爱创课堂前端培训

# ECMAScript 5.0

第2天课堂笔记（本课程共2天）

班级：北京前端训练营19期

讲师：彭帅伟

日期：2018年12月24日

彭帅伟老师

QQ: 284337908

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

目录

[ECMAScript 5.0 1](#_Toc3488)

[复习： 3](#_Toc21577)

[一、 严格模式 6](#_Toc24374)

[1.1 开启严格模式 6](#_Toc14822)

[1.2 在严格模式下定义变量 7](#_Toc11999)

[1.3 八进制 8](#_Toc6788)

[1.4 arguments.callee 9](#_Toc10966)

[1.5 delete 10](#_Toc10065)

[1.6 使用保留字定义变量 11](#_Toc29695)

[1.7 eval 11](#_Toc10307)

[1.8 with 12](#_Toc23164)

[1.9 全局函数作用域 13](#_Toc15605)

[1.10 同名属性 14](#_Toc14222)

[1.11 函数参数 15](#_Toc31084)

[二、 对象的拓展 16](#_Toc6084)

[2.1 特性 16](#_Toc26129)

[2.2 Object.defineProperty 16](#_Toc16283)

[下午复习： 18](#_Toc25693)

[2.3 特性--配置值 19](#_Toc9702)

[2.4 特性--可修改性 19](#_Toc17934)

[2.5 特性--可枚举性 20](#_Toc6847)

[2.6 特性--可配置性 20](#_Toc4830)

[2.7 特性方法 21](#_Toc18138)

[2.8 设置多个属性特性 22](#_Toc6416)

[三、 原型方法 23](#_Toc20063)

[3.1 isPrototypeOf 23](#_Toc1875)

[3.2 getPrototypeOf 24](#_Toc31680)

[3.3 setPrototypeOf 24](#_Toc715)

[四、 对象拓展 24](#_Toc14891)

[4.1 preventExtensions 24](#_Toc567)

[4.2 seal 25](#_Toc9342)

[4.3 freeze 25](#_Toc16280)

[4.4 对象创建的新方式 26](#_Toc21738)

# 复习：

JSON拓展：

ES5为了json字符串与js对象互转方便提供了JSON内置对象，分别提供了两个方法：

parse:

作用：将json字符串解析为js对象的

使用方式：

JSON.parse(str, fn)

str: 要处理的字符串

fn: 要处理的函数

stringify:

作用： 将js对象转为json字符串的

使用方式：

JSON.stringify(obj, fn)

obj: 要处理的对象

fn: 要执行的函数

数组的拓展：

判断数组：

第一种方式判断对象的类型

Object.prototype.toString.call(obj)

第二种方式判断是否是实例化对象

obj instanceof Array

第三种方式判断构造函数是否是Array

obj.constructor === Array

第四种方式数组的静态方法

Array.isArray(obj)

获取成员的索引：

ES5为数组提供了两个方法用于获取成员的索引值:

indexOf(): 从前向后开始查找

lastIndexOf(): 从后向前开始查找

参数就是要查找的成员，如果找到该成员就返回该成员的索引，反之返回-1

forEach():

数组的迭代器方法，用来代替for循环，并没有移除for循环，只是将for循环封装在了forEach方法的内部

接受一个参数就是执行的函数

有三个参数：成员值、索引值、原数组

作用域是window

返回值对forEach执行的结果的是没有影响的

forEach方法的返回值始终是undefined

map：

使用方式和forEach类似， 只是map方法的返回值是有意义的

参数：执行的函数

有三个参数： 成员值、索引值、原数组

返回值新的数组成员

map方法的返回值是一个新数组

fill:

用于填充数组

参数就是要填充的成员

断言方法：

some:

判断数组中是否有成员满足条件

接受一个参数就是要执行的函数

有三个参数：成员值、索引值、原数组

返回值就是判断的依据

some方法的返回值:

true: 至少有一个成员满足条件

false: 都不满足条件

every:

判断数组中是否都满足条件

接受一个参数：函数

三个参数：成员值、索引值、原数组

返回值就是判断的依据

every方法的返回值：

true: 都满足条件

false: 至少有一个是不满足条件的

filter:

用于过滤数组

接受一个参数： 函数

三个参数：成员值、 索引值、原数组

返回值就是过滤的条件

filter方法的返回值：满足过滤条件的成员组成新数组

累加方法：

reduce

reduceRight:

这两个是累加方法， reduce是从前向后开始遍历， reduceRight是从后向前开始遍历

会对每个成员逐一处理，并将结果返回

参数：函数

有四个参数： 上一次的累积结果， 当前成员值， 当前索引值， 原数组

返回值当次累积的结果， 将会在下一次遍历的时候作为第一个参数传递

reduce是从第二个成员开始遍历， 第一个成员在第一次遍历的时候作为第一个参数传递

reduceRight是从倒数第二个成员开始遍历，倒数第一个成员会在第一次遍历的时候作为第一个参数传递

addNum(num1, num2)

首先确定最大值和最小值

创建数组确定数组的长度

填充数组

用map方法构建一个从最小值到最大值之间的数组（用最小值加上索引值）

调用reduce或者是redecueRight方法即可

bind方法：

为函数绑定作用域

bind方法与call方法类似， 第一个参数就是要改变的this指向，从第二个参数开始都是原函数所需要的参数

bind方法与call | apply方法区别

call | apply 调用即执行

bind调用不会执行 返回的是一个新的方法

日期拓展：

var date = new Date()

console.log(date.toJSON()) (转为标准化格式)

# 严格模式

ES5拓展了严格模式，可以使我们的代码更加的安全可靠

在以前我们书写的代码都是处于非严格模式（正常模式）

在正常模式下， 定义一个变量，可以不使用var, 在严格模式下是不允许的

在正常模式下， 定义数值的时候，可以使用八进制， 严格模式下是不允许

在正常模式下， 可以使用arguments.callee， 在严格模式下是不允许

## 开启严格模式

使用方式： 加入“use strict”字符串

开启严格模式之后，认识它的浏览器会自动开启严格模式，如果不认识它的浏览器，只是会当做一个字符串定义出来，不会影响后续代码

当开启严格模式之后， 就必须按照严格模式的规范来书写代码

在js第一行加入“use strict”此时，将处于“全局严格模式”

如果在某个函数中的第一行加入“use strict”， 当函数执行的时候，此时处于“局部严格模式”

举例：

正常模式下：

|  |
| --- |
| 1. // 定义变量 2. var a = 10; 3. // 不使用var 定义变量 4. b = 20; |

输出：

|  |
| --- |
|  |

严格模式下：

|  |
| --- |
| 1. // 开启严格模式 在开启严格模式之前的代码仍处于正常模式 2. "use strict" 3. // 定义变量 4. var a = 10; 5. // 不使用var 定义变量 6. b = 20; |

局部严格模式：

正常模式下：

|  |
| --- |
| 1. // 定义函数 2. function fun() { 3. // 定义变量 4. var a = 10; 5. // 不使用var 定义变量 6. b = 20; 7. } 8. fun(); |

输出：

|  |
| --- |
|  |

开启严格模式：

|  |
| --- |
| 1. // 定义函数 2. function fun() { 3. // 开启严格模式 4. "use strict" 5. // 定义变量 6. var a = 10; 7. // 不使用var 定义变量 8. b = 20; 9. } 10. fun(); |

结果：

|  |
| --- |
|  |

## 在严格模式下定义变量

在严格模式定义变量，必须使用var 关键字

允许：

|  |
| --- |
| 1. var a = 10; |

不允许：

|  |
| --- |
| 1. b = 20; |

正常模式下：

|  |
| --- |
| 1. // 定义变量 使用var 2. var a = 10; 3. // 不使用var 4. b = 20; |

输出：

|  |
| --- |
|  |

在严格模式下：

|  |
| --- |
|  |

## 八进制

在js中有三种： 八进制、 十进制、 十六进制

默认是十进制

以0开头的是八进制

以0x、 ox开头是十六进制

允许：

|  |
| --- |
| 1. var num = 15; 2. var num1 = 0xff; |

不允许：

|  |
| --- |
| 1. var num2 = 011; |

举例：

正常模式下：

|  |
| --- |
| 1. // 定义数值 2. var num = 15; 3. var num1 = 0xff; 4. var num2 = 011; |

输出：

|  |
| --- |
|  |

严格模式下：

|  |
| --- |
| 1. // 开启严格模式 2. "use strict" 3. // 定义数值 4. var num = 15; 5. var num1 = 0xff; 6. var num2 = 011; |

结果：

|  |
| --- |
|  |

## arguments.callee

代码写的越“具体”， 越能提升性能

而arguments.callee在编译阶段是无法被解析的， 只有执行到的时候才能确定下来，所以也就无从提升性能

举例：

正常模式下：

|  |
| --- |
| 1. // 安全类 2. function People(name, age, sex) { 3. // 判断this指向 4. if (this instanceof arguments.callee) { 5. // 说明是使用new 来调用 6. this.name = name; 7. this.age = age; 8. this.sex = sex; 9. } else { 10. // 说明没有使用new 来调用 11. return new arguments.callee(name, age, sex); 12. } 13. } |

结果：

|  |
| --- |
|  |

在严格模式下：

|  |
| --- |
| 1. // 开启严格模式 2. "use strict" 3. // 安全类 4. function People(name, age, sex) { 5. // 判断this指向 6. if (this instanceof arguments.callee) { 7. // 说明是使用new 来调用 8. this.name = name; 9. this.age = age; 10. this.sex = sex; 11. } else { 12. // 说明没有使用new 来调用 13. return new arguments.callee(name, age, sex); 14. } 15. } |

结果：

|  |
| --- |
|  |

## delete

delete用于删除对象中的属性

如果使用delete删除一个变量，在正常模式下，可以执行， 但是不会删除成功， 在严格模式下会报错

举例：

在正常模式下：

|  |
| --- |
| 1. // 定义全局变量 2. var a = 10; 3. window.b = 20; 4. // 使用delete 删除变量 5. console.log(delete a); 6. console.log(delete b); |

输出：

|  |
| --- |
|  |

在严格模式下:

|  |
| --- |
| 1. // 开启严格模式 2. "use strict" 3. // 定义全局变量 4. var a = 10; 5. window.b = 20; 6. // 使用delete 删除变量 7. console.log(delete a); 8. console.log(delete b); |

结果：

|  |
| --- |
|  |

## 使用保留字定义变量

在正常模式下可以使用保留字定义变量， 在严格模式下是不允许的

举例：

正常模式下：

|  |
| --- |
| 1. // 使用保留字 2. var public = "hehe"; 3. console.log(public); |

结果：

|  |
| --- |
|  |

严格模式下：

|  |
| --- |
| 1. // 开启严格模式 2. "use strict" 3. // 使用保留字 4. var public = "hehe"; 5. console.log(public); |

结果：

|  |
| --- |
|  |

## eval

eval是一个函数，可以把字符串当做代码来执行， 但是会污染全局作用域

在严格模式下，是不允许使用eval，避免了对全局作用域的污染

举例：

正常模式下：

|  |
| --- |
| 1. // 定义变量 2. var strcode = "var a = 10"; 3. // eval接受一个参数，执行的结果就是字符串代码 4. eval(strcode); 5. // 当eval执行之后， a真的被定义出来了 |

输出：

|  |
| --- |
|  |

在严格模式下：

|  |
| --- |
| 1. // 开启严格模式 2. "use strict" 3. // 定义变量 4. var strcode = "var a = 10"; 5. // eval接受一个参数，执行的结果就是字符串代码 6. eval(strcode); |

结果：

|  |
| --- |
|  |

## with

with是一个语法， 也是会污染全局作用域

with (target) {

console.log(a) // 如果target中存在a，那么会优先输出target中的a , 如果没有回沿着词法作用域一直往上找

}

举例：

在正常模式下:

|  |
| --- |
| 1. // 定义对象 2. var obj = { 3. a: 1, 4. b: 2 5. } 6. var a = 10; 7. var b = 20; 8. with (obj) { 9. // 因为obj中存在a和b， 所以会优先输出obj中的a和b 10. console.log(a); 11. console.log(b); 12. // 因为Obj中存在a属性， 所以优先修改的是obj中a 13. a = 11; 14. // 因为Obj中没有c， 所以就定义在全局中 15. c = 123; 16. } |

结果：

|  |
| --- |
|  |

在严格模式下：

|  |
| --- |
| 1. // 开启严格模式 2. "use strict" 3. // 定义对象 4. var obj = { 5. a: 1, 6. b: 2 7. } 8. var a = 10; 9. var b = 20; 10. with (obj) { 11. // 因为obj中存在a和b， 所以会优先输出obj中的a和b 12. console.log(a); 13. console.log(b); 14. // 因为Obj中存在a属性， 所以优先修改的是obj中a 15. a = 11; 16. // 因为Obj中没有c， 所以就定义在全局中 17. c = 123; 18. } |

结果：

|  |
| --- |
|  |

## 全局函数作用域

在正常模式下全局函数的作用域是window, 会污染全局作用域，

在严格模式下，全局函数作用域是undefined, 从而避免了对全局作用域的污染  
举例：

正常模式下：

|  |
| --- |
| 1. // 定义函数 2. function fun() { 3. var a = 10; 4. console.log(a); 5. // 输出this 6. console.log(this); 7. // 通过this添加属性 8. this.color = "red"; 9. } 10. fun(); |

输出：

|  |
| --- |
|  |

严格模式下：

|  |
| --- |
| 1. // 开启严格模式 2. "use strict" 3. // 定义函数 4. function fun() { 5. var a = 10; 6. console.log(a); 7. // 输出this 8. console.log(this); 9. // 通过this添加属性 10. this.color = "red"; 11. } 12. fun(); |

结果：

|  |
| --- |
|  |

## 同名属性

在正常模式下， 定义对象中的属性名是可以同名， 后面的会覆盖前面的

在严格模式下， 是不允许使用， 但是浏览器检测不出来

举例：

正常模式下：

|  |
| --- |
| 1. // 定义对象 2. var obj = { 3. a: 1, 4. a: 2 5. } 6. console.log(obj.a); |

输出：

|  |
| --- |
|  |

严格模式下：

|  |
| --- |
| 1. // 开启严格模式 2. "use strict" 3. // 定义对象 4. var obj = { 5. a: 1, 6. a: 2 7. } 8. console.log(obj.a); |

结果：

|  |
| --- |
|  |

注意： 结果虽然一样， 并不是没有开启严格模式， 只是浏览器没有实现

## 函数参数

在正常模式下，函数参数是可以同名的， 后面会覆盖前面的

在严格模式下，是不允许的

举例：

正常模式下：

|  |
| --- |
| 1. // 定义函数 2. function fun(color, color) { 3. console.log(color); 4. } 5. fun("red", "blue"); |

输出：

|  |
| --- |
|  |

严格模式下：

|  |
| --- |
| 1. // 开启严格模式 2. "use strict" 3. // 定义函数 4. function fun(color, color) { 5. console.log(color); 6. } 7. fun("red", "blue"); |

结果：

|  |
| --- |
|  |

# 对象的拓展

## 特性

对象是什么，我们属性来描述

属性是什么，我们用特性来描述

特性是用来描述属性的

## Object.defineProperty

该方法用于设置单一属性特性

使用方式：

Object.defineProperty(obj, property, options)

obj: 要设置特性的对象

property: 设置的属性

options: 描述特性对象

举例：

在没有设置特性之前：

|  |
| --- |
| 1. // ES5 2. var obj = { 3. num: 100, 4. color: "red" 5. } 6. // 使用for in循环 7. for (var i in obj) { 8. console.log(i); 9. } |

输出：

|  |
| --- |
|  |

在设置特性之后：

|  |
| --- |
| 1. // 设置特性 2. Object.defineProperty(obj, "color", { 3. // 修改可枚举性 4. enumerable: false 5. }) 6. // 再次使用for in 循环 7. for (var i in obj) { 8. console.log(i); 9. } |

结果：

|  |
| --- |
|  |

## 下午复习：

严格模式：

开启方式：

在js第一行加入“use strict”, 此时将处于“全局严格模式”

如果在某个函数中的第一行加入“use strict”， 当函数执行的时候，此时将处于“局部严格模式”

严格模式与正常模式之间的区别：

在严格模式下，定义变量必须使用var

在严格模式下，定义数值的时候， 不能使用八进制

在严格模式下，不能使用arguments.callee

在严格模式下，不能使用delete删除变量

在严格模式下，不能使用保留字定义变量

在严格模式下，不能使用eval函数

在严格模式下，不能使用with

在严格模式下，全局函数作用域指向undefined

在严格模式下，定义对象中的属性不能是同名的

在严格模式下，函数的参数不能相同

特性：

使用方式：

Object.defineProperty(obj, property, options)

## 特性--配置值

使用方式：

Object.defineProperty(obj, property, {

value: “” // 配置值

})

举例：

|  |
| --- |
| 1. // 在ES5 2. var obj = { 3. color: "green" 4. } 5. // 设置特性 6. Object.defineProperty(obj, "color", { 7. // 配置值 8. value: "red" 9. }) |

## 特性--可修改性

使用方式：

Object.defineProperty(obj, property, {

writable: false // 将可修改性改为false

})

