number.isFinite(5));*//true //Number.isNaN(i) : 判断是否是NaN* **console**.log(Number.isNaN(**NaN**));*//true* **console**.log(Number.isNaN(5));*//falsse  //Number.isInteger(i) : 判断是否是整数* **console**.log(Number.isInteger(5.23));*//false* **console**.log(Number.isInteger(5.0));*//true* **console**.log(Number.isInteger(5));*//true  //Number.parseInt(str) : 将字符串转换为对应的数值* **console**.log(Number.parseInt(**'123abc'**));*//123* **console**.log(Number.parseInt(**'a123abc'**));*//NaN  // Math.trunc(i) : 直接去除小数部分* **console**.log(**Math**.trunc(13.123));*//13* |

### 3.2.3 数组扩展

*1. Array.from(v)：将伪数组对象或可遍历对象转换为真数组  
2. Array.of(v1, v2, v3) : 将一系列值转换成数组  
3. find(function(value, index, arr){return true}) : 找出第一个满足条件返回true的元素  
4. findIndex(function(value, index, arr){return true}) : 找出第一个满足条件返回true的元素下标*

|  |
| --- |
| let ***btns*** = **document**.getElementsByTagName(**'button'**); **console**.log(***btns***.**length**);*//3* Array.from(***btns***).forEach(**function** (item, index) {  **console**.log(item, index); }); *//Array.of(v1, v2, v3) : 将一系列值转换成数组* let ***arr*** = Array.of(1, **'abc'**, **true**); **console**.log(***arr***); |

### 3.2.4 对象扩展

*1. Object.is(v1, v2)  
 \* 判断2个数据是否完全相等  
2. Object.assign(target, source1, source2..)  
 \* 将源对象的属性复制到目标对象上  
3. 直接操作 \_\_proto\_\_ 属性  
 let obj2 = {};  
 obj2.\_\_proto\_\_ = obj1;*

|  |
| --- |
| *//Object.assign(target, source1, source2..)* let ***obj*** = {**name** : **'kobe'**, **age** : 39, **c**: {**d**: 2}}; let ***obj1*** = {}; Object.assign(***obj1***, ***obj***); **console**.log(***obj1***, ***obj1***.**name**);  *//直接操作 \_\_proto\_\_ 属性* let ***obj3*** = {**name** : **'anverson'**, **age** : 41}; let ***obj4*** = {}; ***obj4***.\_\_proto\_\_ = ***obj3***; **console**.log(***obj4***, ***obj4***.**name**, ***obj4***.**age**); |

### 3.2.5 深度克隆

#### 1. 对象类型：

*1、数据分为基本的数据类型(String, Number, boolean, Null, Undefined)和对象数据类型  
2、基本数据类型：  
 特点： 存储的是该对象的实际数据  
3、对象数据类型：  
 特点：存储的是该对象在栈中引用，真实的数据存放在堆内存里*

#### 2. 复制数据

##### 2.1基本数据类型存放的就是实际的数据，可直接复制

*let number2 = 2;  
let number1 = number2;*

##### 2.2 对象/数组

*区别： 浅拷贝/深度拷贝  
判断： 拷贝是否产生了新的数据还是拷贝的是数据的引用  
知识点：对象数据存放的是对象在栈内存的引用，直接复制的是对象的引用  
 let obj = {username: 'kobe'}  
 let obj1 = obj; // obj1 复制了obj在栈内存的引用*

#### 3. 常用拷贝技术

*1). arr.concat(): 数组浅拷贝  
2). arr.slice(): 数组浅拷贝  
3). JSON.parse(JSON.stringify(arr/obj)): 数组或对象深拷贝, 但不能处理函数数据*

*4). Object.assign(target, source)*

#### 4. 实现深度克隆

|  |
| --- |
| *// 深度克隆(复制)* **function** *getObjClass*(obj) {  let result = Object.**prototype**.toString.call(obj).slice(8, -1);  **if**(result === **'Null'**){  **return 'Null'**;  }**else if**(result === **'Undefined'**){  **return 'Undefined'**;  }**else** {  **return** result;  } } *// for in 遍历数组的时候遍历的是下标* let ***testArr*** = [1,2,3,4]; **for**(let ***i*** in ***testArr***){  **console**.log(***i***); *// 对应的下标索引* }  *// 深度克隆* **function** *deepClone*(obj) {  let result, objClass = *getObjClass*(obj);  **if**(objClass === **'Object'**){  result = {};  }**else if**(objClass === **'Array'**){  result = [];  }**else** {  **return** obj; *// 如果是其他数据类型不复制，直接将数据返回* }  *// 遍历目标对象* **for**(let key in obj){  let value = obj[key];  **if**(*getObjClass*(value) === **"Object"** || **'Array'**){  result[key] = *deepClone*(value);  }**else** {  result[key] = obj[key];  }  }  **return** result; }   let ***obj3*** = {**username**: **'kobe'**,**age**: 39, **sex**: {**option1**: **'男'**, **option2**: **'女'**}}; let ***obj4*** = *deepClone*(***obj3***); **console**.log(***obj4***); ***obj4***.**sex**.**option1** = **'不男不女'**; *// 修改复制后的对象不会影响原对象* **console**.log(***obj4***, ***obj3***); |

### 3.2.6 Set和Map容器

*1. Set容器 : 无序不可重复的多个value的集合体  
 \* Set()  
 \* Set(array)  
 \* add(value)  
 \* delete(value)  
 \* has(value)  
 \* clear()  
 \* size  
2. Map容器 : 无序的 key不重复的多个key-value的集合体  
 \* Map()  
 \* Map(array)  
 \* set(key, value)//添加  
 \* get(key)  
 \* delete(key)  
 \* has(key)  
 \* clear()  
 \* size*

|  |
| --- |
| let ***set*** = **new** Set([1,2,3,4,3,2,1,6]); **console**.log(***set***); ***set***.add(**'abc'**); **console**.log(***set***, ***set***.**size**); *//delete(value)* ***set***.delete(2); **console**.log(***set***); *//has(value)* **console**.log(***set***.has(2));*//false* **console**.log(***set***.has(1));*//true //clear()* ***set***.clear(); **console**.log(***set***);  let ***map*** = **new** Map([[**'abc'**, 12],[25, **'age'**]]); **console**.log(***map***); ***map***.set(**'男'**, **'性别'**); **console**.log(***map***); **console**.log(***map***.get(25));*//age //delete(key)* ***map***.delete(**'男'**); **console**.log(***map***); **console**.log(***map***.has(**'男'**));*//false* **console**.log(***map***.has(**'abc'**));*//true* ***map***.clear(); **console**.log(***map***); |

### 3.2.7 for of循环

*1. 遍历数组  
2. 遍历Set  
3. 遍历Map  
4. 遍历字符串  
5. 遍历伪数组*

|  |
| --- |
| let ***arr*** = [1,2,3,4,5]; **for**(let ***num* of *arr***){  **console**.log(***num***); } let ***set*** = **new** Set([1,2,3,4,5]); **for**(let ***num* of *set***){  **console**.log(***num***); } let ***str*** = **'abcdefg'**; **for**(let ***num* of *str***){  **console**.log(***num***); } let ***btns*** = **document**.getElementsByTagName(**'button'**); **for**(let ***btn* of *btns***){  **console**.log(***btn***.**innerHTML**); } |

# 第4章：**ES7知识点**

贴心小提示 ES7大部分语法也还是草案，只有少部分正式发布使用

*1. 指数运算符(幂): \*\*  
2. Array.prototype.includes(value) : 判断数组中是否包含指定value*

|  |
| --- |
| **console**.log(3 \*\* 3);*//27* let ***arr*** = [1,2,3,4, **'abc'**]; **console**.log(***arr***.includes(2));*//true* **console**.log(***arr***.includes(5));*//false* |