# JavaScript进阶

第5天课堂笔记

# 目录

[JavaScript进阶 1](#_Toc16665)

[目录 2](#_Toc7486)

[一、复习 3](#_Toc31124)

[二、继承 6](#_Toc11503)

[三、对象与属性 9](#_Toc10685)

[3.1 in运算符 9](#_Toc24295)

[3.2 hasOwnProperty方法 9](#_Toc11289)

[3.3 对象直接打点验证某个属性是否存在 10](#_Toc28705)

[四、instanceof运算符 11](#_Toc3103)

# 一、复习

如何理解下面的代码：

|  |
| --- |
| 1. function Dog(name,age){ 2. this.name = name; 3. this.age = age; 4. } 5. var xiaobai = new Dog("小白",12); |

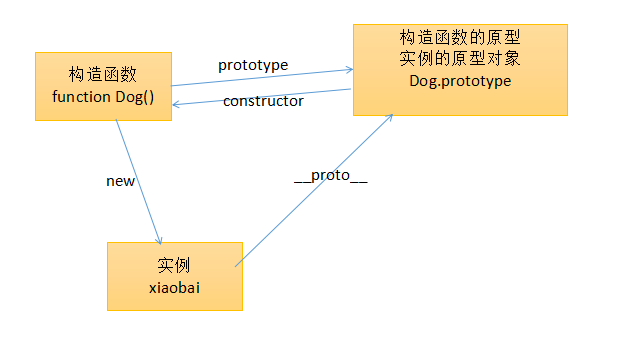
实际上面向对象开发经验的同学，比如学过Java、C++、C#等等的同学，就会思考成Dog是一个类，new操作就是实例化的操作。把类实例化出一个xiaobai实例。

这种思考模式，有点高看JavaScript了。在JavaScript中，没有强的“类”的概念。所以我们应该弱看这个东西：

Dog就是一个函数，new只不过是调用它的一种方式而已。就如同函数有很多方法调用，比如()直接调用，比如事件监听调用，比如定时器调用，new无非就是新的调用方式而已。上面说的所有调用方式，函数内部的this不一样。甚至可以极端的说：函数的调用方式不同，体现在了this的值不同。

new的时候，就是会调用函数，并且会创建一个空对象，然后执行语句，this指的是创建出来的新对象。构造函数里面的语句执行完毕之后，一个对象会返回。

返回的对象，可以叫做类的实例。你会发现，这个实例的原型对象，就是构造函数的原型。



原型链的查找：如果我们现在

|  |
| --- |
| 1. xiaobai.name |

首先会在xiaobai实例身上去寻找这个属性，如果有这个属性，就返回值。

如果自己身上没有，去找原型对象、原型对象的原型对象……Object.prototype。

所谓的原型链，就是\_\_proto\_\_的链条。

我们有两种写法去往构造函数的原型身上加方法：

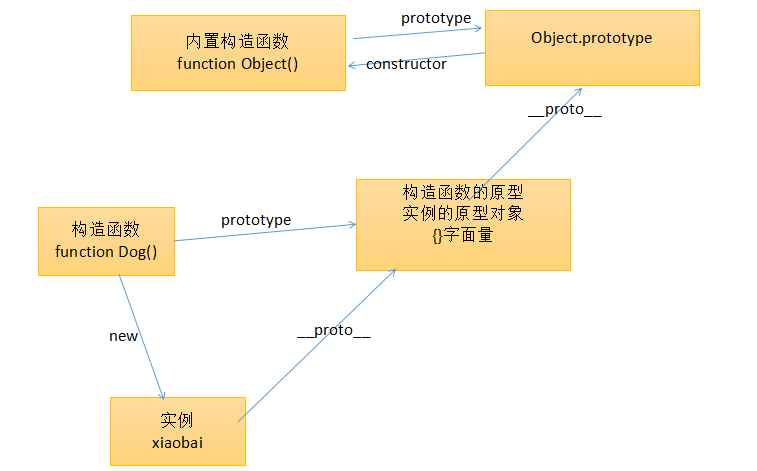
第一种，追加属性：

|  |
| --- |
| 1. Dog.prototype.sayHello = function(){ 2. alert(this.name); 3. } |

第二种，简单粗暴直接覆盖整个原型：

|  |
| --- |
| 1. Dog.prototype = { 2. sayHello : function(){ 3. alert(this.name + this.age); 4. } 5. } |

第二种的方法，有隐患，就是你的构造函数的原型(Dog.prototype)与构造函数(Dog)失联了。无法通过Dog.prototype.constructor访问构造函数Dog了。这是因为对象字面量的constructor一定是Object



简单粗暴法就是有这个弊端，如何解决呢？如果你觉得constructor真的这么重要，那么就补一个constructor属性就行了：

|  |
| --- |
| 1. Dog.prototype = { 2. constructor : Dog, 3. sayHello : function(){ 4. alert(this.name + this.age); 5. } 6. } |

constructor经常弄丢，不重要。比如在继承的实现的时候，小学生的实例的construcor居然是人类的构造函数。

C++是非常经典的纯面向对象语言，这个语言中都是对象，比如123、”String”都是对象。Java里面123也是数字，不是对象。

JavaScript本身就是函数式的，只不过有new关键字，有构造函数的特性。所以，我们昨天学习的面向对象的编程风格，在JS中只是用于组织代码用的。不要学习了面向对象之后，连轮播图都不会写了。

|  |
| --- |
| 1. 函数风格： 2. animate(obj,{"left":400},1000); 3. 对象风格： 4. $obj.animate({"left":400},1000); |

函数风格不是错误，该怎么写还是怎么写，有弊端。页面上如果有多个这个效果，比如有3个传统轮播。代码不好维护。

看看面向对象能解决什么问题？

让一个气球运动，我们发现普通的编程思维轻松解决：

|  |
| --- |
| 1. var top = 600; 2. setInterval(function(){ 3. top -= 2; 4. oDiv.style.top = top + "px"; 5. },30); |

让三个气球运动，怎么办？

|  |
| --- |
| 1. var top1 = 600; 2. var top2 = 600; 3. var top3 = 600; 4. setInterval(function(){ 5. top1 -= 2; 6. top2 -= 4; 7. top3 -= 6; 8. if(top1 < 0){ 9. } 10. if(top2 < 0){ 11. } 12. if(top3 < 0){ 13. } 14. oDiv.style.top = top + "px"; 15. },30); |

这种编程风格，就是“保姆式”、“大管家式”的，什么都要操心。每30毫秒，都要看看自己的每个气球有没有小于0，如果小于0让它爆照。如果大儿子饿了，就要给大儿子吃饭；然后看看二儿子有没有饿了，然后看看三儿子有没有饿了……。函数编程思维，注重的是情况的检测，在这种情况下每个对象要干什么。

面向对象：

面向对象说白了，就是让每个对象有智商，教会他们自己判断自己的处境，在这个处境要干什么事儿。就不用大管家一个个操心了。儿子饿了自己吃饭，儿子要上厕所，自己去上厕所了。

|  |
| --- |
| 1. function Ballon(){ 2. this.top = 600; 3. } 4. Ballon.prototype.fly = function(){ 5. var self = this; 6. this.timer = setInterval(function(){ 7. self.top -= 2; 8. if(self.top < 0){ 9. self.bomb(); 10. } 11. },30); 12. } |

面向对象的思维，注重的是每个对象自己的能力。

# 二、继承

“人”类 ， “小学生”类。 小学生也是人，只不过“丰富了”人。

“人”类有的属性，“小学生”都有，比如姓名、年龄、性别。

“人”类有的方法，“小学生”都有，比如打招呼、吃饭、睡觉。

除此之外，

“小学生”还丰富了一些属性：学号、班级、小红花个数、是否少先队员

“小学生”还丰富了一些方法：学习、lol、上课

小学生类，继承了人类。一说继承，千万不要想到人生的遗产继承，感觉小学生比人类少，是人类的局部。千万不要这么想。**在计算机世界中，继承是“丰富”，是比原来的类要多东西**。

小学生继承于人类， 小学生更丰富。

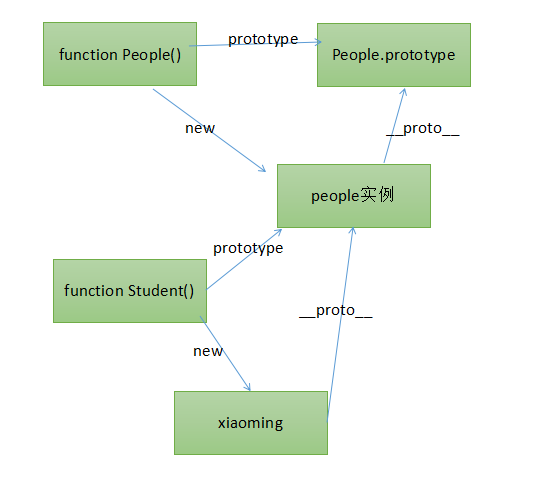
说白了，People、Student两个类，Student类的实例，拥有 People的全部属性属性、方法。

现在，你就应该朦朦胧胧的感觉到，用原型链来实现。JavaScript中没有extends。

People叫做父类、超类； Student叫做子类。

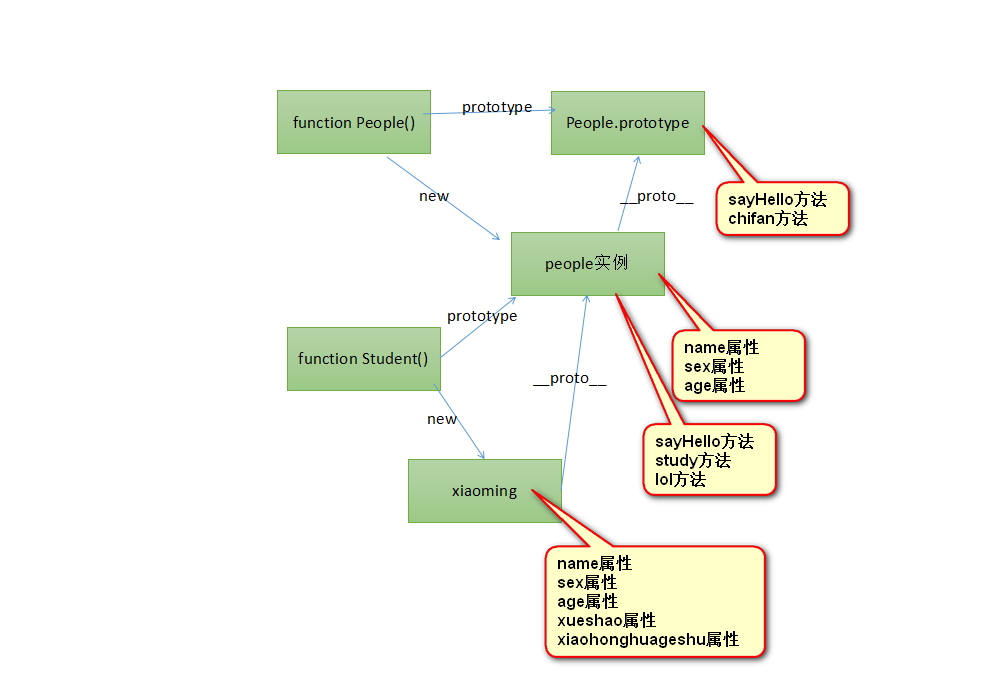
JS中实现继承的口号：

**子类的prototype等于父亲的某个实例**



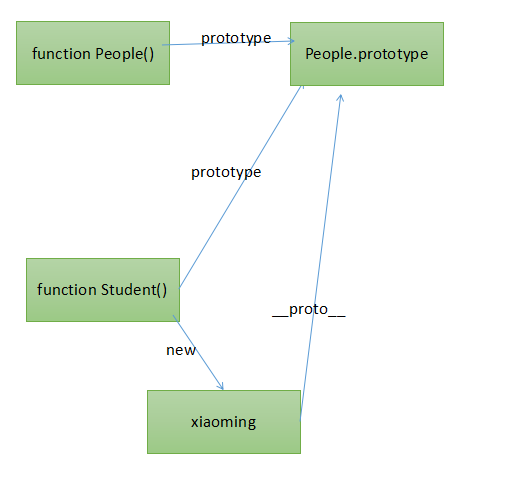
来看一个具体案例：

|  |
| --- |
| 1. function People(name,age,sex){ 2. this.name = name; 3. this.age = age; 4. this.sex = sex; 5. } 6. People.prototype.sayHello = function(){ 7. alert("你好我是人" + this.name); 8. } 9. People.prototype.chifan = function(){ 10. alert("我吃饭了！mia mia"); 11. } 12. function Student(name,age,sex,xuehao,xiaohonghuageshu){ 13. this.name = name; 14. this.age = age; 15. this.sex = sex; 16. this.xuehao = xuehao; 17. this.xiaohonghuageshu = xiaohonghuageshu; 18. } 19. **//核心语句，继承的实现全靠这条语句了：** 20. **Student.prototype = new People();** 22. Student.prototype.sayHello = function(){ 23. alert("你好我是小学生，我的学号是" + this.xuehao); 24. } 25. Student.prototype.study = function(){ 26. alert("好好学习，天天向上"); 27. } 28. Student.prototype.lol = function(){ 29. alert("玩儿英雄联盟啊！"); 30. } 31. var xiaohong = new Student("小红",11,"女",20160001,4); 32. //xiaohong.lol(); 33. //xiaohong.chifan(); 34. xiaohong.sayHello(); |



子类可以覆盖父类的一些方法，父类的方法不冲突，因为我们子类追加的方法，追加到了父类的实例上。

这种方法不行：追加子类方法的时候，父类也被改了：



# 三、对象与属性

## 3.1 in运算符

返回一个布尔值，表示这个属性是不是对象的属性。

|  |
| --- |
| 1. var obj = { 2. a : 1, 3. b : 2, 4. c : false 5. } 6. console.log("a" in obj); //true 7. console.log("b" in obj); //true 8. console.log("c" in obj); //true 9. console.log("d" in obj); //false |

in不仅仅检测是对象自己有没有这个属性，如果原型链上有这个属性，那么也会返回true。整个原型链如果没有这个属性，就返回false。也就是说，in操作符会进行原型链查找。

for in 这个循环，会把原型链上所有的可枚举的属性列出来：

|  |
| --- |
| 1. for(var k in obj){ 2. console.log(k); 3. } |

什么是可枚举，系统默认的属性（比如constructor）都是不可枚举的。for in循环能够把自己添加的属性罗列出来，罗列的不仅仅是自己身上的属性，还有原型链上的所有属性。

## 3.2 hasOwnProperty方法

这个方法定义在了Object.prototype对象上面，所以任何一个Object都能够拥有这个方法。

这个方法返回true、false。表示自己是否拥有这个属性，不考虑原型链。就看自己身上有没有这个属性，不进行原型链查找。

|  |
| --- |
| 1. var obj = { 2. a : 1, 3. b : 2, 4. c : 3 5. } 6. obj.\_\_proto\_\_ = { 7. d : 4 8. } 9. console.log(obj.hasOwnProperty("a")); //t 10. console.log(obj.hasOwnProperty("b")); //t 11. console.log(obj.hasOwnProperty("c")); //t 12. console.log(obj.hasOwnProperty("d")); //f |

for……in考虑原型链，所以我们可以内嵌一个判断，把自己身上的属性输出：

|  |
| --- |
| 1. for(var k in obj){ 2. obj.hasOwnProperty(k) && console.log(k); 3. } |

## 3.3 对象直接打点验证某个属性是否存在

对象打点调用属性，我们之前的课程已经讲过，遍历原型链。所以就可以看出来属性是否在自己身上、原型链上。如果在，就返回值；如果不在就返回undefined.

|  |
| --- |
| 1. var obj = { 2. a : 1, 3. b : 2, 4. c : 3 5. } 6. obj.\_\_proto\_\_ = { 7. d : 4 8. } 9. console.log(obj.m); //undefined 10. console.log(obj.a); //1 11. console.log(obj.b); //2 12. console.log(obj.c); //3 13. console.log(obj.d); //4 |

有个误会，比如obj.m值就是undefined，那么obj.m还是返回undefined。不知道m属性存在不存在。

# 四、instanceof运算符

类在英语里面叫做class，实例在英语里面叫做instance。

instaceof运算符：

|  |
| --- |
| 1. A instaceof B |

验证A对象是不是B类的实例。

比如：

|  |
| --- |
| 1. //类，构造函数 2. function Dog(){ 3. } 4. //实例化一个对象 5. var d = new Dog(); 6. //验证d是不是Dog的一个实例 7. console.log(d instanceof Dog); |



这里要注意一个事儿：“HelloKitty是狗”：

|  |
| --- |
| 1. //类，构造函数 2. function Dog(){ 3. } 5. function Cat(){ 6. } 7. Cat.prototype = new Dog(); //继承 8. var hellokitty = new Cat(); //通过cat来实例一个 9. console.log(hellokitty.constructor); //Dog 10. console.log(hellokitty instanceof Cat); //true 11. console.log(hellokitty instanceof Dog); //true |

instanceof 运算符的机理： 遍访hellokitty这个对象的原型链上的每个原型对象，如果遍访到这个原型对象，是某个构造函数的prototype，那么就认为hellokitty是这个构造函数的实例，返回true。

一个数组用typeof检测的时候，返回object

|  |
| --- |
| 1. var arr = []; 2. console.log(typeof arr); |



正则表达式也是：

|  |
| --- |
| 1. var obj2 = /ABC/g; 2. console.log(typeof obj2); |



用instanceof运算符能够轻松解决数组的识别：

|  |
| --- |
| 1. var arr = []; 2. console.log(**arr instanceof Array**); |



ECMAScript5标准中新增了一个API验证数组：

|  |
| --- |
| 1. Array.isArray(arr) |



IE9开始兼容。

总结一下，A instanceof B， 不能证明A是new B()出来的，因为可能是继承。

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |