

javascript学习--2017-11-8：闭包

笔记本： javascript

创建时间： 2017/11/8 8:26

更新时间： 2017/11/17 20:31

作者： xiethan

URL： <https://zhidao.baidu.com/question/120507977.html>

困惑的问题：call与apply的区别

为什么存在严格模式

instanceof

闭包：函数对象可以通过作用域链相互关联起来，函数体内部的变量可以保存在函数作用域内

```
//=====闭包的测试
var scope="globe scope";
function checkscope(){
var scope="local scope";
function f(){
return scope;
}
return f();
}
console.log(checkscope());//local scope

function checkscope2(){
var scope="local scope";
function f2(){
return scope;
}
return f2;
}
console.log(checkscope2());//local scope
```

规则说明：；javascript函数的执行用到了作用域链，这个作用域链是函数定义时创立的嵌套函数f()定义在这个作用域里，其中的变量scope一定是局部变量，不管何时何地执行这个函数f()，这种绑定在执行f()是依然有效

```
//私有状态
var uniqueInteger=(function(){
var count=2;
return function(){return count++;};//++count --->3 4
})();
console.log(uniqueInteger);//function(){return count++;}
console.log(uniqueInteger());//demo1.html:31 2
console.log(uniqueInteger());//3
var uniqueInteger2=uniqueInteger;
console.log(uniqueInteger2());//4 引用赋值
```

定义了个立即调用的函数，函数的返回值赋值给变量uniqueinteger，嵌套的函数可以访问作用域内的变量，而且可以访问外部函数中定义的counter变量，其他的代码都无法访问到ta

```
//闭包的注意：（共享的变量）
function constfunc(v){
return function() {return v;};
}
//创建一个数组来存储常数函数
var funcs=[];
for(var i=0;i<10;i++){
funcs[i]=constfunc(i);
}
console.log(funcs[5]());//5

//比较不同，第二个例子
function constfuncs(){
var funcs=[];
for(var i=0;i<10;i++){
funcs[i]=function() {return i;};
}
return funcs;
}
```

```
var funcs2=constfuncs();
console.log(funcs2[5]());//10
console.log(constfuncs());//10个函数，共享函数的变量
```

```
▼ 0: function (x)
  arguments: null
  caller: null
  length: 1
  name: ""
  ▼ prototype: Object
    ▼ constructor: function (x)
      arguments: null
      caller: null
      length: 1
      name: ""
      ► prototype: Object
      ► __proto__: function ()
      ► <function scope>
      ► __proto__: Object
    ▼ __proto__: function ()
      ► apply: function apply()
        arguments: (...)
      ► get arguments: function ThrowTypeError()
      ► set arguments: function ThrowTypeError()
      ► bind: function bind()
      ► call: function call()
        caller: (...)
      ► get caller: function ThrowTypeError()
      ► set caller: function ThrowTypeError()
      ► constructor: function Function()
        length: 0
        name: ""
      ► toString: function toString()
      ► Symbol(Symbol.hasInstance): function [Symbol.hasInstance]()
      ► __proto__: Object
      ► <function scope>
    ▼ <function scope>
      ► Closure
      ► Global: Window
```

上面第二个例子，创建了10个闭包，并把它存储在一个数组中，这些闭包都是同一个函数调用定义的。因此他们可以共享变量

this是javascript的一个关键字，不是变量，每个函数调用都包含一个this，如果闭包在外部函数里是无法访问this的，除非外部函数将this存为一个变量

var self = this; //保存变量，以便嵌套的函数能够访问到它

绑定arguments的问题与其类似，它并不是一个关键字

var outerArguments =arguments;

函数属性、方法和构造函数

1.length属性

- argument.length 表示传入函数实参的个数
- 函数本身的length，是指形参个数的个数，即函数定义时的个数

2.call () 与apply () 方法

对象的方法，通过调用方法的形式来间接调用，第一个实参是要调用函数的母对象，它是调用上下文，在函数体内通过this来获得对它的引用

例子：要想对象o的方法来调用函数f()，可以使用call与apply

f.call(o);

f.apply(o);

相对于：

o.m=f;//将f存储在o中临时方法

```
o.m(); //调用它，不传入参数
delete o.m; //将临时方法删除
```

区别：传入参数时不一样

```
f.call(o,1,2);
f.apply(o,[1,2]);
```

3.bind () 方法

将函数绑定到某个对象，这个方法将返回一个新函数。调用新函数将会把原始的函数f()当作o的方法来调用。

```
//bind
function f(y){
  return this.x+y;
}
var o={x:1};
var g=f.bind(o);
console.log(g(2)); // => 3
```

```
//另一种实现方式(支持ecmascript3 没有这个方法)
function bind(f,o){
  if(f.bind) return f.bind(o);
  else return function(){
    return f.apply(o.arguments);
  }
}
```

但：ECMAScript5的bind除了将函数绑定至一个对象，还附带一些其它应用（除了第一个实参以外，传入bind () 的实参也会绑定到至this=柯里化）

```
//bind的其他应用
var sum=function(x,y){
  return x+y;
}
var succ=sum.bind(null,1); //this的值绑定到Null.
succ(2); // = 3

//2
function f(y,z){return this.x+y+z};
var g=f.bind({x:1},2);
g(3); // = 6
```

ECMAScript3实现以上的功能也是可以的。需将 其绑定到function.prototype.bind===P192

4.toString

和所有javascript对象一样，函数也有toString一样，ECMAScript规范规定这个方法返回一个字符串，这个字符串和函数声明语句的语法相关。此方法一般都是返回函数的源码，内置函数往往返回一个类似【native code】的字符串作为函数体

5.function () 构造函数方式

不管是通过函数定义语句还是函数直接量表达式，函数都要使用function关键字，但函数还可以通过function构造函数来定义

如：

```
var f=new function("x","y","return x*y");//最后一个是函数体
```

等于：var f=function(x,y) {return x*y;}

6.可调用的对象

“可调用一个对象”是一个对象，可以在函数调用表达式调用这个对象，所有的函数可以调用，但并非所有可调用的函数都是对象

如：客户端方法：window.alert document.getElementById().....

```
//检测一个对象是否是真正的函数对象
function isFuntion(x){
  return Object.prototype.toString.call(x)=="[object Function]";
}
```

函数式编程

使用函数处理数组：map() reduce() 求标准差

高阶函数：就是操作函数的函数

代码

```
<!doctype html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<script type="text/javascript">
//=====闭包的测试
//作用域链
var scope="globe scope";
function checkscope(){
var scope="local scope";
function f(){
return scope;
}
return f();
}
console.log(checkscope());

function checkscope2(){
var scope="local scope";
function f2(){
return scope;
}
return f2;
}
console.log(checkscope2());
//私有状态
var uniqueInteger=(function(){
var count=2;
return function(){return count++;} //++count --->3 4
})();
console.log(uniqueInteger); //function(){return count++;}
console.log(uniqueInteger()); //demo1.html:31 2
console.log(uniqueInteger()); //3
var uniqueInteger2=uniqueInteger;
console.log(uniqueInteger2()); //4 引用赋值

//闭包的注意：（共享的变量）
function constfunc(v){
return function() {return v;};
}
//创建一个数组来存储常数函数
var funcs=[];
for(var i=0;i<10;i++){
funcs[i]=constfunc(i);
}
console.log(funcs[5]()); //5

//比较不同
function constfuncs(){
var funcs=[];
for(var i=0;i<10;i++){
funcs[i]=function(x) {return i;};
}
return funcs;
}
var funcs2=constfuncs();
console.log(funcs2[5]()); //10
console.log(constfuncs()); //10个函数，共享函数的变量

//抛出异常： throw Error("this error is test example");

//bind
function f(y){
```

```

return this.x+y;
}
var o={x:1};
var g=f.bind(o);
console.log(g(2));//=>3
//另一种实现方式(支持ecmascript3 没有这个方法)
function bind(f,o){
if(f.bind) return f.bind(o);
else return function(){
return f.apply(o.arguments);
}
}

//bind的其他应用
var sum=function(x,y){
return x+y;
}
var succ=sum.bind(null,1);//this的值绑定到Null.
succ(2);//=3

//2
function f(y,z){return this.x+y+z};
var g=f.bind({x:1},2);
g(3);//=6

//检测一个对象是否是真正的函数对象
function isFuntion(x){
return Object.prototype.toString.call(x)=="[object Function]";
}
</script>
<title>无标题文档</title>
</head>
<body>
</body>
</html>

```

实际案例：二级菜单

```

<ul>
<li>111</li>
<li>222</li>
<li>333</li>
<ul>
<script>
var lis = document.getElementsByTagName("li");
for(var i = 0;i<lis.length;i++){
(function(i){//外部函数
lis[i].onclick = function(){//内部函数
alert(i);
}
})(i);
}
</script>

```

高晓松：

```

var name = "The Window";
var object = {
  name : "My Object",
  getNameFunc : function(){
// var name="hello";
return function(){
return name;
};
}
};
console.log(object.getNameFunc());

```

简单粗暴地理解js原型链--js面向对象编程

原型链理解起来有点绕了，网上资料也是很多，每次晚上睡不着的时候总喜欢在网上找点原型链和闭包的文章看，效果极好。

不要纠结于那一堆术语了，那除了让你脑筋拧成麻花，真的不能帮你什么。简单粗暴点看原型链吧，想点与代码无关的事，比如人、妖以及人妖。

1) 人是人他妈生的，妖是妖他妈生的。人和妖都是对象实例，而人他妈和妖他妈就是原型。原型也是对象，叫原型对象。



2) 人他妈和人他爸啪啪啪能生出一堆人宝宝、妖他妈和妖他爸啪啪啪能生出一堆妖宝宝，啪啪啪就是构造函数，俗称造人。



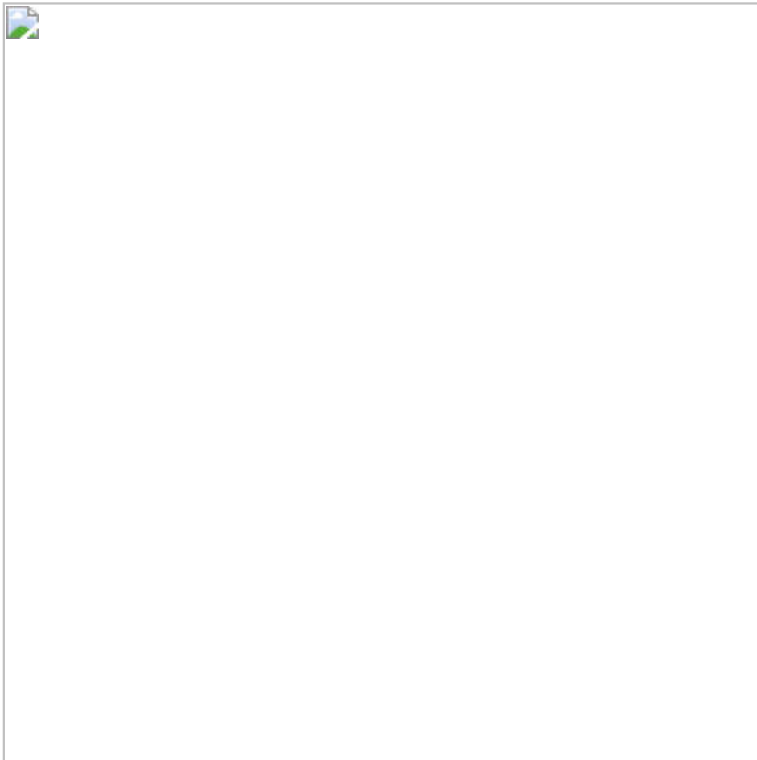
3) 人他妈会记录啪啪啪的信息，所以可以通过人他妈找到啪啪啪的信息，也就是说能通过原型对象找到构造函数。

4) 人他妈可以生很多宝宝，但这些宝宝只有一个妈妈，这就是原型的唯一性。

5) 人他妈也是由人他妈他妈生的，通过人他妈找到人他妈他妈，再通过人他妈他妈找到人他妈他妈.....，这个关系叫做原型链。

6) 原型链并不是无限的，当你通过人他妈一直往上找，最后发现你会发现人他妈他妈他妈.....的他妈都不是人，也就是原型链最终指向null。

7) 人他妈生的人会有人的样子，妖他妈生的妖会有妖的丑陋，这叫继承。



8) 你继承了你妈的肤色，你妈继承了你妈他妈的肤色，你妈他妈.....，这就是原型链的继承。

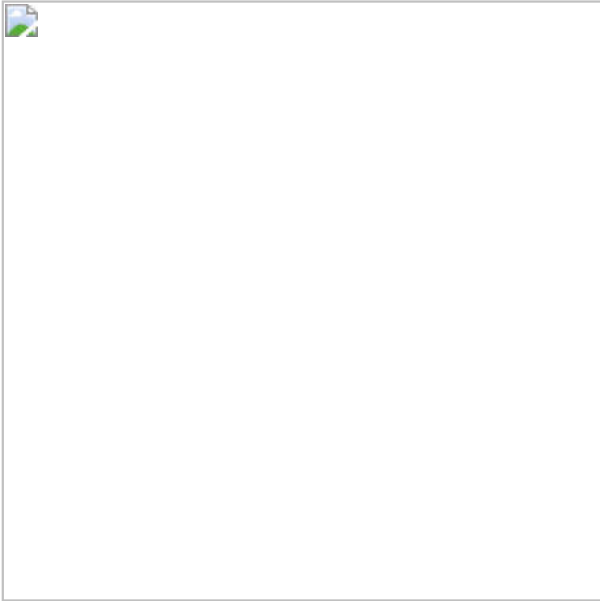
9) 你谈对象了，她妈让你带上房产证去提货，你若没有，那她妈会问你妈有没有，你妈没有那她妈会问你妈她妈有没有.....这就是原型链的向上搜索。

10) 你会继承你妈的样子，但是你也可以去染发洗剪吹，就是说对象的属性可以自定义，会覆盖继承得到的属性。

11) 虽然你洗剪吹了染成黄毛了，但你不能改变你妈的样子，你妈生的弟弟妹妹跟你的黄毛洗剪吹没一点关系，就是说对象实例不能改动原型的属性。

12) 但是你家被你玩火烧了的话，那就是说你家你妈家你弟们家都被烧了，这就是原型属性的共享。

13) 你妈外号阿珍，邻居大娘都叫你阿珍儿，但你妈头发从飘柔做成了金毛狮王后，隔壁大婶都改口叫你包租仔，这叫原型的动态性。



14) 你妈爱美，又跑到韩国整形，整到你妈他妈都认不出来，即使你妈头发换回飘柔了，但隔壁邻居还是叫你金毛狮王子。因为没人认出你妈，整形后的你妈已经回炉再造了，这就是原型的整体重写。

尼玛！你特么也是够了！ Don't BB！ Show me the code！

```
function Person (name) { this.name = name; }
function Mother () {}
Mother.prototype = { //Mother的原型
age: 18,
home: ['Beijing', 'Shanghai']
};
Person.prototype = new Mother(); //Person的原型为Mother

//用chrome调试工具查看，提供了__proto__接口查看原型，这里有两层原型，各位还是直接看chrome好一点。var p1 =
new Person('Jack'); //p1: 'Jack'; __proto__: {__proto__: 18, ['Beijing', 'Shanghai']} var p2 = new Person('Mark');
//p2: 'Mark'; __proto__: {__proto__: 18, ['Beijing', 'Shanghai']}
p1.age = 20;
/* 实例不能改变原型的基本值属性，正如你洗剪吹染黄毛跟你妈无关
* 在p1实例下增加一个age属性的普通操作，与原型无关。跟var o={}; o.age=20一样。
* p1：下面多了个属性age，而__proto__跟 Mother.prototype一样，age=18。
* p2：只有属性name，__proto__跟 Mother.prototype一样
*/

p1.home[0] = 'Shenzhen';
/* 原型中引用类型属性的共享，正如你烧了你家，就是烧了你全家的家
* 这个先过，下文再仔细唠叨一下可好？
* p1：'Jack',20, {__proto__: {__proto__: 18, ['Shenzhen', 'Shanghai']}}
* p2：'Mark'; __proto__: {__proto__: 18, ['Shenzhen', 'Shanghai']}}
*/

p1.home = ['Hangzhou', 'Guangzhou'];
/* 其实跟p1.age=20一样的操作。换成这个理解: var o={}; o.home=['big','house']
* p1：'Jack',20,['Hangzhou','Guangzhou']; __proto__: {__proto__: 18, ['Shenzhen', 'Shanghai']}}
* p2：'Mark'; __proto__: {__proto__: 18, ['Shenzhen', 'Shanghai']}}
*/
```



```
delete p1.age;
```

/* 删除实例的属性之后，原本被覆盖的原型值就重见天日了。正如你剃了光头，遗传的迷人小卷发就长出来了。

* 这就是向上搜索机制，先搜你，然后你妈，再你妈他妈，所以你妈的改动会动态影响你。

* p1 : 'Jack', ['Hangzhou', 'Guangzhou']; __proto__: {__proto__: 18, ['Shenzhen', 'Shanghai']}

* p2 : 'Mark'; __proto__: {__proto__: 18, ['Shenzhen', 'Shanghai']}

*/

```
Person.prototype.lastName = 'Jin';
```

/* 改写原型，动态反应到实例中。正如你妈变新潮了，邻居提起你都说你妈真潮。

* 注意，这里我们改写的是Person的原型，就是往Mother里加一个lastName属性，等同于Mother.lastName='Jin'

* 这里并不是改Mother.prototype，改动不同的层次，效果往往会有很大的差异。

* p1 : 'Jack', ['Hangzhou', 'Guangzhou']; __proto__: {__proto__: 18, ['Shenzhen', 'Shanghai']}

* p2 : 'Mark'; __proto__: {__proto__: 18, ['Shenzhen', 'Shanghai']}

*/

```
Person.prototype = {
```

```
  age: 28,
```

```
  address: { country: 'USA', city: 'Washington' }
```

```
};
```

```
var p3 = new Person('Obama');
```

/* 重写原型！这个时候Person的原型已经完全变成一个新的对象了，也就是说Person换了个妈，叫后妈。

* 换成这样理解：var a=10; b=a; a=20; c=a。所以b不变，变得是c，所以p3跟着后妈变化，与亲妈无关。

* p1 : 'Jack', ['Hangzhou', 'Guangzhou']; __proto__: {__proto__: 18, ['Shenzhen', 'Shanghai']}

* p2 : 'Mark'; __proto__: {__proto__: 18, ['Shenzhen', 'Shanghai']}

* p3: 'Obama'; __proto__: 28 {country: 'USA', city: 'Washington'}

*/

```
Mother.prototype.no = 9527;
```

/* 改写原型的原型，动态反应到实例中。正如你妈他妈变新潮了，邻居提起你都说你丫外婆真潮。

* 注意，这里我们改写的是Mother.prototype，p1p2会变，但上面p3跟亲妈已经了无瓜葛了，不影响他。

* p1 : 'Jack', ['Hangzhou', 'Guangzhou']; __proto__: {__proto__: 18, ['Shenzhen', 'Shanghai'], 9527}

* p2 : 'Mark'; __proto__: {__proto__: 18, ['Shenzhen', 'Shanghai'], 9527}

* p3: 'Obama'; __proto__: 28 {country: 'USA', city: 'Washington'}

*/

```
Mother.prototype = {
```

```
  car: 2,
```

```
  hobby: ['run', 'walk']
```

```
};
```

```
var p4 = new Person('Tony');
```

/* 重写原型的原型！这个时候Mother的原型已经完全变成一个新的对象了！人他妈换了个后妈！

* 由于上面Person与Mother已经断开联系了，这时候Mother怎么变已经不影响Person了。

* p4: 'Tony'; __proto__: 28 {country: 'USA', city: 'Washington'}

*/

```
Person.prototype = new Mother(); //再次绑定var p5 = new Person('Luffy');
```

// 这个时候如果需要应用这些改动的话，那就要重新将Person的原型绑定到Mother上了

// p5: 'Luffy'; __proto__: {__proto__: 2, ['run', 'walk']}

p1.__proto__.__proto__.__proto__.__proto__ //null，你说原型链的终点不是null？

Mother.__proto__.__proto__.__proto__ //null，你说原型链的终点不是null？

看完基本能理解了吧？

现在再来说说 p1.age = 20、p1.home = ['Hangzhou', 'Guangzhou'] 和 p1.home[0] = 'Shenzhen' 的区别。p1.home[0] = 'Shenzhen'; 总结一下是 p1.object.method，p1.object.property 这样的形式。

p1.age = 20; p1.home = ['Hangzhou', 'Guangzhou'];这两句还是比较好理解的，先忘掉原型吧，想想我们是怎么为一个普通对象增加属性的：

```
var obj = new Object();
obj.name='xxx';
obj.num = [100, 200];
```

这样是不是就理解了呢？一样一样的呀。

那为什么 p1.home[0] = 'Shenzhen' 不会在 p1 下创建一个 home 数组属性，然后将其首位设为 'Shenzhen'呢？我们还是先忘了这个，想想上面的obj对象，如果写成这样：var obj.name = 'xxx', obj.num = [100, 200]，能得到你要的结果吗？显然，除了报错你什么都得不到。因为obj还未定义，又怎么能往里面加入东西呢？同理，p1.home[0]中的 home 在 p1 下并未被定义，所以也不能直接一步定义 home[0] 了。如果要在p1下创建一个 home 数组，当然是这么写了：

```
p1.home = [];
p1.home[0] = 'Shenzhen';
```

这不就是我们最常用的办法吗？

而之所以 p1.home[0] = 'Shenzhen' 不直接报错，是因为在原型链中有一个搜索机制。当我们输入 p1.object 的时候，原型链的搜索机制是先在实例中搜索相应的值，找不到就在原型中找，还找不到就再往上一级原型中搜索.....一直到了原型链的终点，就是到null还没找到的话，就返回一个 undefined。当我们输入 p1.home[0] 的时候，也是同样的搜索机制，先搜索 p1 看有没有名为 home 的属性和方法，然后逐级向上查找。最后我们在Mother的原型里面找到了，所以修改他就相当于修改了 Mother 的原型啊。

一句话概括：p1.home[0] = 'Shenzhen' 等同于 Mother.prototype.home[0] = 'Shenzhen'。

由上面的分析可以知道，**原型链继承的主要问题在于属性的共享，很多时候我们只想共享方法而并不想要共享属性，理想中每个实例应该有独立的属性。**因此，原型继承就有了下面的两种改良方式：

1) 组合继承

```
function Mother (age) {
  this.age = age;
  this.hobby = ['running','football']
}
Mother.prototype.showAge = function () {
  console.log(this.age);
};

function Person (name, age) {
  Mother.call(this, age);    //第二次执行
  this.name = name;
}
Person.prototype = new Mother(); //第一次执行
Person.prototype.constructor = Person;
Person.prototype.showName = function () {
  console.log(this.name);
}
```

```
var p1 = new Person('Jack', 20);
p1.hobby.push('basketball'); //p1: 'Jack'; __proto__:20,['running','football']
var p2 = new Person('Mark', 18);
//p2: 'Mark'; __proto__:18,['running','football']
```

[View Code](#)

结果是酱紫的：

这里第一次执行的时候，得到 `Person.prototype.age = undefined`, `Person.prototype.hobby = ['running','football']`，第二次执行也就是 `var p1 = new Person('Jack', 20)` 的时候，得到 `p1.age = 20`, `p1.hobby = ['running','football']`，push后就变成了 `p1.hobby = ['running','football', 'basketball']`。其实分辨好 `this` 的变化，理解起来也是比较简单的，把 `this` 简单替换一下就能得到这个结果了。如果感觉理解起来比较绕的话，试着把脑子里面的概念扔掉吧，把自己当浏览器从上到下执行一遍代码，结果是不是就出来了呢？

通过第二次执行原型的构造函数 `Mother()`，我们在对象实例中复制了一份原型的属性，这样就做到了与原型属性的分离独立。细心的你会发现，我们第一次调用 `Mother()`，好像什么用都没有呢，能不调用他吗？可以，就有了下面的寄生组合式继承。

2) 寄生组合式继承

```
function object(o){
function F(){}
F.prototype = o;
return new F();
}

function inheritPrototype(Person, Mother){
var prototype = object(Mother.prototype);
prototype.constructor = Person;
Person.prototype = prototype;
}

function Mother (age) {
this.age = age;
this.hobby = ['running','football']
}
Mother.prototype.showAge = function () {
console.log(this.age);
};

function Person (name, age) {
Mother.call(this, age);
this.name = name;
}

inheritPrototype(Person, Mother);

Person.prototype.showName = function () {
console.log(this.name);
}
```

```
var p1 = new Person('Jack', 20);
p1.hobby.push('basketball'); //p1: 'Jack'; __proto__:20,['running','football']
var p2 = new Person('Mark', 18);
//p2: 'Mark'; __proto__:18,['running','football']
```

[View Code](#)

结果是酱紫的：

原型中不再有 age 和 hobby 属性了，只有两个方法，正是我们想要的结果！

关键点在于 object(o) 里面，这里借用了一个临时对象来巧妙避免了调用 new Mother()，然后将原型为 o 的新对象实例返回，从而完成了原型链的设置。很绕，对吧，那是因为我们不能直接设置 Person.prototype = Mother.prototype 啊。

小结

说了这么多，其实核心只有一个：属性共享和独立的控制，当你的对象实例需要独立的属性，所有做法的**本质都是在对象实例里面创建属性**。若不考虑太多，你大可以在 Person 里面直接定义你所需要独立的属性来覆盖掉原型的属性。总之，使用原型继承的时候，要对于原型中的属性要特别注意，因为他们都是牵一发而动全身的存在。

下面简单罗列下 js 中创建对象的各种方法，现在最常用的方法是组合模式，熟悉的同学可以跳过到文章末尾点赞了。


1) 原始模式

```
//1.原始模式，对象字面量方式var person = {
  name: 'Jack',
  age: 18,
  sayName: function () { alert(this.name); }
};
//1.原始模式，Object构造函数方式var person = new Object();
person.name = 'Jack';
person.age = 18;
person.sayName = function () {
  alert(this.name);
};
```

显然，当我们要创建批量的人 person1、person2.....时，每次都要敲很多代码，资深 copypaster 都吃不消！然后就有了批量生产的工厂模式。

2) 工厂模式

```
//2.工厂模式，定义一个函数创建对象function creatPerson (name, age) {
  var person = new Object();
  person.name = name;
  person.age = age;
  person.sayName = function () {
    alert(this.name);
  };
  return person;
}
```

工厂模式就是批量化生产，简单调用就可以进入造人模式 ( 啪啪啪.....)。指定姓名年龄就可以造一堆小宝宝啦，解放双手。但是由于是工厂暗箱操作的，所以你不能识别这个对象到底是什

么类型、是人还是狗傻傻分不清 (instanceof 测试为 Object) , 另外每次造人时都要创建一个独立的temp对象, 代码臃肿, 雅蠃蝶啊。

3) 构造函数

```
//3.构造函数模式, 为对象定义一个构造函数function Person (name, age) {  
  this.name = name;  
  this.age = age;  
  this.sayName = function () {  
    alert(this.name);  
  };  
}  
var p1 = new Person('Jack', 18); //创建一个p1对象  
Person('Jack', 18); //属性方法都给window对象, window.name='Jack', window.sayName()会输出Jack
```

构造函数与C++、JAVA中类的构造函数类似, 易于理解, 另外Person可以作为类型识别 (instanceof 测试为 Person 、 Object) 。但是所有实例依然是独立的, 不同实例的方法其实是不同的函数。这里把函数两个字忘了吧, 把sayName当做一个对象就好理解了, 就是说张三的sayName 和李四的 sayName是不同的存在, 但显然我们期望的是共用一个 sayName 以节省内存。

4) 原型模式

```
//4.原型模式, 直接定义prototype属性function Person () {}  
Person.prototype.name = 'Jack';  
Person.prototype.age = 18;  
Person.prototype.sayName = function () { alert(this.name); };  
//4.原型模式, 字面量定义方式function Person () {}  
Person.prototype = {  
  name: 'Jack',  
  age: 18,  
  sayName: function () { alert(this.name); }  
};  
var p1 = new Person(); //name='Jack'var p2 = new Person(); //name='Jack'
```

这里需要注意的是原型属性和方法的共享, 即所有实例中都只是引用原型中的属性方法, 任何一个地方产生的改动会引起其他实例的变化。

5) 混合模式 (构造+原型)

按 Ctrl+C 复制代码

按 Ctrl+C 复制代码

做法是将需要独立的属性方法放入构造函数中, 而可以共享的部分则放入原型中, 这样做可以最大限度节省内存而又保留对象实例的独立性