举例：

|  |
| --- |
| 1. // ES5 2. var obj = { 3. color: "red" 4. } 5. // 设置特性 6. Object.defineProperty(obj, "color", { 7. // 不可写入 8. writable: false 9. }) |

执行代码：

|  |
| --- |
| 1. // 尝试修改obj.color 2. obj.color = "blue"; 3. console.log(obj); |

结果：

|  |
| --- |
|  |

## 特性--可枚举性

使用方式：

Object.defineProperty(obj, property, {

enumerable: false // 修改可枚举性为false

})

举例：

|  |
| --- |
| 1. // 定义对象 2. var obj = { 3. num: 100, 4. color: "red" 5. } 6. // 默认情况下是可以被枚举的 7. for (var i in obj) { 8. console.log(i, obj[i]); 9. } 10. // 设置特性 11. Object.defineProperty(obj, "color", { 12. // 不可枚举 13. // enumerable: false 14. enumerable: true 15. }) 16. // 再次枚举 17. for (var i in obj) { 18. console.log(i, obj[i]); 19. } |

在设置特性之前：

|  |
| --- |
|  |

在设置特性之后：

|  |
| --- |
|  |

## 特性--可配置性

使用方式：

Object.defineProperty(obj, property, {

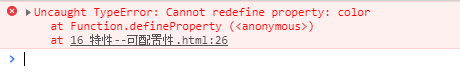
configurable: false // 将可配置性改为false

})

举例：

|  |
| --- |
| 1. // 定义对象 2. var obj = { 3. num: 100, 4. color: "red" 5. } 6. // 设置特性 7. Object.defineProperty(obj, "color", { 8. // 配置值 9. value: "blue", 10. // 不可写入 11. writable: false, 12. // 不可配置 13. configurable: false 14. }) 15. Object.defineProperty(obj, "color", { 16. // 配置值 17. value: "blue", 18. // 不可写入 19. writable: true 20. }) 21. // 尝试修改obj.color 22. obj.color = "green"; 23. console.log(obj); |

结果：



## 特性方法

get用来获取值的

没有参数

返回值就是获取的值

作用域的当前对象

千万不要在方法中，获取该属性值，否则递归死循环

通常我们会获取该属性值的备用属性

set用来设置值的

参数就是要设置的值

没有返回值

作用域是当前对象

千万不要在方法中，设置该属性值，否则会递归死循环

通常我们会设置该属性的一个备用属性

举例：

|  |
| --- |
| 1. // 定义对象 2. var obj = { 3. color: "red" 4. } 5. // 一旦设置了特性方法（get, set）， 当obj.color的时候，会触发get函数， 当obj.color = xx的时候， 会触发set函数 6. // 设置特性 7. Object.defineProperty(obj, "color", { 8. // 设置特性方法 9. // 取值器 10. get: function() { 11. console.log(111, this, arguments); 12. // return this.color; 13. // 通常我们获取该属性的一个备用属性 14. return this.\_color; 15. }, 16. // 赋值器 17. set: function(value) { 18. console.log(222, this, arguments); 19. // this.color = value; 20. // 通常我们设置该属性的备用属性值 21. this.\_color = value; 22. } 23. }) 24. obj.color = "green"; 25. console.log(obj.color); |

## 设置多个属性特性

使用方式：

Object.defineProperties(obj, options)

obj: 要设置的对象

options: 特性对象

key: 特性属性

value: 描述特性对象

可以设置特性属性以及特性方法

举例：

|  |
| --- |
| 1. // 定义对象 2. var obj = { 3. num: 100, 4. color: "red" 5. } 6. // 设置特性 7. Object.defineProperties(obj, { 8. color: { 9. // 配置值 10. value: "blue", 11. // 不可写入 12. writable: false, 13. // 不可枚举 14. enumerable: false, 15. // 不可配置 16. configurable: false 17. }, 18. // 设置特性方法 19. num: { 20. get: function() { 21. return this.\_num; 22. }, 23. set: function(value) { 24. this.\_num = value; 25. }, 26. // 不可枚举 27. enumerable: false, 28. // 一旦设置了特性方法（get, set）， 将无法再次配置writable, value 29. // 不可写入 30. // writable: false 31. // 配置值 32. value: "orange" 33. } 34. }) |

# 原型方法

ES5为原型拓展了几个方法

## 3.1 isPrototypeOf

是原型对象的方法

判断原型对象是否是参数对象的原型

在查找原型的时候， 会查找整个原型链

举例：

|  |
| --- |
| 1. // 获取类数组对象 2. var obj = document.getElementsByTagName("div"); 3. // 定义数组 4. var arr = []; 5. // Array的原型是哪个对象的原型 6. console.log(Array.prototype.isPrototypeOf(obj)); 7. console.log(Array.prototype.isPrototypeOf(arr)); 8. // 在查找原型的时候， 会查找整个原型链 9. console.log(Object.prototype.isPrototypeOf(obj)); 10. console.log(Object.prototype.isPrototypeOf(arr)); |

## 3.2 getPrototypeOf

用来获取原型对象的

是对象的静态方法

在之前，我们可以通过\_\_proto\_\_来获取原型对象， 在ES5中，以\_\_开头，是不推荐使用的， 所以提供了getPrototypeOf方法来获取原型对象

举例：

|  |
| --- |
| 1. // 获取原型对象 2. console.log(arr.\_\_proto\_\_); 3. console.log(Object.getPrototypeOf(arr)); 4. console.log(arr.\_\_proto\_\_ === Object.getPrototypeOf(arr)); |

## 3.3 setPrototypeOf

设置某个对象的原型对象

使用方式：

Object.setPrototypeOf(obj, prototype)

obj： 要设置的对象

prototype: 新的原型对象（可以是一个对象也可以是null）

举例：

|  |
| --- |
| 1. // 设置原型对象 2. // 会修改整个原型链 3. // Object.setPrototypeOf(arr, null); 4. Object.setPrototypeOf(arr, {"name": "王老王"}); 5. console.log(arr.\_\_proto\_\_); |

# 对象拓展

## preventExtensions

该方法用于取消对象的可拓展性

举例：

|  |
| --- |
| 1. // 定义对象 2. var obj = { 3. a: 1 4. } 5. // 默认情况下，对象是可以被拓展的 6. // obj.b = 2; 7. // 取消对象的可拓展性 8. Object.preventExtensions(obj); 9. // 尝试添加属性 10. obj.b = 2; 11. console.log(obj); |

结果：

|  |
| --- |
|  |

**总结：当取消了对象的可拓展性之后，不能再添加属性， 可以删除属性， 可以修改属性值**

**查看对象是否被取消可拓展性：Object.isExtensible(obj) 返回值就是bool**

**如果是true 说明对象可以被拓展， 反之不可以拓展**

## seal

用于封闭对象

举例：

|  |
| --- |
| 1. // 定义对象 2. var obj = { 3. a: 1 4. } 5. // 封闭对象 6. Object.seal(obj); 7. // 添加属性 8. obj.b = 2; 9. // 删除属性 10. delete obj.a; 11. // 修改属性值 12. obj.a = 123; 13. console.log(obj); |

**总结：当封闭了对象之后， 此时，不能可以添加属性， 也不可以删除， 但是可以修改属性值**

**查看对象是否被封闭：Object.isSealed(obj) 返回值是bool**

## freeze

用于冻结对象

举例：

|  |
| --- |
| 1. // 定义对象 2. var obj = { 3. a: 1 4. } 5. // 冻结对象 6. Object.freeze(obj); 7. // 添加属性 8. obj.b = 2; 9. // 删除属性 10. delete obj.a; 11. // 修改属性值 12. obj.a = 123; 13. console.log(obj); |

结果：

|  |
| --- |
|  |

**总结： 当冻结了对象之后， 不可以添加属性， 不可以删除属性， 不可以修改属性值**

**查看对象是否被冻结: Object.isFrozen(obj) 返回值bool**

## 对象创建的新方式

使用方式：

Object.create(prototype, options)

接收两个参数

第一个参数是一个对象 （可以是null）

是通过Object.create创建出来的原型对象

第二个参数是一个对象 （可以省略）

是通过Object.create创建出来的特性对象

举例：

|  |
| --- |
| 1. // 原型对象 2. var prototype = { 3. sayHello: function() { 4. console.log("sayHello"); 5. }, 6. sayNihao: function() { 7. console.log("nihao"); 8. }, 9. hehe: function() { 10. console.log("hehe"); 11. } 12. } 13. // 特性对象 14. var options = { 15. color: { 16. // 配置值 17. value: "red", 18. // 不可写入 19. writable: true, 20. enumerable: true 21. }, 22. num: { 23. // 配置值 24. value: 100, 25. // 不可写入 26. writable: true, 27. enumerable: true 28. } 29. } 30. var obj = Object.create(prototype, options); |

结果：

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |