爱创课堂前端培训

# ES6

第1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1日

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

# ES6

ES6又叫做ECMAScript2015。

在ES6之前，每一次发布的ES版本，分别叫做ES3、 ES3.1、ES5、ES6。

从ES6开始，ECMA决定每一年发布一个新的版本。ES7是2016年发布的。ES8是2017年发布的。ES9是2018年发布的。

# 作用域

在ES6之前，只有函数作用域。（ES5的严格模式下，也是出现了块级作用域的）

在ES6中，块级作用域就正式出现。

所谓的函数作用域，指的是一个函数内算做一个作用域。

例如：在ES6之前

|  |
| --- |
| 1. // 最外层是全局作用域 2. function a() { 3. // 函数作用域 4. function b() { 5. // 函数作用域 6. function c() { 7. // 函数作用域 8. if (true) { 9. var a = 10; // 这个a定义在作用域c下了。if所带来的括号不算做一个作用域。 10. } 11. } 12. } 13. } |

在ES6中，正式出现了块级作用域，也就是 if while 等带来的大括号也会生成一个作用域。

# Let

ES6出现了，块级作用域也出现了。如果直接将所有的if、while等大括号重新定义成块级作用域。那么有一个问题：“以前写的代码会不会出错？”。

为了保证以前的代码不出问题，于是，ES6就决定，新增一个变量定义方式。

以前的变量是使用var定义的。所以if的{}是包不住它的作用域的。

现在的变量是使用let定义的。所以if的{}是能够闭住它的作用域的。

## 3.1 定义变量

let关键字用于定义变量。它所定义出来的变量遵循的是ES6规范。

var关键字依旧可用。它所定义出来的变量遵循的是ES6之前的规范。

比较：

|  |
| --- |
| 1. if (true) { 2. // 通过var定义变量 3. var a = 10; 4. // 通过let定义变量 5. let b = 11; 6. // 在块里面 输出a和b 7. console.log(a); 8. console.log(b); 9. } 10. // 在这里输出a和b 11. console.log(a); 12. console.log(b); |

输出结果：

|  |
| --- |
|  |

结论：

var依旧可用

定义的变量 依旧遵循之前的规则

let可以用来定义变量

let定义的变量受块级作用域的约束

## 3.2 变量声明提升取消

let定义的变量没有声明提升

demo:

|  |
| --- |
| 1. // 提前输出 2. console.log(a); 3. console.log(b); 4. // 使用var和let分别定义变量 5. var a = 10; 6. let b = 11; |

结果：

|  |
| --- |
|  |

## 3.3 不可以重复定义

var 定义的变量可以多次var

let 定义的变量只可以定义一次

demo:

|  |
| --- |
| 1. // var定义变量 2. var a = 10; 3. var a = 11; 4. var a = 12; 5. console.log(a); 6. // let定义变量 7. let b = 11; 8. let b = 12; 9. console.log(b); |

结果：

|  |
| --- |
|  |

## 3.4 在for循环中

当使用var定义循环变量，并在循环中给某些个元素绑定事件，输出该循环变量时。会输出i的最大值。

使用let定义循环变量就没有这个问题。

|  |
| --- |
| 1. // 得到元素 2. var lis = document.getElementsByTagName("li"); 3. // 给每一个元素绑定事件 4. // for (var i = 0; i < lis.length; i++) { 5. // lis[i].onclick = function() { 6. // console.log(i); // 不论点谁，都会输出10 7. // } 8. // } 9. for (let i = 0; i < lis.length; i++) { 10. lis[i].onclick = function() { 11. console.log(i); // 点谁，就输出谁的序号 12. } 13. } |

点击结果：

|  |
| --- |
|  |

# Const

这是一个新的关键字。与let同时出现在ES6中。

let定义的叫做变量。

const定义的叫做常量。或者叫做静态变量。

常量：一经定义， 只能访问， 不能修改的量，叫做常量

## 4.1 定义常量

定义常量的时候，我们推荐使用全部大写字母定义常量。如果中间出现了多个单词，那么每一个单词之间使用下划线分隔。

demo:

|  |
| --- |
| 1. // 定义常量 2. const PI = 3.14; 3. console.log(PI); |

当尝试修改常量值时：

|  |
| --- |
|  |

## 4.2 特点

常量也遵循块级作用域，也遵循不可重复定义的特点。

## 4.3 const管的是变量不是值

如果使用var 来定义变量 或者let定义变量 那么变量保存任何东西，都可以。而且随时可变。

如果使用const来定义变量 那么变量只能够在初始化的时候去保存一个东西。自此之后，就不可变。也就是不能够再去保存别的内容了。但是如果保存的是一个引用类型地址，则可以给该地址添加或者减少内容。这是允许的。

也就是说const管的是变量与值之间的对应关系是否可变。至于值是否增加或者减少内容。const不管。

举例：

|  |
| --- |
| 1. var a = {}; 2. a = []; // a 把原来的对象丢弃 保存了一个新的数组 3. const A = {}; 4. A = []; // 报错 5. A.a = 10; // 允许 |

补充：let定义变量、const定义常量时，不会将变量当做window的属性。

ES3中： 可以通过单例模式去模拟常量

ES5中： 可以通过冻结的方式去模拟常量

（还是通过变量去暴露出去 如果把白能量直接指向别的地方 那还是会更改掉所有的）

# 字符串方法

## 5.1 startsWith

该方法用于判定字符串是否是以某一个字符串开头

startsWith(str, pos);

第一个参数 是开头字符串

第二个参数 是偏移量

demo:

|  |
| --- |
| 1. // 定义一个字符串 2. var str = "今天可以回家了"; 3. // 定义一个开头字符串 4. var str1 = "可以"; 5. // 检测str1在str中的哪个位置 6. var result = str.startsWith(str1) // 此时检测的是是否str以str1开头 返回的结果是一个布尔值 7. console.log(result); // false 8. // 检测str1是否在str的下标2位置 9. var result1 = str.startsWith(str1, 2); 10. console.log(result1); // true |

结果：

|  |
| --- |
|  |

实现该方法：

|  |
| --- |
| 1. // 实现startsWith方法 2. String.prototype.startsWith = function(str, pos) { 3. // 第一种方案 先以pos截取一个新的字符串 检测str是否在该新字符串的开头 indexOf 4. // var newStr = this.slice(pos); 5. // if (newStr.indexOf(str) === 0) { 6. // return true; 7. // } 8. // return false; 9. // 第二种方案 先以pos截取一个新的字符串 检测str是否在该字符串的开头 正则表达式 10. var reg = new RegExp("^" + str); 11. （RegExp是正则表达式的函数 有两个参数 1.表达体 2.修饰符 两个参数都是字符串） 12. （修饰符m: 多行匹配； i： 不区分大小写； g: 匹配全局） 13. return reg.test(this.slice(pos)); 14. } 15. var result = str.startsWith(str1, 2) // 此时检测的是是否str以str1开头 返回的结果是一个布尔值 16. console.log(result); // false |
|  |

正则表达式有两种定义方式：

1 字面量

|  |
| --- |
| 1. var reg = /\w+/g; |

2 构造函数

|  |
| --- |
| 1. var reg = new RegExp("\\w+", "mig"); |

注：因为正则表达式构造函数定义方式接收的是字符串 而\在字符串中也是转义的意思。所以为了让\真的用到正则中，需要两个\\

|  |
| --- |
| 1. var reg = /\w+/; 2. console.log(reg) 3. var reg1 = new RegExp("\w+"); 4. console.log(reg1) |

结果：

|  |
| --- |
|  |

## 5.2 endsWith

str.endsWith(str, pos);

str: 结尾字符串

pos: 偏移量

返回值： 布尔值

demo:

|  |
| --- |
| 1. // 定义一个字符串 2. var str = "今天可以回家了"; 3. // 定义一个开头字符串 4. var str1 = "回家"; 5. // 判定 str是否是以str1结尾 6. var result = str.endsWith(str1, 5); 7. console.log(result); |

结果:、

|  |
| --- |
|  |

还原：

|  |
| --- |
| 1. // 还原 2. String.prototype.endsWith = function(str, pos) { 3. // 截取一个新字符串 4. var newStr = this.slice(0, pos); 5. // 生成正则表达式 6. var reg = new RegExp( str + "$"); 7. return reg.test(newStr); 8. } |

## 5.3 includes

str.includes(str1, pos); 该方法用于判定是否包含一个字符串

str1: 被包含的字符串

pos: 偏移量

return: 布尔值

demo:

|  |
| --- |
| 1. // 判定str是否包含str1 2. var result = str.includes(str1); 3. console.log(result); |

结果：

|  |
| --- |
|  |

还原：

|  |
| --- |
| 1. String.prototype.includes = function(str, pos) { 2. var newStr = this.slice(pos); 3. var reg = new RegExp(str); 4. return reg.test(newStr); 5. } |

## 5.4 多行字符串

在之前的版本中，我们定义字符串可以使用""、 ''、 定义

但是，它们定义出来的都是单行字符串。里面不可以包含回车，换行等内容。

在ES6中，新增了一个字符串的定义方式： ``。

demo:

|  |
| --- |
| 1. // 多行字符串 2. var str2 = `adsads 3. asdasd 4. sadasd 5. asdasd 6. asdasd 7. dsfasd`; |

输出:

|  |
| --- |
|  |

## 5.5 多行字符串中的插值语法

ES6的多行字符串可以当做模板去使用。

插值语法： ${} 大括号内是一个JS执行环境 可以调用任何方法语法等

demo:

|  |
| --- |
| 1. // 定义一个数组 2. var arr = [ 3. {username: "wanglao1"}, 4. {username: "wanglao2"}, 5. {username: "wanglao3"}, 6. {username: "wanglao4"}, 7. {username: "wanglao5"}, 8. {username: "wanglao6"}, 9. {username: "wanglao7"} 10. ]; 11. // 将arr中的每一项的内容上树 12. var list = document.getElementById("list"); 13. // 循环数组 并上树 14. for (var i = 0; i < arr.length; i++) { 15. // 如果是之前 16. // var li = "<li><span>姓名:</span><span>" + arr[i].username + "</span></li>"; 17. // 现在有了多行字符串模板功能 18. var li = `<li><span>姓名:</span><span>${arr[i].username.toUpperCase() + 1}</span></li>`; 19. list.innerHTML += li; 20. } |

## 5.6 repeat方法

该方法用于将某一个字符串重复多少次

demo:

|  |
| --- |
| 1. var str = "☆"; 2. var str1 = str.repeat(10); 3. console.log(str1); |

输出结果:

|  |
| --- |
| 1. ☆☆☆☆☆☆☆☆☆☆ |

# 数组方法

## 6.1 静态方法of

Array.of 该方法用于生成数组

参数就是数组成员

demo:

|  |
| --- |
| 1. var arr = Array.of(1, 2, 3, 4, 5); 2. console.log(arr); |

输出结果：

|  |
| --- |
|  |

## 6.2 静态方法from

Array.from方法用于将类数组对象转为数组。

参数就是类数组对象

demo:

|  |
| --- |
| 1. // 类数组对象：指的是像数组，具备下标属性值，具备length属性 但是没有数组方法的对象。 2. var lis = document.getElementsByTagName("li"); 3. // 之前在将类数组对象转为数组时 4. var arr = [].slice.call(lis); 5. // 现在在ES6中 我们可以使用Array.from方法 6. var arr1 = Array.from(lis); 7. console.log(arr); 8. console.log(arr1); |

输出结果：

|  |
| --- |
|  |

## 6.3 find方法

该方法用于模糊查询数组中的内容

arr.find(handler);

handler: 函数 用于定义查找条件 如果某一个数组成员符合该函数的描述 就返回该成员

返回值：数组成员

demo:

|  |
| --- |
| 1. // 定义一个数组 2. var arr = ["张三", "李四", "王五", "赵六"]; 3. // 查询数组中第一个带字符串“四”的成员 4. var username = arr.find(function(value, index) { 5. console.log(value, index); 6. return value.includes("四"); 7. }); |

输出结果：

|  |
| --- |
|  |

还原：

|  |
| --- |
| 1. // 还原 2. Array.prototype.find = function(handler) { 3. // 循环执行函数 并把成员和索引传递进去 4. for (var i = 0; i < this.length; i++) { 5. // 获取函数的执行结果 6. var result = handler(this[i], i); 7. // 如果为true 就说明找到了 不需要再继续循环 8. if (result === true) { 9. // 返回当前成员 10. return this[i]; 11. } 12. } 13. } |

## 6.4 findIndex

该方法用于模糊查询数组中的内容的索引值

arr.find(handler);

handler: 函数 用于定义查找条件 如果某一个数组成员符合该函数的描述 就返回该成员的索引

返回值：数组成员的索引

demo:

|  |
| --- |
| 1. // 定义一个数组 2. var arr = ["张三", "李四", "王五", "赵六"]; 3. var index = arr.findIndex(function(value, index) { 4. console.log(value, index); 5. return value.includes("四"); 6. }); |

输出结果：

|  |
| --- |
|  |

还原：

|  |
| --- |
| 1. // 还原 2. Array.prototype.findIndex = function(handler) { 3. // 循环执行函数 并把成员和索引传递进去 4. for (var i = 0; i < this.length; i++) { 5. // 获取函数的执行结果 6. var result = handler(this[i], i); 7. // 如果为true 就说明找到了 不需要再继续循环 8. if (result === true) { 9. // 返回当前成员的索引 10. return i 11. } 12. } 13. } |

## 6.5 copyWithin

该方法用于数组的内部复制

arr.copyWithin(pos, start, end);

pos: 覆盖的开始位置

start: 复制的开始位置 包含

end: 复制的结束位置 不包含

返回值：arr自己

demo:

|  |
| --- |
| 1. // 定义数组 2. var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9]; 3. // 调用copyWithin方法 将 5 6 7 复制到 2 3 4的位置并覆盖 4. arr.copyWithin(1, 4, 7); |

输出数组：

|  |
| --- |
|  |

本次copyWithin方法的含义： 从数组的下标4（包含）截取到下标7（不包含） 再将截取到的3个值 从下标1的位置开始覆盖。会改变原数组。

还原1：

|  |
| --- |
| 1. // 实现copyWithin方法 2. Array.prototype.copyWithin = function(pos, start, end) { 3. // 先从start截取到end 4. var newArr = this.slice(start, end); 5. // 备份原来的长度 6. var length = this.length; 7. // 再从数组的pos位置开始覆盖 8. for (var i = 0; i < newArr.length; i++) { 9. // 从数组的pos位置开始覆盖 10. this[pos + i] = newArr[i]; 11. } 12. // 还原长度 13. this.length = length; 14. return this; 15. } |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |