爱创课堂前端培训

# JavaScript进阶

第5天课堂笔记（本课程共8天）

班级：北京前端训练营19期

讲师：李兰波

日期：2018年12月20日

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

# 复习

鼠标滚轮事件:

在IE中：onmousewheel

在其它浏览器中：onmouswheel

在火狐中：DOMMouseScroll

事件对象的属性：

onmousewheel事件中：wheelDelta决定鼠标的滚动方向和滚动速度 120的倍数 正值表示向上滚动 负值表示向下滚动

DOMMouseScroll事件中：detail决定鼠标的滚动方向和速度 3的倍数 正值表示向下滚动 负值表示向上滚动

键盘事件：

onkeydown： 当键盘键按下时触发的事件

onkeyup: 当键盘键抬起的时候触发的事件

onkeypress: 当输入的时候触发事件

事件对象属性：

e.keyCode 键盘的编码 常用的：32 空格 37 ← 38 ↑ 39 → 40 ↓

tabIndex：该属性是一个HTML标准属性。用于决定用户按下tab键时的切换焦点的顺序。

面向对象：是一种编程思想。它把原来的面向过程，进一步抽象，将要书写的目标抽象成一个对象，分析该对象拥有什么样的属性和方法。并定义成类。需要这样的对象的时候，就初始化一个类。

构造函数：面向对象中一个十分重要的概念。类在初始化的时候，要调用构造函数，得到对象。

定义构造函数： 与普通函数的定义一样。 唯一的区别是：首字母大写。

定义该构造函数的时候，如何定义函数体？

要书写成 this.xx = xx;

调用方式也发生了改变： new 构造函数(参数1, 参数2……);

# HTML标准属性

整个前端分三层： HTML、CSS、JS。

HTML是标签语言。它有开启标签与闭合标签。它有属性。

它的属性是W3C规定：

整个属性分成两种：

HTML标准属性：指的HTML标签自带的属性

通用属性：id、class、style这些每一个标签都有的属性，叫做通用属性。

元素特有属性：

img的width、height、src、alt

a的href

meta的content

input的type

自定义属性：现在的自定义属性，通常推荐以data-开头

# 原型

## 2.1 问题的出现

工厂函数，在初始化对象的时候，对象的方法，各自拥有一份。而这些方法的功能却相同。（也就是，完全没有必要每人一份）

demo:

|  |
| --- |
| 1. function factory(name, age, sex) { 2. var obj = { 3. name: name, 4. age: age, 5. sex: sex, 6. sayHello: function() { 7. console.log("大家好，我的名字是" + this.name + "我的年龄是" + this.age + "我的性别是" + this.sex); 8. } 9. } 10. return obj; 11. } 12. // 初始化一个实例化对象 13. var obj = factory("小明", 13, "男"); 14. // 调用方法 15. obj.sayHello(); 16. // 再初始化一个实例化对象 17. var obj1 = factory("小红", 14, "女"); 18. // 调用方法 19. obj1.sayHello(); 20. // 此时，两个对象的sayHello方法地址不同 所以比较时是false 21. console.log(obj.sayHello === obj1.sayHello); // false |

## 2.2 原因

之所以每一个实例都有一套自己的方法，原因是因为方法的定义是在工厂内部。所以，每执行一次，就定义一个新的函数交给该实例。

于是，我们将它提取出来：将方法的定义从工厂的内部提取到外部。

demo:

|  |
| --- |
| 1. // 接上个案例 2. // 定义一个工厂函数 3. function factory(name, age, sex) { 4. var obj = { 5. name: name, 6. age: age, 7. sex: sex, 8. sayHello: sayHello 9. } 10. return obj; 11. } 12. // 将函数的定义从内部提取到外部 13. function sayHello() { 14. console.log("大家好，我的名字是" + this.name + "我的年龄是" + this.age + "我的性别是" + this.sex); 15. } 16. // 初始化一个实例化对象 17. var obj = factory("小明", 13, "男"); 18. // 调用方法 19. obj.sayHello(); 20. // 再初始化一个实例化对象 21. var obj1 = factory("小红", 14, "女"); 22. // 调用方法 23. obj1.sayHello(); 24. // 此时，两个对象的sayHello方法地址相同 所以比较时是true 25. console.log(obj.sayHello === obj1.sayHello); |

## 2.3 当方法有多个时

2.2案例中解决了方法为一个时的问题。但是当方法有多个时，2.2的解决方案会如下：

|  |
| --- |
| 1. // 接上个案例 2. // 定义一个工厂函数 3. function factory(name, age, sex) { 4. var obj = { 5. name: name, 6. age: age, 7. sex: sex, 8. sayHello: sayHello, 9. say1: say1, 10. say2: say2, 11. say3: say3, 12. say4: say4, 13. say5: say5 14. } 15. return obj; 16. } 17. // 将函数的定义从内部提取到外部 18. function sayHello() { 19. console.log("大家好，我的名字是" + this.name + "我的年龄是" + this.age + "我的性别是" + this.sex); 20. } 21. function say1() {} 22. function say2() {} 23. function say3() {} 24. function say4() {} 25. function say5() {} |

结论：当方法为多个时，可以都提取出来，但是方法的数量太多，会造成全局的变量过多。

## 2.4 对象

变量太多了，于是就定义一个对象，让这个对象去装载这些函数。

demo:

|  |
| --- |
| 1. // function say1() {} 2. // function say2() {} 3. // function say3() {} 4. // function say4() {} 5. // function say5() {} 6. var prototype = { 7. sayHello: function() { 8. console.log("大家好，我的名字是" + this.name + "我的年龄是" + this.age + "我的性别是" + this.sex); 9. }, 10. say1: function() {}, 11. say2: function() {}, 12. say3: function() {}, 13. say4: function() {}, 14. say5: function() {}, 15. } |

## 2.5 原型

经过2.4的优化之后，已经解决了方法太多时，全局变量的问题。但是还是有一个全局变量，就是prototype自己。

这一个也不想要。所以，就将该对象挂载在函数身上。

|  |
| --- |
| 1. function factory(name, age, sex) { 2. var obj = { 3. name: name, 4. age: age, 5. sex: sex, 6. sayHello: factory.prototype.sayHello, 7. say1: factory.prototype.say1, 8. say2: factory.prototype.say2, 9. say3: factory.prototype.say3, 10. say4: factory.prototype.say4, 11. say5: factory.prototype.say5 12. } 13. return obj; 14. } 15. factory.prototype = { 16. sayHello: function() { 17. console.log("大家好，我的名字是" + this.name + "我的年龄是" + this.age + "我的性别是" + this.sex); 18. }, 19. say1: function() {}, 20. say2: function() {}, 21. say3: function() {}, 22. say4: function() {}, 23. say5: function() {}, 24. } |

问题1 ：多个实例的函数的地址不同 解决： 将函数的定义从工厂内抽取到工厂外。

问题2 ：多个方法时 会造成全局变量过多 解决：放到了一个对象中

问题3 ：这一个对象也不想要 解决：将该对象放到了factory身上

现在，什么问题也没有了。

我们现在，称呼这个prototype对象，叫做“原型”。

# 构造函数的原型

每一个函数都有prototype属性。其中，普通函数的该属性，没有任何作用。构造函数的该属性，作用很大。

作用：让每一个实例共享同一套方法。

## 3.1 prototype属性

|  |
| --- |
| 1. function People(name, age, sex) { 2. this.name = name; 3. this.age = age; 4. this.sex = sex; 5. } |

输出People.prototype:

|  |
| --- |
|  |

该属性天生指向构造函数People

## 3.2 prototype作用

prototype就是用来盛放所有的构造函数的实例的方法的。

|  |
| --- |
| 1. // 该prototype属性上的所有内容都可以被构造函数的实例去调用。 2. People.prototype.sayHello = function() { 3. console.log("大家好,我是" + this.name) 4. } 5. // 初始化 6. var p = new People("小明", 13, "男"); 7. p.sayHello(); 8. var p1 = new People("小明1", 13, "男"); 9. p1.sayHello(); |

输出：

|  |
| --- |
|  |

地址：

|  |
| --- |
|  |

**注：以后在定义构造函数的时候，属性，定义在函数体内。方法，定义在原型上。**

# instanceof

问题：如何判定一个对象，是否是某一个构造函数的实例？

## 4.1 判定构造函数

第一种：通过对象调用constructor 得到构造函数 再与指定的构造函数比较

obj.constructor === 构造函数

优点：可行

缺点：当对象无法访问到constructor时，将不可使用

第二种：instanceof

|  |
| --- |
| 1. function People() { 2. } 3. People.prototype.constructor = null; 4. var p1 = new People(); 5. // console.log(p1.constructor === People); 6. // instanceof 是一个关键字 用于判定一个对象是否是另一个构造函数的实例。 7. console.log(p1 instanceof People, 11111) |

使用规范： 对象 instanceof 构造函数

# 继承

JS中的继承，不是指对象与对象之间的属性的继承、方法的继承。

而是类与类之间的事情。

子类继承了父类的方法和属性。这叫做“继承”。

## 5.1 构造函数式继承

|  |
| --- |
| 1. // 继承: 类与类之间的方法和属性的继承 2. // 定义父类 3. function People(name, age, sex) { 4. this.name = name; 5. this.age = age; 6. this.sex = sex; 7. } 8. // 定义子类 9. function Student(name, age, sex, grade) { 10. // 让Student继承People的属性 11. // 属性的继承 叫做构造函数式继承 12. People.apply(this, arguments); 13. this.grade = grade; 14. } 15. // 初始化一个人 16. var p1 = new People("小明", 12, "男"); 17. console.log(p1); 18. // 初始化一个学生 19. var s1 = new Student("小红", 13, "女", 6); 20. console.log(s1) |

## 5.2 类式继承

类式继承：继承方法。

|  |
| --- |
| 1. 思路1: 既然是继承方法 那么我直接将子类的原型指向父类的原型就行了 |

此时，子类的实例在调用自身原型上的方法的时候，其实访问的是父类的原型。可以调用方法。

但是，当子类添加独属于子类自身的方法的时候，会修改到父类的原型，就会影响到父类的实例。

所以，这种方法不完善。

|  |
| --- |
| 1. 思路2: 直接指向父类的原型不可取，那么还有谁能够访问到父类的原型，同时又是一个对象，同时修改该对象的时候，不会影响到父类的原型上的内容？ |

答：父类的实例。

所以，我们现在不直接将子类的原型指向父类的原型，而是将子类的原型指向父类的实例。

|  |
| --- |
|  |

此时，子类的实例，在调用方法的时候，会先查找自身是否具备该方法，结果发现没有，于是去查找自身的构造函数的原型，发现也没有。注意，查找子类的原型，就等于父类的实例在查找自身，就找到父类的原型上去了。找到了。就能够调用。

最终代码：

|  |
| --- |
| 1. // 继承: 类与类之间的方法和属性的继承 2. // 定义类 3. function People(name, age, sex) { 4. this.name = name; 5. this.age = age; 6. this.sex = sex; 7. } 8. // 添加方法 9. People.prototype.sayHello = function() { 10. console.log("我的名字是" + this.name); 11. } 12. // 定义另外一个类 13. function Student(name, age, sex, grade) { 14. this.name = name; 15. this.age = age; 16. this.sex = sex; 17. this.grade = grade; 18. } 19. // 将子类的原型指向父类的实例 20. Student.prototype = new People(); 21. // 因为缺失了constructor属性 所以要补回来 22. Student.prototype.constructor = Student; 23. // 给子类添加特有方法 24. Student.prototype.hi = function() { 25. console.log("hi"); 26. } |

## 5.3 组合式继承

构造函数式继承 加 类式继承就是组合式继承。

优点：既能够继承属性又能够继承方法

缺点：执行了两次父类构造函数 继承完毕后 原型上会有无用的属性

注意：

一定要将构造函数式继承代码放在第一条之后再给子类添加特有属性

一定要在类式继承完毕之后补回constructor属性

一定要在类式继承完毕之后增加新的方法

## 5.4 寄生式继承

|  |
| --- |
| 1. function jisheng(father, child, obj) { 2. // 定义一个空函数 3. var F = function() { 4. } 5. // 将该函数的原型指向父类的原型 6. F.prototype = father.prototype; 7. // 继承 8. child.prototype = new F(); 9. child.prototype.constructor = child; 10. for (var i in obj) { 11. child.prototype[i] = obj[i]; 12. } 13. return; 14. } |

解决了无用属性的问题。

寄生式继承加构造函数式继承，就是寄生组合式继承。

# 安全类

这是一种设计模式的体现。

它解决了别人在调用构造函数的时候，不使用new的问题。

解决方案：通过区分new与不new的区别 决定代码的书写

|  |
| --- |
| 1. // 定义一个构造函数 2. function People(name, age, sex) { 3. // 区分外界的调用方式 4. // 如果new 那么this是People的实例 5. if (this instanceof People) { 6. this.name = name; 7. this.age = age; 8. this.sex = sex; 9. } else { 10. // 如果不new 那么this就必定不是People的实例 11. // 强行返回一个People的实例 12. return new People(name, age, sex); 13. } 14. } 15. // 程序员1： 规规矩矩的调用 16. var p1 = new People("王老五", 22, "男"); 17. // 程序员2： 不使用new调用 18. var p2 = People("王老四", 21, "男"); |

输出：p1和p2都是People的实例

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |