爱创课堂前端培训

# ES6

第2天课堂笔记（本课程共3天）

班级：北京前端训练营25期

讲师：彭帅伟

日期：2019年9月15日

爱创课堂官网 ：[www.icketang.com](http://www.icketang.com)

目录

[ES6 1](#_Toc16836)

[复习： 3](#_Toc32507)

[一、 解构 4](#_Toc8511)

[1.1 解构对象 4](#_Toc26345)

[1.2 解构数组 5](#_Toc8699)

[二、 函数拓展 6](#_Toc22598)

[2.1 默认参数 6](#_Toc7119)

[2.2 获取剩余参数 6](#_Toc1834)

[2.3 三个点语法 7](#_Toc10149)

[2.4 箭头函数 8](#_Toc20799)

[2.4.1 省略 8](#_Toc23365)

[2.4.2 特点 9](#_Toc28902)

[下午复习： 10](#_Toc31831)

[三、 Symbol数据 11](#_Toc32433)

[四、 代理 12](#_Toc11409)

[五、 Reflect 13](#_Toc23388)

[六、 聚合数据对象 14](#_Toc24614)

[6.1 Set 14](#_Toc12020)

[6.2 WeakSet 15](#_Toc14585)

[6.3 Map 16](#_Toc19916)

[6.4 WeakMap 17](#_Toc29730)

[七、 Promise规范 17](#_Toc6757)

[7.1 三个状态 17](#_Toc17596)

[7.2 动态流向 18](#_Toc29576)

[7.3 使用 18](#_Toc3616)

[7.4 简单实现Promise 18](#_Toc30739)

# 复习：

ES6：

let关键字： 用于定义变量 支持块级作用域

特点：1 支持块作用域 2 不能重复定义 3 不能声明前置 4 在for循环中可以存储变量 5 不能被window挂载

const: 1 支持块作用域 2 不能重复定义 3 不能声明前置 4 不能作为循环变量 5 不能被window挂载

字符串拓展：

多行字符串： `` {}

重复字符串： repeat

原始字符串： String.raw``

字符串的位置： startsWith(str, pos) endsWith(str, pos) includes(str, pos)

数字拓展：

isNaN :判断是否是NaN（not a number）

isFinite:判断是否是有限的

isInteger: 判断是否是整型

对象拓展：

字面量： 如果属性名和属性值变量是同名的可以省略属性名和冒号

可以通过[]语法动态的设置属性名。并且还可以使用表达式

在对象中定义方法的时候可以省略冒号和function关键字

is: 判断两个参数是否全等

assign(target): 用于复制对象

数组拓展：

from方法： 用于将类数组对象转为数组，并且可以传递第二个参数，也就是说可以遍历类数组对象

of方法：创建数组

find方法：用于查找成员

findIndex: 用于获取成员的索引值

迭代器：

数组实现了迭代器方法： 分别提供了 keys values entries

由于实现了数组迭代器方法我们可以使用for of方法遍历还可以使用next方法

# 解构

所谓解构就是解析聚合数据

聚合数据： 对象，数组

在之前我们想要获取对象中的某个属性值的时候，始终都要使用点语法

在之前我们想要获取数组中的某个成员，始终都要使用[]语法

ES6简化了这样的操作，提供了解构的语法

## 解构对象

语法: let { key1, key2, ...keys } = obj

key1 相当于 obj.key1

key2 相当于 obj.key2

keys获取剩余的属性

当没有剩余属性的时候，获取的是一个空对象

注意： 解构的属性、解构的变量一定要与对象中的属性是同名

解构问题：

1 如果使用var进行解构，将会污染全局作用域，我们可以通过let关键字来解决

2 对象方法来说，一旦解构，方法中的this将发生改变

3 对于引用类型来说，只是指针改变，对于值类型来说，是真正的复制

对象解构的逆运用，我们可以使用三个点语法，将对象中的数据传递给一个对象字面量中

demo:

|  |
| --- |
| 1. // 定义对象 2. let obj = { 3. num: 100, 4. color: 'red', 5. colors: ['red', 'green'], 6. getColor() { 7. console.log(this); 8. return this.color; 9. } 10. } 11. // 之前获取属性值 12. // console.log(obj.num); 13. // console.log(obj.color); 14. // ES6提供解构语法 15. // 使用var进行解构 16. var { num, color } = obj; 17. let { num, color } = obj; 18. console.log(num, color); 19. 使用var解构， 解构出来的变量将挂载到window 20. 解构方法 21. let { num, getColor, colors } = obj; 22. console.log(obj.getColor()); 23. console.log(getColor()); 24. 值类型复制 25. obj.num = 200; 26. 引用类型是指针改变 27. obj.colors.push('blue'); 28. 获取剩余属性 29. let { num, color, colors, getColor, ...keys } = obj; |

## 解构数组

语法： let [ arg1, arg2, ...args ] = arr;

arg1表示第一个成员

arg2 表示第二个成员

args获取的是剩余成员

如果没有剩余的成员 将得到一个空数组

如果使用var解构， 也会污染全局作用域，同样的可以使用let关键字解决

获取剩余成员的语法：

如果有剩余的成员，将获取所有的成员

如果没有剩余成员，将得到一个空数组

前面的解构的变量，可以正常使用

逆运用：我们可以通过三个点语法，将数组中的参数，传递给一个数组字面量

此时将得到一个新数组

demo:

|  |
| --- |
| 1. // 定义数组 2. let colors = ['red', 'green', 'blue', 'pink', 'orange']; 3. // 之前获取成员的方式 4. // console.log(colors[0]); 5. // console.log(colors[1]); 6. // 解构方式 7. let [ red, green, blue, ...args ] = colors; 8. // console.log(red, green, blue); 9. // console.log(args); 10. // 解构 11. let [ ...args ] = colors; 12. // 逆运用 13. let arr = [...args]; 14. console.log(arr); |

# 函数拓展

ES6为函数做了很多拓展

## 2.1 默认参数

配置函数默认参数的方式有：

1 通过||进行默认参数适配

但是使用||运算符会忽略6个值为false的参数： 0, ‘’, null, NaN，undefined, false

2 可以通过三元运算符 ？：

但是参数过的，书写起来很麻烦

所以ES6为了解决这样的问题，提供了默认参数的方法直接在参数中适配就可以了

如果传递了参数，形参就使用传递的参数（实参）

如果没有传递参数，形参就使用默认参数

这种语法与 ? : 适配默认参数是等价的。

demo:

|  |
| --- |
| 1. // ES6拓展的方式 2. function demo(arg = 'red') { 3. console.log(arg); 4. } 5. demo(); 6. demo(undefined); 7. demo(''); 8. demo(0); 9. demo(NaN); 10. demo(null); 11. demo(false); 12. demo('orange'); |

结果：

|  |
| --- |
|  |

## 2.2 获取剩余参数

之前获取参数可以使用arguments，但是arguments不是数组，不能使用数组方法

ES6拓展了获取剩余参数的语法

function demo(arg1, arg2, ...arg) {}

arg1 代表第一个参数

arg2 代表第二个参数

arg 则获取剩余的参数

获取剩余参数的语法，

可以获取所有的参数

获取的剩余参数是一个数组，不包含前面的参数

前面的参数可以正常使用

在箭头函数中有广泛的应用

如果没有获取剩余的参数，arg则是一个空数组

逆运用：

我们可以将一个数组，作为参数传递给一个方法中，

语法 demo(...arr);

apply可以将一个数组作为参数传递给函数，但是要考虑this的问题

而获取剩余参数的语法将数组传递给方法的时候，不用考虑this的指向问题。

demo:

|  |
| --- |
| 1. // 定义结果变量 2. let result = 0; 3. // 定义函数 4. function add() { 5. // console.log(arguments); 6. for (let i = 0; i < arguments.length; i++) { 7. result += arguments[i]; 8. } 9. console.log(result); 10. } 11. // ES6拓展的三个点语法 12. // 定义函数 13. function add(...args) { 14. // console.log(args); 15. // 调用累加方法 16. let result = args.length && args.reduce(function(pre, item) { 17. return pre + item; 18. }) 19. console.log(result); 20. } 21. add(); 22. add(1, 2); 23. add(1, 2, 3); 24. add(1, 2, 3, 4); 25. add(1, 2, 3, 4, 5); 26. add(1, 2, 3, 4, 5, 6); |

## 2.3 三个点语法

对象解构

let {...keys} = obj

逆运用 let obj = { ...keys }

数组解构

let [...items] = arr;

逆运用 let arr = [...items]

获取剩余参数

function demo (...args) {}

逆运用 demo(...args);

## 2.4 箭头函数

定义函数的方式： 1 构造函数式 2 函数表达式 3 函数定义式

ES6拓展了箭头函数：

语法：

let demo = () => {}

(): 表示参数结合

=> 表示箭头函数的标志

{}: 函数体

## 2.4.1 省略

1 如果参数集合中只有一个参数可以省略参数集合

注意： 如果使用三个点语法获取剩余参数，不能省略参数集合()

2 如果函数体有只有返回值或者只有一句话的时候可以省略return 和{}函数体

|  |
| --- |
| 1. // 省略 2. let demo = (msg) => { 3. console.log('hello ' + msg); 4. } 5. demo('world'); 6. 省略参数集合 7. let demo = msg => { 8. console.log('hello ' + msg); 9. } 10. demo('world'); 11. 定义数组 12. let arr = [1, 2, 3, 4, 5]; 13. 求平方数 14. let demo = (...arg) => { 15. let result = arg.map(item => { 16. return item \* item; 17. }) 18. console.log(result); 19. } 20. demo(1, 2, 3, 4) 21. 第二点省略 22. let demo = (...arg) => { 23. let result = arg.map(item => item \* item \* item); 24. console.log(result); 25. } 26. demo(1, 2, 3, 4) |

## 2.4.2 特点

1 无法使用arguments，但是我们可以通过三个点语法来获取剩余参数

2 无法作为构造函数来使用

3 箭头函数的this指向永远是定义时的，也就是说在哪里定义this就指向谁

箭头函数的this指向，无论使用call或者是apply或者是bind都无法改变它的this指向

所以，唯一可以改变箭头函数this的的方式，就是改变其寄生环境的this指向

改变外部函数的作用域

|  |
| --- |
| 1. // 严格模式 2. 'use strict'; 3. // 定义fn1 4. let fn1 = function() { 5. console.log('fn1', this); 6. } 7. // 测试this 8. let obj = { 9. fn1, 10. // 普通函数 11. fn2: function() { 12. console.log('fn2', this); 13. }, 14. // 省略语法 15. fn3() { 16. console.log('fn3', this); 17. }, 18. // 箭头函数 19. fn4: () => { 20. console.log('fn4', this); 21. }, 22. // 普通函数 23. fn5: function() { 24. console.log('fn5', this); 25. // 定义箭头函数 26. let fn6 = () => { 27. console.log('fn6', this); 28. } 29. fn6(); 30. }, 31. ickt: { 32. fn7: function() { 33. console.log('fn7', this); 34. }, 35. // 箭头函数 36. fn8: () => { 37. console.log('fn8', this); 38. } 39. } 40. } |

## 下午复习：

解构：解析聚合数据

解构对象：

语法： let { key1, key2, ...keys } = obj;

逆运用： let obj = { ...keys };

解构数组：

语法： let [arg1, arg2, ...args] = arr;

逆运用： let arr = [...args];

函数拓展：

默认参数： function demo(arg = 10) {}

获取剩余参数: function demo(...args) {}

三个点语法： demo(...arr)

箭头函数：

语法： let demo = () => {}

省略： 1 如果参数集合中只有一个参数 可以省略参数集合

2 如果函数体中只有一句话或者是返回值 即可省略return 和函数体

特点： 1 无法使用arguments 但是可以通过使用三个点的语法来获取剩余参数

2 无法作为构造函数来使用

3 this指向用于是定义时的

# Symbol数据

在js中的数据类型有6种： 数字、字符串、布尔值、undefined、 null, 对象

ES6拓展了一个叫做: Symbol类型数据 表示独一无二的数据

通过Symbol方法来创建Symbol类型

参数是对Symbo数据的描述，但是不影响结果

我们可以通过typeof 来查看Symbol数据的类型

如果参数传递的是一个对象，默认会调用其toString()方法转为字符串

我们还可以重写其toString方法

Symbol类型的数据就是为了解决对象中同名属性覆盖的问题

我们通过定义Symbol类型的数据即可解决对象中同名属性覆盖的问题

我们通过Symbol定义的数据，是无法通过for in查看的，也无法通过Object.keys方法查看数据

我们可以通过Object.getOwnPropertySymbols来获取属性名称

或者是通过Symbo变量l来获取属性数据

Object.keys:

该方法用于获取对象中所有的属性名称

demo:

|  |
| --- |
| 1. // 定义对象 2. let obj = { 3. color: 'red', 4. num: 400 5. } 6. // 定义Symbol 7. let s1 = Symbol(); 8. let s2 = Symbol('ickt'); 9. let s3 = Symbol('hello'); 10. obj[s1] = 100; 11. obj[s2] = 'hello'; 12. obj[s3] = 'ickt'; 13. // console.log(obj); 14. // 使用for in遍历 15. for (let i in obj) { 16. // console.log(i); 17. } 18. // console.log(Object.keys(obj)); 19. // 通过变量的方式获取 20. console.log(obj[s1]); 21. console.log(obj[s2]); 22. console.log(obj[s3]); 23. // 通过Object.getOwnPropertySymbols来获取属性名称 24. let keys = Object.getOwnPropertySymbols(obj); 25. console.log(keys); 26. // 获取之后在获取属性数据 27. console.log(obj[keys[0]]); 28. console.log(obj[keys[1]]); 29. console.log(obj[keys[2]]); |

# 代理

在一些应用程序中，总有一些数据比较重要，不希望别人看到，但是在js中只要可以访问到一个对象，那么该对象中的结构都可以被操作

于是ES6为了解决这样的问题，拓展了代理方式，用于拦截保护重要的数据

使用方式：

return Proxy(data, { get, set... }) {}

data: 表示目标对象

{}：

操作对象

get(obj, key) 获取属性数据

obj: 目标对象

key: 属性名称

返回值就是获取到的属性数据

this指向当前操作对象

set(obj, key, value) 用于修改属性数据的

obj：目标对象

key: 属性名称

vlaue: 属性值

this指向当前操作对象

使用的代理与之前设置特性的方式雷同

只不过设置特性是针对一个对象

而设置代码是针对两个对象

demo:

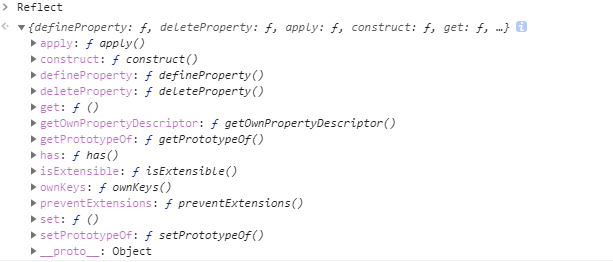
|  |
| --- |
| 1. // 定义对象 2. let star = (function() { 3. // 重要的数据 4. let data = { 5. name: '赵丽颖', 6. boyFriend: '冯绍峰' 7. } 8. return new Proxy(data, { 9. // 取值器 10. get(obj, key) { 11. // console.log(1111, this, arguments); 12. // 如果key 是boyFriend的话不能告诉其他人 13. if (key === 'boyFriend') { 14. return '不能告诉你'; 15. } 16. // 其它数据可以返回 17. return obj[key]; 18. }, 19. // 赋值器 20. set(obj, key, value) { 21. // console.log(2222, this, arguments); 22. // 如果修改的是男朋友 则不允许 23. if (key === 'boyFriend') { 24. console.log('不能修改'); 25. return; 26. } 27. // 其它数据可以修改 28. obj[key] = value; 29. } 30. }) 31. })() |

# Reflect

Reflect对对象的一些方法的封装

在之前，对象所有的方法都给Object添加了，它们都是内部方法，于是在ES6中，将这些方法拿出来给了Reflect，所以在将来一些方法可以被移除，给Reflect添加

查看：



比如：

现在设置特性：

Object.definePropety

未来：

Reflect.definProperty

has: 用于替代in运算符的， 判断对象中是否包含某个属性

Reflect可以将一些运算符作为函数来使用

deleteProperty: 该方法用于删除一个对象中某个属性

getOwnPropertyDescriptor： 用于获取一个对象中某个属性的描述对象

get方法用于获取属性数据

demo:

|  |
| --- |
| 1. let obj = { 2. num: 100, 3. color: 'red' 4. } 5. in运算符 6. console.log('color' in obj); 7. has方法 8. console.log(Reflect.has(obj, 'color')); 9. deleteProperty 10. Reflect.deleteProperty(obj, 'color'); 11. // 定义对象 12. let obj = { 13. num: 100, 14. color: 'red', 15. get doubleNum() { 16. return this.num \* 2; 17. }, 18. set doubleNum(value) { 19. this.num = value / 2; 20. } 21. } 22. // 设置 23. Reflect.set(obj, 'doubleNum', 200); 24. // 获取 25. console.log(Reflect.get(obj, 'doubleNum')); |

# 聚合数据对象

聚合数据： 数组，对象

在ES6中拓展了一些聚合数据： Set 、(WeakSet)、Map、(WeakMap)

## 6.1 Set

实现了迭代器接口对象的去重数组

在去重的时候不会做类型转换

使用new Set创建一个Set对象

Set实现了迭代器接口，因此可以使用for of方法遍历

内部提供了大量的方法用于操作Set对象

add: 添加数据

clear: 清空数据

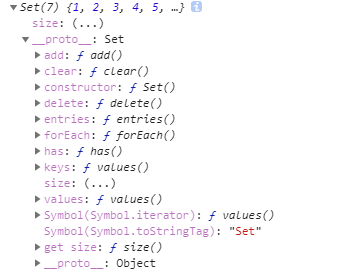
delete: 删除某一项

forEach: 用于遍历

has: 判断是否包含某一项

size: 获取成员长度

keys values entries: 得到迭代器接口对象



## 6.2 WeakSet

是弱set对象，参数只能是对象

通过new WeakSet创建WeakSet对象

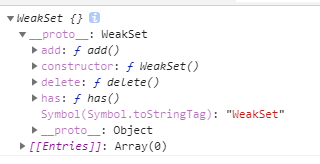
内部也提供了一些少量方法用于操作该对象

add: 添加成员

delete: 删除成员

has: 判断某个成员是否存在

查看：



demo:

|  |
| --- |
| 1. // WeakSet 2. var ws = new WeakSet(); 3. // 添加数据 4. ws.add(window); 5. ws.add([]); 6. ws.add({}); 7. console.log(ws); |

结果：

|  |
| --- |
|  |

## 6.3 Map

是一个超级对象

在啊ES5中，给对象添加属性的时候只能是字符串

但是Map对象，定义对象中的属性可以是任意类型（7中类型都可以）

内部也提供了大量的方法，用于操作该对象

clear: 清空数据

delete: 删除某一项数据

forEach: 遍历数据

get: 获取属性数据

set: 设置数据

size: 获取成员长度

has: 判断某一项数据是否存在

keys values entries: 得到迭代器接口对象

查看：



demo:

|  |
| --- |
| 1. // 创建Map对象 2. let map = new Map(); 3. // 设置数据 4. map.set(null, 1); 5. map.set(window, 2); 6. map.set([], 2); 7. map.set({}, 2); 8. map.set(undefined, 2); 9. map.set(1, 2); 10. console.log(map); |

结果：

|  |
| --- |
|  |

## 6.4 WeakMap

弱Map对象

使用new WeakMap创建

参数必须是对象

内部提供一些必要的方法用于该对象

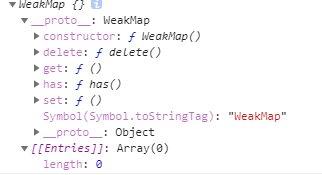
delete: 删除某一项数据

get: 获取属性数据

set: 设置数据

size: 获取成员长度

查看：



# Promise规范

Promise规范是将异步操作的写法变为同步操作的写法的一种规范

在异步操作中，有广泛的应用

只是写法的改变，操作没有改变

异步操作：在回调函数中，一层一层的嵌套

同步操作：将方法写在外部

## 三个状态

在Promise中有三个状态：

1 pending 操作正在操作

2 resolved 操作结束执行完毕

3 rejected 操作失败执行完毕

## 动态流向

在Promise中的一个动态流向有两个方向：

1 是由pending状态，流向为resolved状态 说明操作成功的执行完毕

2 是由pending状态，流向状态为rejected 状态，说明操作失败的执行完毕

## 使用

new Promise((resolve, reject) => {

在这里面执行异步操作， 执行成功

resolve()

执行失败 则执行

reject()

})

通过then方法监听异步操作成功还是失败

.then(success, fail)

success: 成功时候执行的回调函数

函数中的参数就是resolve执行的时候传递的参数

fail: 失败时候执行的回调函数

函数中的参数就是reject执行的时候传递的参数

then方法的返回值是Promise对象，因此我们可以进行链式调用

上一个then方法返回值将作为下一个then方法中的参数使用

注意：

多次调用then方法，如果失败，第一个then方法调用失败回调函数，后面的方法执行成功的回调函数。

我们可以逐一在失败回调函数中，返回错误的数据，此时就会执行失败的回调函数，而不是执行成功的回调函数了

如果前面的操作已经执行完毕，此时所有的then方法立即执行。

## 7.4 简单实现Promise

|  |
| --- |
| 1. // 定icktPromise 2. let IcktPromise = function(callback) { 3. // 定义状态 4. this.status = 'pending'; 5. // 定义操作成功执行完毕之后的队列 6. this.resolvedArr = []; 7. // 定义操作失败执行完毕之后的队列 8. this.rejectedArr = []; 9. // 定义resolve方法 10. let resolve = (value) => { 11. // 缓存value 12. this.doneValue = value; 13. // 改变状态 14. this.status = 'resolved'; 15. // 执行操作成功执行完毕之后的队列 16. this.resolvedArr.forEach(fn => value = fn(value)); 17. } 18. // 定义reject方法 19. let reject = (value) => { 20. // 缓存value 21. this.doneValue = value; 22. // 改变状态 23. this.status = 'rejected'; 24. // 执行操作失败执行完毕之后的队列 25. this.rejectedArr.forEach(fn => value = fn(value)); 26. } 28. try { 29. callback(resolve, reject); 30. } catch (e) { 31. reject(e); 32. } 33. } 34. // 添加then方法 35. IcktPromise.prototype.then = function(success, fail) { 36. // 判断状态 37. if (this.status === 'pending') { 38. // 使用&& 优化 39. typeof success === 'function' && this.resolvedArr.push(success); 40. typeof fail === 'function' && this.rejectedArr.push(fail); 41. } else if (this.status === 'resolved') { 42. // 立即执行 43. typeof success === 'function' && success(this.doneValue); 44. } else { 45. typeof fail === 'function' && fail(this.doneValue); 46. } 47. // 链式调用 48. return this; 49. } |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |