[浅谈 JS 创建对象的 8 种模式](https://segmentfault.com/a/1190000003862596)

* [javascript](https://segmentfault.com/t/javascript/blogs)

[**Tomson**](https://segmentfault.com/u/tomson) 2015年10月15日发布

* |   3 收藏  |  117
* 4.1k 次浏览
* Objct 模式
* 工厂模式
* 构造器模式
* 通过 Function 对象实现
* prototype 模式
* 构造器与原型方式的混合模式
* 动态原型模式
* 混合工厂模式

**1.Object 模式**

**var** o1 = {};//字面量的表现形式

**var** o2 = **new** Object;

**var** o3 = **new** Object();

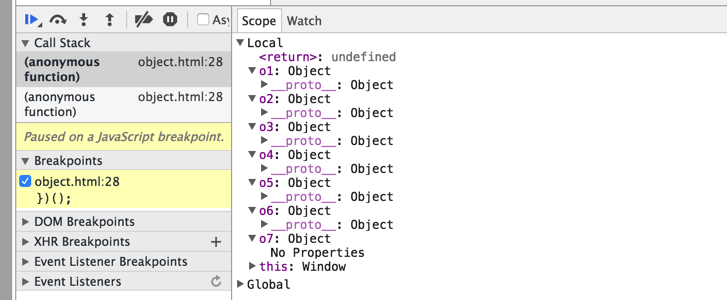
**var** o4 = **new** Object(null);

**var** o5 = **new** Object(undefined);

**var** o6 = Object.create(Object.prototype);//等价于 var o = {};//即以 Object.prototype 对象为一个原型模板,新建一个以这个原型模板为原型的对象

//区别

**var** o7 = Object.create(null);//创建一个原型为 null 的对象

在 chrome 里查看各个新建对象的区别:  


可以看出前6种模式创建出来的对象都是一样的,第七种不同点在于其虽然也为 Object 对象但其无任何属性(包括没有任何可以继承的属性,因为创建的时候没有指定其原型)

**2.工厂模式**

//工厂方法1 通过一个方法来创建对象 利用 arguments 对象获取参数设置属性(参数不直观,容易出现问题)

**function** **createCar**(){

**var** oTemp = **new** Object();

oTemp.name = arguments[0];//直接给对象添加属性，每个对象都有直接的属性

oTemp.age = arguments[1];

oTemp.showName = **function** () {

alert(**this**.name);

};//每个对象都有一个 showName 方法版本

**return** oTemp;

}

createCar("tom").showName();//在 JS 中没有传递的实参,实际形参值为 undefined(这里的 age 为 undefined)

createCar("tim",80).showName();

alert(createCar("tom") **instanceof** Object);//true 判断对象是否 Object 类或子类

//工厂方法2 通过传参设置属性(参数直观明了)

**function** **createCar**(name,age){

**var** oTemp = **new** Object();

oTemp.name = name;//直接给对象添加属性，每个对象都有直接的属性

oTemp.age = age;

oTemp.showName = **function** () {

alert(**this**.name);

};//每个对象都有一个 showName 方法版本

**return** oTemp;

}

createCar("tom").showName();

createCar("tim",80).showName();

alert(createCar("tom") **instanceof** Object);//true 判断对象是否 Object 类或子类

**3.构造器模式**

//构造器方法1

**function** **Car**(sColor,iDoors){ //声明为构造器时需要将函数名首字母大写

**this**.color = sColor; //构造器内直接声明属性

**this**.doors = iDoors;

**this**.showColor = **function**(){

**return** **this**.color;

};//每个 Car 对象都有自己的 showColor方法版本

**this**.showDoor = **function** () {

**return** **this**.doors;

}

}

使用方法1的问题很明显,没办法是 showDoor 方法重用,每次新建一个对象就要在堆里新开辟一篇空间.改进如下

//构造器方法2

**function** **showDoor**(){ //定义一个全局的 Function 对象

**return** **this**.doors;

}

**function** **Car**(sColor,iDoors){//构造器

**this**.color = sColor; //构造器内直接声明属性

**this**.doors = iDoors;

**this**.showColor = **function**(){

**return** **this**.color;

};

**this**.showDoor = showDoor();//每个 Car 对象共享同一个 showDoor 方法版本(方法有自己的作用域，不用担心变量被共享)

}

alert(**new** Car("red",2).showColor());//通过构造器创建一个对象并调用其对象方法

上面出现的问题就是语义不够清除,体现不出类的封装性,改进为 prototype 模式

**4.通过Function对象实现创建对象**

我们知道每声明一个函数实际是创建了一个Function 实例 [JS 函数](http://segmentfault.com/a/1190000003848925).

**function** **function\_name**(param1,param2){alert(param1);}

//等价于

**var** function\_name = **new** **Function**("param1","pram2","alert(param1);");

**var** Car2 = **new** **Function**("sColor","iDoors",

"this.color = sColor;"+

"this.doors = iDoors;"+

"this.showColor = function(){ return this.color; }"

);

alert(**new** Car2("blue",3).showColor());

**5.prototype模式**

* 类通过 prototype 属性添加的属性与方法都是绑定在这个类的 prototype 域(实际为一个 Prototype 对象)中，绑定到这个域中的属性与方法只有一个版本，只会创建一次.
* 类的实例对象可以直接像调用自己的属性一样调用该类的 prototype 域中的属性与方法，类可以通过调用 prototype 属性来间接调用prototype 域内的属性与方法.

注意：通过类实例化出对象后对象内无 prototype 属性，但对象可直接像访问属性一样的访问类的 prototype 域的内容，实例对象有个私有属性\_\_proto\_\_,\_\_proto\_\_属性内含有类的 prototype 域内的属性与方法

方法1

**function** **Car3**(){}//用空构造函数设置类名

Car3.prototype.color = "blue";//每个对象都共享相同属性

Car3.prototype.doors = 3;

Car3.prototype.drivers = new Array("Mike","John");

Car3.prototype.showColor = **function**(){

alert(this.color);

};//每个对象共享一个方法版本，省内存。

var car3\_1 = new Car3();

var car3\_2 = new Car3();

alert(car3\_1.color);//blue

alert(car3\_2.color);//blue

alert(Car3.prototype.color);//blue

car3\_1.drivers.push("Bill");

alert(car3\_1.drivers);//"Mike","John","Bill"

alert(car3\_2.drivers);//"Mike","John","Bill"

alert(Car3.prototype.drivers);//"Mike","John","Bill"

//直接修改实例对象的属性,解析器会先去找实例对象是否有这个属性(不会去找实例对象的 \_proto\_ 属性内的那些类的 prototype 属性，而是直接查看这个实例是否有对应的属性(与\_proto\_同级))

//如果没有则直接给这个实例对象添加该属性，但不会修改类的prototype域的同名属性，既实例对象的\_proto\_属性内的那些类 prototype 域属性不会被修改

car3\_1.color = "red";//car3\_1对象内无名为 color 的对象属性，故将该属性添加到该对象上

//解析器对实例对象读取属性值的时候会先查找该实例有无同名的直接属性

//如果没有，则查找\_\_proto\_\_属性内保存的那些 当前类的 prototype 域的属性

//有就返回，无则继续查找是否有原型链中的对应的方法属性

//有就返回，无则返回undefined

alert(car3\_1.color);//red

alert(car3\_2.color);//blue

alert(car3\_2.color2);//undefined

//直接修改类的 prototype 域内的属性，不会影响该类的实例对象的对象属性，但会影响实例对象的\_proto\_属性(\_proto\_属性内存放的是类的 prototype 域的内容)

Car3.prototype.color = "black";

alert(car3\_1.color);//red 该对象有同名的直接属性，故不会去\_proto\_属性内查找类的 prototype 域的属性

alert(car3\_2.color);//black 受影响

//直接修改实例对象的方法,解析器会先去找实例对象是否有这个方法(不会去找实例对象的 \_proto\_ 属性内的那些类的 prototype 域的方法，而是直接查看这个实例是否有对应的方法(与\_proto\_同级))

//如果没有则直接给这个实例对象添加该方法，但不会修改类的prototype域的同名方法，既实例对象的\_proto\_属性内的那些类 prototype 域方法不会被修改

//car3\_1对象内无名为 showColor 的对象方法属性，故将该方法属性添加到该对象上

car3\_1.showColor = **function** () {

alert("new function");

}

//解析器对实例对象调用方法属性的时候会先查找该实例有无同名的直接方法属性

//如果没有，则查找\_proto\_属性内保存的那些 当前类的 prototype 域的方法属性

//有就返回，无则继续查找是否有原型链中的对应的方法属性

//找到就返回,无则报错

car3\_1.showColor();//new function

car3\_2.showColor();//blue

car3\_1.abcd();//直接报错

//直接修改类的 prototype 域内的方法属性，不会影响该类的实例对象的方法属性，但会影响实例对象的\_proto\_属性(\_proto\_属性内存放的是类的 prototype 域的内容)

Car3.prototype.showColor = **function** () {

alert("second function");

}

car3\_1.showColor();//new function 该对象有同名的方法属性，故不会去\_proto\_属性内查找类的 prototype 域的方法属性

car3\_2.showColor();//second function 受影响

可以看出使用该方法虽然说打打减少了内存的浪费,但依旧有问题,某个对象的属性一旦改变,所有由该类实例化得到的对象的\_\_proto\_\_内属性值也会跟着变(实为引用),改进如下

**6.构造器方式与原型方式的混合模式**

//每个对象有专属的属性不会与其他对象共享

**function** **Car4**(sColor,iDoors){

this.\_color = sColor;//私有属性变量名称头加下划线标识

this.\_doors = iDoors;

this.drivers = **new** **Array**("Mike","John");//公有属性标识

}

//所有对象共享一个方法版本，减少内存浪费

Car4.prototype.showColor = **function** () {

alert(this.\_color);

};

**var** car4\_1 = **new** **Car4**("red",4);

**var** car4\_2 = **new** **Car4**("blue",3);

car4\_1.drivers.push("Bill");

alert(car4\_1.drivers);//"Mike","John","Bill"

alert(car4\_2.drivers);//"Mike","John"

这也是常用的创建对象方式之一

**7.动态原型模式**

**function** **Car5**(sColor,iDoors,iMpg){

**this**.color = sColor;

**this**.doors = iDoors;

**this**.mpg = iMpg;

**this**.drivers = **new** Array("Mike","John");

//使用标志(\_initialized)来判断是否已给原型赋予了任何方法,保证方法永远只被创建并赋值一次

**if**(**typeof** Car5.\_initialized == "undefined"){//因为这里的标记是附加在类上,故如果后期直接对其进行修改,还是有可能出现再次创建的情况

Car5.prototype.showColor = **function** () {//为Car5添加一个存放在 prototype 域的方法

alert(**this**.color);

};

Car5.\_initialized = true;//设置一个静态属性

}

}

**var** car5\_1 = **new** Car5("red",3,25);

**var** car5\_2 = **new** Car5("red",3,25);

这种模式使得定义类像强类型语言例如 java 等语言的定义模式

**8.混合工厂模式**

**function** **Car6**(){

**var** oTempCar = **new** Object;

oTempCar.color = "blue";

oTempCar.doors = 4;

oTempCar.showColor = **function** () {

alert(**this**.color);

};

**return** oTempCar;

}

**var** car6 = **new** Car6();

由于在 Car6()构造函数内部调用了 new 运算符,所以将忽略第二个 new 运算符(位于构造函数之外),在构造函数内部创建的对象被传递回变量car6,这种方式在对象方法的内部管理方面与经典方式(工厂方法)有着相同的问题.应尽量避免