爱创课堂前端培训

# ES6

第3天课堂笔记（本课程共3天）

班级：北京前端训练营19期

讲师：李兰波

日期：2019年2月18日

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

# 复习

对象方法的拓展

Object.is 该方法用于比较两个值是否完全相等

Object.assign 该方法用于将从第二个参数开始的剩余参数的所有属性和方法赋予第一个参数 浅复制

数字方法

Number.isNaN 判断一个数字是否是NaN

Number.isFinite 判断一个数字是否是有限的

Number.isInteger 判断一个数字是否是整数

代理

一个新的构造函数 Proxy

接收两个参数

第一个参数是被保护的对象

第二个参数是当通过代理对象访问时 如何处理的对象 该对象有set方法和get方法

Symbol

一个新的基本数据类型 表示一种独一无二的符号

...语法

在函数中获取剩余参数

解构数组时 将剩余的数组项组成一个数组

在传递参数时，可以将数组拆分成参数列表并传递

解构

将变量组织成对象叫做组成结构

将对象拆分成变量叫做解除结构

解构数组

let [var1, var2, var3……] = arr;

解构对象

let {prop1, prop2, prop3, prop4} = obj;

箭头函数

语法 () => {}

不能作为构造函数 不能够再被new调用

没有arguments 想要获取剩余参数 需要通过...语法来获取

this的指向与定义该函数的时候所在的作用域中的this有关

参数默认值 可以通过在形参之后使用=给定一个参数默认值

省略写法

当形参只有一个时 可以省略圆括号

当函数体中只有一条代码并且还是返回语句时 可以省略{} 和 return

迭代器

ES6中已经实现了迭代器的接口 并且有一部分数据结构已经实现了迭代器

定义 在不暴露内部结构的情况下 顺序的访问数据

数组的迭代器方法

keys、 values、 entires

当调用这三个方法时 会返回一个迭代器对象 该对象可以通过调用next方法进行迭代一次

返回的数据的格式：

value： 迭代的结果 根据方法的不同而不同

done：是否迭代完成

for……of循环

该循环语句只可以循环实现了迭代器接口的那些数据

里面的循环变量就是next方法返回的对象中的value

新的数据结构

以前的数据结构只有数组和对象 ES6中又新增了4种

Set：一个去重数组 里面每一个值只能够保存一次

WeakSet：一个弱Set 里面的每一个值都必须是引用类型 而且它所保存的内容不会被垃圾回收机制计入次数

Map：一个超对象 里面的每一个键和值都可以是任意类型

WeakMap：一个弱Map 里面的每一个键都必须是引用类型 而且它所保存的内容不会被垃圾回收机制计入次数

对象的字面量的改变

var obj = {

如果key和value相同则只需要书写一个名称即可

可以在定义字面量中使用[]语法定义属性名

当定义方法时 可以省略: function

}

# Generator

## 1.1 简介

Generator是一个新的函数类别。

定义方式 function \* name() {}

函数体内的定义方式:

|  |
| --- |
| 1. function \* generator() { 2. yield "1"; 3. yield "2"; 4. yield "3"; 5. yield "4"; 6. yield "5"; 7. yield "6"; 8. } |

当函数执行的时候，得到的不是代码中的执行结果。而是一个迭代器对象。通过该迭代器对象来执行内部的代码。

每当迭代器对象调用一次next就会执行里面的一段代码。

注：它不是构造函数，不可以使用new去调用。



初始化:

|  |
| --- |
| 1. var g = generator(); |

输出g:

|  |
| --- |
|  |

demo:

|  |
| --- |
| 1. // 定义一个状态函数 2. var generator = function \* () { 3. // 定义第1段代码 4. console.log(1); 5. yield "a"; 6. // 定义第2段代码 7. console.log(2); 8. yield "b"; 9. // 定义第3段代码 10. console.log(3); 11. yield "c"; 12. // 定义第4段代码 13. console.log(4); 14. yield "d"; 15. // 定义第5段代码 16. console.log(5); 17. yield "e"; 18. // 定义第6段代码 19. console.log(6); 20. yield "f"; 21. // 定义第7段代码 22. console.log(7); 23. yield "g"; 24. // 定义第8段代码 25. console.log(8); 26. yield "h"; 27. } 28. // var g = new generator(); // 不可以使用new去调用 29. // 执行generator 得到一个迭代器 30. var g = generator(); 31. console.log(g); 32. // 绑定点击事件 每点击一次 就会执行一次g.next() 就会执行一段函数中的代码 33. document.onclick = function() { 34. console.log(g.next()); 35. } |

每当点击一次document时：

|  |
| --- |
|  |

再次点击：

|  |
| --- |
|  |

再次点击：

|  |
| --- |
|  |

g.next()有返回结果，是一个对象 有value属性有done属性

value是什么要看本次的yield后面跟的是什么。

## 1.2 解决异步编程问题

|  |
| --- |
| 1. // 定义一个generator函数 2. var gen = function \* () { 3. // 定义第一个任务 4. yield taskOne(); 5. // 定义第二个任务 6. yield taskTwo(); 7. } 8. // 定义第一个任务 9. function taskOne() { 10. $.ajax({ 11. url: "/a", 12. type: "get", 13. data: "", 14. dataType: "json", 15. success: function() { 16. console.log("发送第一个任务结束"); 17. g.next(); 18. } 19. }) 20. } 21. // 定义第二个任务 22. function taskTwo() { 23. $.ajax({ 24. url: "/b", 25. type: "get", 26. data: "", 27. dataType: "json", 28. success: function() { 29. console.log("发送第二个任务结束") 30. } 31. }) 32. } 33. // 获取迭代器对象 34. var g = gen(); 35. g.next(); |

书写顺序：将各个任务单独拆分出去，并封装成函数 最后，将每一个函数任务排列在generator中 代码纵向发展。

## 1.3 传递数据

从内部向外部传递数据的方式

传递:

|  |
| --- |
| 1. yield "nihao"; |

接收:

|  |
| --- |
| 1. var result = g.next().value; |

从外部向内部传递数据

传递:

|  |
| --- |
| 1. g.next("nihao"); // g.next可以传递参数 只能传递一个 |

接收:

|  |
| --- |
| 1. var result = yield taskOne(); |

# Promise

这也是一个异步编程的解决方案。

它的作用是将异步代码的书写方式变为同步书写。

## 2.1 演示

|  |
| --- |
| 1. // Promise是一个构造函数 2. // 接收一个函数作为参数 通常该函数中必须要有异步代码 3. var p = new Promise(function(resolve, reject) { 4. // 发出ajax请求 5. $.ajax({ 6. url: "/a", 7. dataType: "json", 8. success: function(data) { 9. // 在回调函数中 我们可以根据返回的数据的error属性判定这是一个成功的请求还是一个失败的请求 10. if (!data.error) { 11. // 这是一个成功的请求 就执行resolve 12. resolve(); 13. } else { 14. // 这是一个失败的请求 就执行reject 15. reject(); 16. } 17. } 18. }); 19. }); 20. // 为了处理回调函数中的内容 p对象提供了then方法用于监听Promise中异步代码的执行情况 21. p.then(function() { 22. console.log("成功"); 23. }, function() { 24. console.log("失败"); 25. }); 26. 现在的处理代码不再位于回调函数中，而位于发送代码的下方。与同步代码的书写方式一致。 |

## 2.2 详细介绍

Promise是一个构造函数 接收一个函数参数

Promise所接收的函数是一段立即被执行的代码通常我们会在它的内部设置异步请求的发送代码。

它有两个参数 分别是resolve 和reject。它们都是函数。

当resolve执行的时候，会将Promise实例的状态从pending变成resolve

当reject执行的时候，会将Promise实例的状态从pending变成reject

Promise的返回值是一个Promise实例。它就表示一个Promise对象。它有3种状态。分别对应请求的三种状态。

pending: 发送中



resolve: 成功



reject: 失败



p对象的状态可能性有两种

pending => resolve

pending => reject

p有一个方法then。它接收两个函数参数，分别对应resolve和reject。当状态变化时，会自动执行对应的函数。

## 2.3 依次发出多个请求

发送一次请求，并监听该请求。我们已经通过2.1案例演示过。

现在我们要做的是：在第一次请求回来之后，再发出新的请求。

demo:

|  |
| --- |
| 1. // 使用Promise连续发送两个请求 并且第二个请求要携带第一个请求得到的数据 2. // 初始化第一个 3. var p = new Promise((resolve, reject) => { 4. // 发送第一个ajax 5. $.ajax({ 6. url: "/a", 7. dataType: "json", 8. success: (data) => { 9. console.log(data); 10. resolve(); 11. } 12. }); 13. }); 14. // 将Promise代码放在一个普通函数中 这样不会立即执行 15. function sendAjax2() { 16. // 返回Promise的实例 17. return new Promise((resolve, reject) => { 18. $.ajax({ 19. url: "/b", 20. dataType: "json", 21. success: (data) => { 22. console.log(data); 23. resolve(); 24. } 25. }); 26. }); 27. } 28. // 监听p状态的改变 29. p.then(() => { 30. console.log("成功"); 31. // return一个Promise实例 这样我们可以继续打点调用then方法 32. return sendAjax2(); 33. }, () => { 34. console.log("失败"); 35. }) 36. // 此时 因为上一个函数中返回了Promise的实例 所以这个then监听的就不是第一个p了 而是新的Promise的实例 37. .then(() => { 38. console.log("成功1"); 39. }, () => { 40. console.log("失败1"); 41. }) |

注：因为new Promise所接收的函数会立即被执行，所以为了暂缓执行，我们通常会将初始化Promise的代码放在一个函数内。等到需要的时候再执行。

## 2.4 传递参数

Promise将后续代码从回调函数中，转移到p.then的函数中。

传递参数的方式：resolve或者reject执行的时候可以传递一个参数。

demo:

|  |
| --- |
| 1. // 定义第一个Promise 2. function sendAjax1() { 3. // 返回一个Promise 4. return new Promise((resolve, reject) => { 5. // 发送ajax 6. $.ajax({ 7. url: "/a", 8. data: "", 9. type: "get", 10. dataType: "json", 11. success: (data) => { 12. // 传递四个参数 13. resolve(data, 1111, 222, 33); 14. } 15. }); 16. }); 17. } 18. // 执行第一个 19. var p = sendAjax1(); 20. // 绑定事件 21. p.then((...data) => { 22. // 获取全部参数 发现只得到一个 23. console.log(data); 24. }, () => { 25. console.log("failed"); 26. }) |

此时，data是一个数组，内部是全部传递过来的参数。

|  |
| --- |
|  |

注：resolve和reject只能够传递一个参数。

## 2.5 监听多个Promise

Promise.all可以同时监听多个Promise实例。

参数就是数组，数组中的每一个成员都是Promise实例。

demo:

|  |
| --- |
| 1. // 定义第一个Promise 2. function sendAjax1() { 3. // 返回一个Promise 4. return new Promise((resolve, reject) => { 5. // 发送ajax 6. $.ajax({ 7. url: "/a", 8. data: "", 9. type: "get", 10. dataType: "json", 11. success: (data) => { 12. // 传递四个参数 13. resolve(data, 1111, 222, 33); 14. } 15. }); 16. }); 17. } 18. // 定义第二个Promise 19. function sendAjax2(data) { 20. // 返回一个Promise 21. return new Promise((resolve, reject) => { 22. // 发送ajax 23. $.ajax({ 24. url: "/b", 25. data: data, 26. type: "get", 27. dataType: "json", 28. success: (data) => { 29. resolve(data); 30. } 31. }); 32. }); 33. } 34. // Promise.all是一个方法 可以监听多个Promise 它的特点是：当所有的Promise都完成的时候执行某些事情 35. var p = Promise.all([sendAjax1(), sendAjax2()]); 36. p.then((arr) => { 37. console.log(arr); 38. }, (data) => { 39. console.log(data); 40. }) |

Promise.all返回的也是一个Promise实例。但是该实例的状态取决于数组内的每一个成员的状态。如果都成功，则该实例的状态变为resolved.只要有任何一个成员的状态变为rejected，则该实例的状态变为rejected。对应的函数执行。

当成功时。因为每一个数组中的Promise都可能传递数据回来。所以，成功的回调函数中的参数，是一个数组。数组的每一项成员就是每一个Promise实例传递回来的数据。与Promise.all所接收的数组一一对应。

|  |
| --- |
|  |

当失败时，该实例的状态直接变为rejected，并执行失败回调函数。函数中的内容就是数组中失败的那个Promise所传递回来的数据。

除了Promise.all之外，还有一个Promise.race。用来监听第一个请求。

|  |
| --- |
| 1. // Promise.race是一个方法 可以监听多个Promise 它的特点是：只要有一个完成就完成。 2. var p = Promise.race([sendAjax1(), sendAjax2()]); 3. p.then((arr) => { 4. console.log(arr); 5. }, (data) => { 6. console.log(data); 7. }) |

## 2.6 catch

catch方法是Promise实例的方法。等于then接收的第二个参数。

作用：在连续的then中，如果每一次都写一个处理错误的函数，那么会显得代码很臃肿。于是，将第二个参数省略，并统一放在catch中进行处理。

demo:

|  |
| --- |
| 1. // 定义第一个Promise 2. function sendAjax1() { 3. // 返回一个Promise 4. return new Promise((resolve, reject) => { 5. // 发送ajax 6. $.ajax({ 7. url: "/a", 8. data: "", 9. type: "get", 10. dataType: "json", 11. success: (data) => { 12. // 传递四个参数 13. // reject("第一个错误"); 14. resolve(); 15. } 16. }); 17. }); 18. } 19. // 定义第二个Promise 20. function sendAjax2(data) { 21. // 返回一个Promise 22. return new Promise((resolve, reject) => { 23. // 发送ajax 24. $.ajax({ 25. url: "/b", 26. data: data, 27. type: "get", 28. dataType: "json", 29. success: (data) => { 30. // resolve(data); 31. reject("第二个错误") 32. } 33. }); 34. }); 35. } 36. // 执行第一个Promise 37. var p = sendAjax1(); 38. p.then(() => { 39. return sendAjax2(); 40. }) 41. .then(() => { 43. }) 44. // .catch表示捕获 当之前的代码中Promsie的状态变为rejected 或者出现异常时 会调用catch捕获内部的错误 45. .catch((msg) => { 46. console.log(msg) 47. }) |

以上的catch的作用：当第一个Promise出错时，处理第一个错误。当第二个Promise出错时，处理第二个错误。

# 类

ES6中，实现了class关键字。

该关键字用于定义类。

## 3.1 定义类

语法：

|  |
| --- |
| 1. class className {} |

demo：

|  |
| --- |
| 1. // 定义类 2. class Student { 3. } 4. // 初始化实例 5. var s = new Student(); 6. console.log(s); |

输出结果：

|  |
| --- |
|  |

## 3.2 定义构造函数

以前的构造函数，是没有类时的替代品。

现在的构造函数真的就是类中的一个函数。

举例:

|  |
| --- |
| 1. // 定义类 2. class Student { 3. // 定义本类的构造函数 4. constructor (name, age, sex) { 5. this.name = name; 6. this.age = age; 7. this.sex = sex; 8. } 9. } 10. // 初始化 11. var s = new Student("小明", 20, "男"); 12. console.log(s); |

输出:

|  |
| --- |
|  |

## 3.3 定义普通方法

在类体内定义方法

|  |
| --- |
| 1. // 定义类 2. class Student { 3. // 定义本类的构造函数 4. constructor (name, age, sex) { 5. this.name = name; 6. this.age = age; 7. this.sex = sex; 8. } 9. // 定义方法 10. sayHello() { 11. console.log("你好，我的名字是" + this.name + "我的年龄是" + this.age + "我的性别是" + this.sex); 12. } 13. } 14. // 初始化 15. var s = new Student("小明", 20, "男"); 16. console.log(s); |

输出结果：类体内定义的普通方法 效果等于定义在原型上

|  |
| --- |
|  |

调用：s.sayHello();

|  |
| --- |
|  |

## 3.4 定义静态方法

静态方法：类调用的方法 比如: Array.of、 Array.from、 Object.is、 Object.freeze等。

定义方式:

|  |
| --- |
| 1. class Student { 2. static hi() {} 3. } |

调用时：

|  |
| --- |
| 1. Student.hi(); |

demo:

|  |
| --- |
| 1. // 定义类 2. class Student { 3. // 定义静态方法 4. static hi() { 5. console.log("你好"); 6. } 7. } 8. // 调用静态方法 9. Student.hi(); |

输出:

|  |
| --- |
|  |

## 3.5 定义存取器方法

存取器方法是用于获取、设置实例的属性

demo:

|  |
| --- |
| 1. // 定义类 2. class Student { 3. // 定义构造函数 4. constructor(name, age, sex) { 5. this.name = name; 6. this.age = age; 7. this.sex = sex; 8. } 9. // 定义存取器方法 10. get getName() { 11. return this.name; 12. } 13. set setName(value) { 14. console.log("123"); 15. this.name = value; 16. } 17. } |

初始化：

|  |
| --- |
| 1. var s = new Student("xiaobai", 13, "男"); |

通过存取器属性进行获取:

|  |
| --- |
| 1. console.log(s.getName); |

通过存取器属性进行设置：

|  |
| --- |
| 1. s.setName = "hahaha"; |

## 3.6 定义静态存取器方法

静态存取器方法设置的是类本身的属性

|  |
| --- |
| 1. // 定义类 2. class Student { 3. // 定义静态存取器方法 4. static set setAge(value) { 5. // console.log("设置静态属性"); 6. this.age = value; 7. } 8. static get getAge() { 9. // console.log("获取静态属性"); 10. return this.age; 11. } 12. } 13. // 先获取一次该属性的值 14. console.log(Student.getAge); 15. // 静态属性是给类使用的 16. Student.setAge = 123; 17. // 再获取一次该属性的值 18. console.log(Student.getAge); |

注：以上代码都是类自身在调用

## 3.7 继承

继承语法： extends

demo:

|  |
| --- |
| 1. // 定义父类 2. class People { 3. constructor(name, age, sex) { 4. this.name = name; 5. this.age = age; 6. this.sex = sex; 7. } 8. sayHello() { 9. console.log(this.name); 10. } 11. static hi() { 12. console.log("hi"); 13. } 14. get getName() { 15. return this.name; 16. } 17. set setName(value) { 18. this.name = value; 19. } 20. static get getAge() { 21. return this.age; 22. } 23. static set setAge(value) { 24. this.age = value; 25. } 26. } 27. // 定义子类 28. class Student extends People { 29. constructor(name, age, sex, grade) { 30. // 构造函数式继承 31. super(name, age, sex); 32. // 赋值自己的特有属性 33. this.grade = grade; 34. } 35. } 36. var s = new Student("小明", 12, "男", 6); |

输出s：

|  |
| --- |
|  |

继承属性：

|  |
| --- |
| 1. // 定义子类 2. class Student extends People { 3. constructor(name, age, sex, grade) { 4. // 构造函数式继承 5. super(name, age, sex); 6. // 赋值自己的特有属性 7. this.grade = grade; 8. } 9. } |

如果不调用super: 会报错

|  |
| --- |
|  |

注：继承时，子类将父类的一切都继承过来。

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |