###### ES6怎么来的

● ECMAScript 和 JavaScript

ECMA 是标准，JS 是实现

CMAScript 简称 ECMA 或 ES

● 历史版本

1996, ES1.0 Netscape 将 JS 提交给 ECMA 组织，ES 正式出现

1999, ES3.0 被广泛支持

2011, ES5.1 成为 ISO 国际标准

2015, ES6.0 正式发布

###### ES6兼容性

● ES6(ES2015) 支持的环境 IE10+, Chrome, FireFox, 移动端, NodeJS

● 解决不兼容办法，编译、转换

在线转换

或者提前编译

● [Babel 中文网](https://www.babeljs.cn/)

[Babel 入门教程 阮一峰](http://www.ruanyifeng.com/blog/2016/01/babel.html)

Babel 是一个 JavaScript 编译器

一个广泛使用的转码器，可以将ES6代码转为ES5代码，从而在现有环境执行

现在就用 ES6 编写程序，而不用担心现有环境是否支持

###### 变量 let 和 常量 const

var 的问题

可以重复声明，没有报错和警告

无法限制修改

函数级 没有块级作用域， { }

说明：用var声明的变量，会在所有代码执行前辈声明（不赋值）

let 不能重复声明 块级作用域 变量 可以修改

Const 不能重复声明 块级作用域 常量 不能修改

块级作用域, { } 块内声明的，块外无效

块级作用域举例

原来用 var 的方式，结果弹出的都是 3

或者将变量 封装到函数里，限制作用域，但比较麻烦

用 let 最简单，直接 var 改 let，解决作用域问题



###### 4.函数-箭头函数（定义匿名函数且是回调函数）

* 箭头函数，就是函数的简写
  + 如果只有一个参数，() 可以省
  + 如果只有一个return，{}可以省

// 普通函数function name() {

}

// 箭头函数，去掉 function， 加上 =>

() => {

}

let show1 = function () {

console.log('abc')

}

let show2 = () => {

console.log('abc')

}

show1() // 调用函数show2()

let show4 = function (a) {

return a\*2

}

let show5 = a => a \* 2 //简洁，类似python lambda 函数

console.log(show4(10))

console.log(show5(10))

**箭头函数还会继承当前上下文的 this 关键字**

this 先用函数内部没有就一层层往外找

1. 箭头函数的this绑定看的是this所在的函数定义在哪个对象下，绑定到哪个对象则this就指向哪个对象（）

说明： 取决于其在定义时的外部函数：

①在全局环境下定义 this >> window

②在A函数下定义 this >> A (若A为Vue实例，this >> vue)

2.如果有对象嵌套的情况，则this绑定到最近的一层对象上

箭头函数不会被 all 、bind、applay 等改变 this

**不能使用箭头函数的几种情况：**

**①作为对象的属性**：

###### var obj = {

###### a: () => {

###### console.log(this) //window

###### }

###### }

②构造函数

③作为原型方法

④需要动态获取this的时候

###### 5.函数-参数

● 参数扩展／数组展开  ...args

收集剩余的参数，必须在最后一个参数位置

展开数组，简写，效果和直接把数组的内容写在这儿一样

● 默认参数

function show(a, b, ...args) {

console.log(a)

console.log(b)

console.log(args)

}

console.log(show(1, 2, 3, 4, 5)) //收集剩余的参数

let arr1 = [1, 2, 3]

let arr2 = [4, 5, 6]

let arr3 = [...arr1, ...arr2] //展开数组（组装数组）

console.log(arr3) //默认参数

function show2(a, b=5, c=8) {

console.log(a, b, c)

}

show2(88, 12)

获取数组的部分项。

const arr = ['a', 'b', 'c'];

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

rest; // ['b', 'c']

// With ignoreconst

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

rest; // ['c']

代替 apply。

function foo(x, y, z) {}

const args = [1,2,3];

// 下面两句效果相同

foo.apply(null, args);

foo(...args);

对于 Object 而言，用于组合成新的 Object 。(ES2017 stage-2 proposal)

const foo = {

a: 1,

b: 2,

};

const bar = {

b: 3,

c: 2,

};

const d = 4;

const ret = { ...foo, ...bar, d }; // { a:1, b:3, c:2, d:4 }

###### 解构赋值

解构赋值

* 左右两个边结构必须一样
* 右边必须是个东西
* 声明和赋值赋值不能分开，必须在一句话里

let [a, b, c] = [1, 2, 3]

console.log(a, b, c)

let [json, arr, num, str] = [{ a: 1, b: 2 }, [1, 2, 3], 8, 'str']

console.log(json, arr, num, str)

对象结构赋值-重命名 (左边是对象字面量)

let {x:a, y:b, z:c} = {x: 1, y: 2, z: 3}

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

当对象的key-value相同时

let {x, y, z} = {x: 1, y: 2, z: 3} //**let x= {x: 1, y: 2, z: 3} .x**

console.log(x, y, z) //1 2 3

const {todo, index, deleteTodo} = this todo=this.todo

const { name, age } = people name=people.name

**解构有趣的用途**

1. 变量交换

通常交换两个变量的方法需要一个额外的临时变量：

let a = 1; let b = 2;

let temp; //中间变量

temp = a;

a = b;

b = temp;

a; // => 2 b; // => 1

解构赋值：

let a = 1;

let b = 2;

[a, b] = [b, a]; //右边，创建了一个数组[b, a]，即[2,1]

a; // => 2

b; // => 1

数组

const arr = [1, 2];

const [foo, bar] = arr;

console.log(foo); // 1

2. 访问数组中元素

const colors = [];

const [ ,firstColor = 'white'] = colors;

firstColor; // => 'white'

解构将colors数组的第二个元素赋给firstColor变量。如果数组在索引1处没有任何元素，则分配“white”默认值。

**对象字面量改进**

这是析构的反向操作，用于重新组织一个 Object 。

const name = 'duoduo';

const age = 8;

const user = { name, age }; // { name: 'duoduo', age: 8 }

定义对象方法时，还可以省去 :function 关键字。

app.model({

reducers: {

add() {} // 等同于 add: function() {}

},

effects: {

\*addRemote() {} // 等同于 addRemote: function\*() {}

},});

###### 7.数组 callback的三个参数（item，index，array）

新增4个方法(map、reduce、filter、forEach)

**[Array.from(xx)](https://www.cnblogs.com/kongxianghai/p/7417210.html)** [类数组 > 数组](https://www.cnblogs.com/kongxianghai/p/7417210.html) (有length属性就能用,字符串和array都有)

**Array.of(xx)** 一组值 > 数组

说明：扩展运算符(...)也可以将某些数据结构转为数组

let arr = Array.from('juejin');

console.log(arr); //["j", "u", "e", "j", "i", "n"]

function doSomething (){

return [...arguments]

}

doSomething('a','b','c'); // ["a","b","c"]

**JS遍历方法总结**

**for循环**

适用范围：Array、String  
点评：性能最高但不够优雅

**Array.forEach(callback)**  遍历数组中的每一项

没有返回值，对原数组没有影响（可以用数组方法改变数组，）

适用范围：Array、Set、Map  
点评：无法中途跳出循环，break命令不能奏效,只能用return退出本次回调，进行下一次回调。

var arr = [12, 4, 8, 9]

var result = arr.forEach(item => console.log(item))

var result = arr.forEach((item, index)=>console.log(item, index))

let arr=[1,2,3,4,5,6,7,8]

arr.forEach(item => {

if(item%2===0){

return // return退出本次回调

}

console.log(item); // 1 3 5 7

});

**arr.map(****callback)** 映射 一个对一个

有返回值，返回新的数组，对原数组没有影响（原数组克隆一份来遍历）

适用于要对源数据进行加工或修改的业务场景

let arr = [12, 5, 8]

let result = arr.map(function (item) {

return item\*2

})

let result2 = arr.map(item=>item\*2) // 简写

console.log(result)

console.log(result2)

let score = [18, 86, 88, 24]

let result3 = score.map(item => item >= 60 ? '及格' : '不及格')console.log(result3)

// 结果

[ 24, 10, 16 ]

[ 24, 10, 16 ]

[ '不及格', '及格', '及格', '不及格' ]

**for...in**

适用范围：Array、**Object**、String  
点评：会枚举原型属性，就是说会遍历来自继承对象的可枚举属性(数组别用)

const data = { key1: "a", key2: "b", key3: "c" };

for (let key in data) {

console.log(key); // key1 key2 key3

console.log(data[key]); // a b c

}

const data = ["a", "b", "c", "d"];

for (let index in data) {

console.log(index); // 0 1 2 3

}

**for...of**

适用范围：Array、String、Set、Map  
点评：ES6新增的作为遍历所有数据结构的统一的方法。唯一的缺点也许就是兼容性了吧。

**const** data = ["a", "b", "c", "d"];

**for** (**let** item **of** data) {

console.log(item); // a b c d

}

**Array.filter(callback)** 过滤器 保留为true的项 有返回值，返回一个新数组

var arr = [12, 4, 8, 9]

var result = arr.filter(item => (item % 3 === 0) ? true :false)

console.log(result)

var result = arr.filter(item => item % 3 === 0)

console.log(result)

var arr = [{ title: '苹果', price: 10 },

{ title: '西瓜', price: 20 }, ]

var result = arr.filter(json => json.price >= 20)

console.log(result)

适用于想从源数据中筛选/提取出指定数据的业务场景。

**Array.reduce(callback)** 汇总 一堆出来一个 (合并为一个值)

用于比如，算个总数，算个平均

var arr = [1, 3, 5, 7]

var result = arr.reduce(function (tmp, item, index) {

//tmp 上次结果，item当前数，index次数1开始

console.log(tmp, item, index)

return tmp + item })

console.log(result)

var arr = [1, 3, 5, 7]

var result = arr.reduce(function (tmp, item, index) {

if (index != arr.length - 1) { // 不是最后一次

return tmp + item

} else { return (tmp + item)/arr.length

}

Var result=arr.reduce((a,b,c)=>{c!=arr.length-1?(a+b):(a+b)/arr.length})

})console.log(result) // 平均值

**Array.find****(****callback)** 找到第一个符合条件的数组成员**返回,**没有则返回undefined

(return true 就返回当前item， return false 则查找下一个)

**Array.****some(callback)** 检测任一元素是否满足条件，并返回一个Boolean值。

 如果有一个元素满足条件，则表达式返回true , 剩余的元素不会再执行检测。如果没有满足条件的元素，则返回false。(任一返回true 则返回true)

**Array.every(callback)** 检测所有元素是否满足条件，并返回一个Boolean值。 如果数组中检测到有一个元素不满足，则整个表达式返回 false ，且剩余的元素不会再进行检测。如果所有元素都满足条件，则返回 true。(全部返回true，则返回true)

扩展：

数组的3个方法，均返回 Array的迭代器对象

Array.keys() //遍历数组键名

Array.values() //遍历数组键值

Array.entries() //遍历数组键名和键值

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

let obj=arr.entries()

console.log(obj) //Array Iterator {} 不能正常打印

console.log([...obj]); //能打印出对象数组

说明 迭代器对象可以被 for...of 遍历 扩展运算符(...)也可以使用

数组方法entries()可以用于访问迭代项的索引。

该方法在每次迭代时返回一对 [index，item]。

就地解构是 for...of 的另一个重要功能

对象的方法：

返回遍历后的数组

Object.keys(xx)

Object.values(xx)

###### 8.字符串

多了两个新方法

startsWith（） 以xx开始

endsWith（） 以xx结束

var url = 'http://qq.com'

console.log(url.startsWith('http'))

console.log(url.endsWith('com'))// 都是 true

字符串模版 字符串连接

` `使用反引号，${变量}

可以折行

let a = 12

let str1 = `asdf${a}`

console.log(str1)

let title = '标题'

let content = '内容'

let str = `<div>

<h1>${title}</h1>

<p>${content}</p>`

console.log(str)

<div>

<h1>标题</h1>

<p>内容</p>

###### 9.面向对象-基础

原来写法

类和构造函数一样

属性和方法分开写的

// 老版本function User(name, pass) {

this.name = name;

this.pass = pass;

}

User.prototype.showName = function () {

console.log(this.name);

};

User.prototype.showPass = function () {

console.log(this.pass);

};

var u1 = new User('able', '1233');

u1.showName();

u1.showPass();// 老版本继承

function VipUser(name, pass, level) {

User.call(this, name, pass);

this.level = level;}

VipUser.prototype = new User()

VipUser.prototype.constructor = VipUser;

VipUser.prototype.showLevel = function () {

console.log(this.level);

};

var v1 = new VipUser('blue', '1234', 3)

v1.showName();

v1.showLevel();

新版面向对象

有了 class 关键字、构造器constructor

class 里面直接加方法

继承extends，super 超类==父类(调用父类的属性或方法)

class User {

constructor(name, pass) {

this.name = name;

this.pass = pass;

}

showName() {

console.log(this.name);

}

showPass() {

console.log(this.pass);

}

}

var u1 = new User('able2', '111')

u1.showName()

u1.showPass()// 新版本继承

class VipUser extends User {

constructor(name, pass, level) {

super(name, pass) //调用父类的 constructor(name,pass)

this.level = level

}

showLevel(){

console.log(this.level)

}

}

v1 = new VipUser('blue', '123', 3)

v1.showLevel()

###### 10.面向对象应用

[React](https://www.reactjscn.com/)

用于构建用户界面的 JavaScript 库

组件化，一个组件就是一个 class

JSX == bable == browser.js

###### 11.json

JSON 格式

JavaScript Object Notation 的缩写，是一种用于数据交换的文本格式

JSON 是 JS对象 的严格子集

JSON 的标准写法

只能用双引号

所有的key都必须用双引号包起来

JSON 对象

JSON 对象是 JavaScript 的原生对象，用来处理 JSON 格式数据，有两个静态方法

JSON.parse(string) ：接受一个 JSON 字符串并将其转换成一个 JavaScript 对象。

JSON.stringify(obj) ：接受一个 JavaScript 对象并将其转换为一个 JSON 字符串。

var json = {a: 12, b: 5};

var str = 'hi,' + JSON.stringify(json)

var url = 'http://www.xx.com/' + encodeURIComponent(JSON.stringify(json))

console.log(str)

console.log(url)

var str = '{"a": 12, "b": 4, "c": "abc"}' //json对象

var json = JSON.parse(str)

console.log(json)

hi,{"a":12,"b":5}

http://www.xx.com/%7B%22a%22%3A12%2C%22b%22%3A5%7D

{ a: 12, b: 4, c: 'abc' }

对象（object）

是 JavaScript 语言的核心概念，也是最重要的数据类型

对象就是一组“键值对”（key-value）的集合，是一种无序的复合数据集合

对象的所有键名都是字符串, 所以加不加引号都可以

如果键名是数值，会被自动转为字符串

对象的每一个键名又称为“属性”（property），它的“键值”可以是任何数据类型

如果一个属性的值为函数，通常把这个属性称为“方法”，它可以像函数那样调用

in 运算符用于检查对象是否包含某个属性（注意，检查的是键名，不是键值

for...in循环用来遍历一个对象的全部属性

对象 简写

key-value 一样时可以简写

方法 ：function 一块删

里面函数可以简写, 去掉 show :functon(){} show (){}

var a = 12, b = 5

console.log({a:a, b:b})

console.log({a, b})

console.log({a, b, c:"c"})

console.log({ a, b, show(){ console.log('a') }})

{ a: 12, b: 5 }

{ a: 12, b: 5 }

{ a: 12, b: 5, c: 'c' }

{ a: 12, b: 5, show: [Function: show] }

###### 12.Promise

异步和同步

异步，操作之间没有关系，同时执行多个操作， 代码复杂

同步，同时只能做一件事，代码简单

Promise 对象 ？ (消除异步)

用同步的方式来书写异步代码

Promise 让异步操作写起来，像在写同步操作的流程，不必一层层地嵌套回调函数

改善了可读性，对于多层嵌套的回调函数很方便

充当异步操作与回调函数之间的中介，使得异步操作具备同步操作的接口

Promise 也是一个构造函数

接受一个回调函数f1作为参数，f1里面是异步操作的代码

返回的p1就是一个 Promise 实例

所有异步任务都返回一个 Promise 实例

Promise 实例有一个then方法，用来指定下一步的回调函数

function f1(resolve, reject) {

// 异步代码... resolve 成功 reject 失败

}

var p1 = new Promise(f1);

p1.then(f2); // f1的异步操作执行完成，就会执行f2。

Promise 使得异步流程可以写成同步流程

// 传统写法

step1(function (value1) {

step2(value1, function(value2) {

step3(value2, function(value3) {

step4(value3, function(value4) {

// ...

});

});

});

});

// Promise 的写法

(new Promise(step1))

.then(step2)

.then(step3)

.then(step4);

Promise.all(promiseArray)方法

将多个Promise对象实例包装，生成并返回一个新的Promise实例

promise数组中所有的promise实例都变为resolve的时候，该方法才会返回

并将所有结果传递results数组中

promise数组中任何一个promise为reject的话，则整个Promise.all调用会立即终止，并返回一个reject的新的promise对象

var p1 = Promise.resolve(1),

p2 = Promise.resolve(2),

p3 = Promise.resolve(3);

Promise.all([p1, p2, p3]).then(function (results) {

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

});

Promise.race([p1, p2, p3])

Promse.race就是赛跑的意思

哪个结果获得的快，就返回那个结果

不管结果本身是成功状态还是失败状态

有了Promise 之后的异步：

Promis.all ( [ $.ajax( ),

$ajax( )

] ).then( right=>{

},err=>{ });

https://www.imooc.com/article/23838

###### 13.generator-认识生成器函数

generator 生成器函数

普通函数，一路到底

generator函数，中间可以停，到哪停呢，用 yield 配合，交出执行权

yield 有 放弃、退让、退位的意思(暂停执行)

需要调用next()方法启动执行，需要遇到 yield 停, 踹一脚走一步

generator函数前面加一个 \* 两边可以有空格，或靠近函数或function

背后实际生成多个小函数，实现走走停停

function show() {

console.log('a');

console.log('b');

}

show() // 普通函数

function \*show2() {

console.log('1');

Yield;

console.log('2');

}

let genObj = show2();

genObj.next(); // 1

genObj.next(); // 2

genObj.next(); // 最后了，没有结果

###### 14.generator-yield是啥

yield

既可传参，又可以返回

第一个next()传参无效，只用来启动

如果函数前漏掉 \*

就是普通函数

如果有yield会报错， ReferenceError: yield is not defined

yield 只能在Generator函数内部使用

function \* show() {

console.log('1')

var a = yield

console.log('2')

console.log(a) }

// yield 传参

var gen = show()

gen.next() // 1

gen.next() // 2 和 undefined 因为没有传参，yield没有返回值

var gen = show()

gen.next(10) // 1 第一次执行到yield，但没有执行赋值

gen.next(20) // 2 和 20

function\* show2() {

console.log('1')

yield 10

console.log('2')

}

// yield 返回var gen = show2()

var res1 = gen.next()

console.log(res1) // { value: 10, done: false }

var res2 = gen.next()

console.log(res2)

// { value: undefined, done: true } 最后的value需要return返回

###### 15.generator-实例

Promise 适合一次读一堆

generator 适合逻辑性的

// 带逻辑-generator

runner(function \* () {

let userData = yield $.ajax({url: 'getUserData'})

if (userData.type == 'VIP') {

let items = yield $.ajax({url: 'getVIPItems'})

} else {

let items = yield $.ajax({url: 'getItems'})

}

})

// yield 实例，用同步方式写异步

server.use(function \* () {

let data = yield db.query(`select \* from user\_table`)

this.body = data

})

###### 16.ES7 预览

数组 keys/values/entries (所有的key-value对拿出来)

arr.includes() 数组是否包含某个东西

数组的 arr.keys(), arr,entries()

for ... in 遍历数组 下标 key

for ... of 遍历数组 值 value, 不能用于json

let arr = ['a', 'b', 'c'];

console.log(arr.includes(1));

for (let i in arr) {

console.log(i); // 循环的时下标 key

}

for (let i of arr) {

console.log(i); // 循环的是值 value

}

for (let i of arr.keys()) {

console.log('>'+i);

}

for (let [key, value] of arr.entries()) {

console.log(`${key}=${value}`);

}

let json = { a: 12, b: 5, c: 7 };

for (let i in json) {

console.log(i);

}

求幂 Math.pow(3,8); (3\*\*8);

字符串

padStart()/padEnd() 指定宽度，不够就补空格或指定字符

console.log('=' + 'abcd'.padStart(6, '0') + '=');

console.log('=' + 'abcd'.padEnd(6, '0') + '=');

=00abcd

==abcd00=

容忍度

[1, 2, 3,] 老版数组最后不能有逗号，新的可以有

函数参数最后多的逗号也可以

async await

和 generator yield 类似

不依赖外部runner 统一 性能

generator 不可以写成箭头函数 ， async 可以

async function show() {

console.log(1)

await

console.log(2)

}

Async function readData(){

let data1=await $.ajax{ url:’data/1.txt’ ,dataType:’jason’};

let data1=await $.ajax{ url:’data/2.txt’ ,dataType:’jason’};

let data1=await $.ajax{ url:’data/3.txt’ ,dataType:’jason’};

Console.log(data1,data2,data3);

}