[**JavaScript面向对象编程深入分析(1)**](http://www.cnblogs.com/jenry/archive/2013/01/01/2841411.html)

一. **Javascript 面向对象编程：封装**

Javascript是一种基于对象(object-based)的语言，你遇到的所有东西几乎都是对象。但是，它又不是一种真正的面向对象编程(OOP)语言，因为它的语法中没有class(类)。

那么，如果我们要把"属性"(property)和"方法"(method)，封装成一个对象，甚至要从原型对象生成一个实例对象，我们应该怎么做呢?

**1. 生成对象的原始模式**

假定我们把猫看成一个对象，它有"名字"和"颜色"两个属性。

1. var Cat = {
2. name : '',
3. color : ''
4. }

现在，我们需要根据这个原型对象，生成两个实例对象。

1. var cat1 = {}; // 创建一个空对象
2. cat1.name = "大毛"; // 按照原型对象的属性赋值
3. cat1.color = "黄色";
4. var cat2 = {};
5. cat2.name = "二毛";
6. cat2.color = "黑色";

好了，这就是最简单的封装了。但是，这样的写法有两个缺点，一是如果多生成几个实例，写起来就非常麻烦;二是实例与原型之间，没有任何办法，可以看出有什么联系。

**2. 原始模式的改进**

我们可以写一个函数，解决代码重复的问题。

1. function Cat(name,color){
2. return {
3. name:name,
4. color:color
5. }
6. }

然后生成实例对象，就等于是在调用函数：

1. var cat1 = Cat("大毛","黄色");
2. var cat2 = Cat("二毛","黑色");

这种方法的问题依然是，cat1和cat2之间没有内在的联系，不能反映出它们是同一个原型对象的实例。

**3. 构造函数模式**

为了解决从原型对象生成实例的问题，Javascript提供了一个构造函数(Constructor)模式。

所谓"构造函数"，其实就是一个普通函数，但是内部使用了this变量。对构造函数使用new运算符，就能生成实例，并且this变量会绑定在实例对象上。

比如，猫的原型对象现在可以这样写，

1. function Cat(name,color){
2. this.name=name;
3. this.color=color;
4. }

我们现在就可以生成实例对象了。

1. var cat1 = new Cat("大毛","黄色");
2. var cat2 = new Cat("二毛","黑色");
3. alert(cat1.name); // 大毛
4. alert(cat1.color); // 黄色

这时cat1和cat2会自动含有一个constructor属性，指向它们的构造函数。

1. alert(cat1.constructor == Cat); //true
2. alert(cat2.constructor == Cat); //true

Javascript还提供了一个instanceof运算符，验证原型对象与实例对象之间的关系。

1. alert(cat1 instanceof Cat); //true
2. alert(cat2 instanceof Cat); //true

**4. 构造函数模式的问题**

构造函数方法很好用，但是存在一个浪费内存的问题。

请看，我们现在为Cat对象添加一个不变的属性"type"(种类)，再添加一个方法eat(吃老鼠)。那么，原型对象Cat就变成了下面这样：

1. function Cat(name,color){
2. this.name = name;
3. this.color = color;
4. this.type = "猫科动物";
5. this.eat = function(){alert("吃老鼠");};
6. }

还是采用同样的方法，生成实例：

1. var cat1 = new Cat("大毛","黄色");
2. var cat2 = new Cat ("二毛","黑色");
3. alert(cat1.type); // 猫科动物
4. cat1.eat(); // 吃老鼠

表面上好像没什么问题，但是实际上这样做，有一个很大的弊端。那就是对于每一个实例对象，type属性和eat()方法都是一模一样的内容，每一次生成一个实例，都必须为重复的内容，多占用一些内存。这样既不环保，也缺乏效率。

1. alert(cat1.eat == cat2.eat); //false

能不能让type属性和eat()方法在内存中只生成一次，然后所有实例都指向那个内存地址呢?回答是可以的。

**5. Prototype模式**

Javascript规定，每一个构造函数都有一个prototype属性，指向另一个对象。这个对象的所有属性和方法，都会被构造函数的实例继承。

这意味着，我们可以把那些不变的属性和方法，直接定义在prototype对象上。

1. function Cat(name,color){
2. this.name = name;
3. this.color = color;
4. }
5. Cat.prototype.type = "猫科动物";
6. Cat.prototype.eat = function(){alert("吃老鼠")};

然后，生成实例。

1. var cat1 = new Cat("大毛","黄色");
2. var cat2 = new Cat("二毛","黑色");
3. alert(cat1.type); // 猫科动物
4. cat1.eat(); // 吃老鼠

这时所有实例的type属性和eat()方法，其实都是同一个内存地址，指向prototype对象，因此就提高了运行效率。

1. alert(cat1.eat == cat2.eat); //true

**6. Prototype模式的验证方法**

**6.1 isPrototypeOf()**

这个方法用来判断，某个proptotype对象和某个实例之间的关系。

1. alert(Cat.prototype.isPrototypeOf(cat1)); //true
2. alert(Cat.prototype.isPrototypeOf(cat2)); //true

**6.2 hasOwnProperty()**

每个实例对象都有一个hasOwnProperty()方法，用来判断某一个属性到底是本地属性，还是继承自prototype对象的属性。

1. alert(cat1.hasOwnProperty("name")); // true
2. alert(cat1.hasOwnProperty("type")); // false

**6.3 in运算符**

in运算符可以用来判断，某个实例是否含有某个属性，不管是不是本地属性。

1. alert("name" in cat1); // true
2. alert("type" in cat1); // true

in运算符还可以用来遍历某个对象的所有属性。

1. for(var prop in cat1) { alert("cat1["+prop+"]="+cat1[prop]); }

[**JavaScript面向对象编程深入分析(2)**](http://www.cnblogs.com/jenry/archive/2013/01/01/2841413.html)

**二. Javascript 面向对象编程：构造函数的继承**

本节主要介绍，如何生成一个"继承"多个对象的实例。

比如，现在有一个"动物"对象的构造函数，

1. function Animal(){
2. this.species = "动物";
3. }

还有一个"猫"对象的构造函数，

1. function Cat(name,color){
2. this.name = name;
3. this.color = color;
4. }

怎样才能使"猫"继承"动物"呢?

**1. 构造函数绑定**

最简单的方法，大概就是使用call或apply方法，将父对象的构造函数绑定在子对象上，也就是在子对象构造函数中加一行：

1. function Cat(name,color){
2. Animal.apply(this, arguments);
3. this.name = name;
4. this.color = color;
5. }
6. var cat1 = new Cat("大毛","黄色");
7. alert(cat1.species); // 动物

**2. prototype模式**

更常见的做法，则是使用prototype属性。

如果"猫"的prototype对象，指向一个Animal的实例，那么所有"猫"的实例，就能继承Animal了。

1. Cat.prototype = new Animal();
2. Cat.prototype.constructor = Cat;
3. var cat1 = new Cat("大毛","黄色");
4. alert(cat1.species); // 动物

代码的第一行，我们将Cat的prototype对象指向一个Animal的实例。

1. Cat.prototype = new Animal();

它相当于完全删除了prototype 对象原先的值，然后赋予一个新值。但是，第二行又是什么意思呢?

1. Cat.prototype.constructor = Cat;

原来，任何一个prototype对象都有一个constructor属性，指向它的构造函数。也就是说，Cat.prototype 这个对象的constructor属性，是指向Cat的。

我们在前一步已经删除了这个prototype对象原来的值，所以新的prototype对象没有constructor属性，所以我们必须手动加上去，否则后面的"继承链"会出问题。这就是第二行的意思。

总之，这是很重要的一点，编程时务必要遵守。下文都遵循这一点，即如果替换了prototype对象，

1. o.prototype = {};

那么，下一步必然是为新的prototype对象加上constructor属性，并将这个属性指回原来的构造函数。

1. o.prototype.constructor = o;

**3. 直接继承prototype**

由于Animal对象中，不变的属性都可以直接写入Animal.prototype。所以，我们也可以让Cat()跳过 Animal()，直接继承Animal.prototype。

现在，我们先将Animal对象改写：

1. function Animal(){ }
2. Animal.prototype.species = "动物";

然后，将Cat的prototype对象，然后指向Animal的prototype对象，这样就完成了继承。

1. Cat.prototype = Animal.prototype;
2. CatCat.prototype.constructor = Cat;
3. var cat1 = new Cat("大毛","黄色");
4. alert(cat1.species); // 动物

与前一种方法相比，这样做的优点是效率比较高(不用执行和建立Animal的实例了)，比较省内存。缺点是 Cat.prototype和Animal.prototype现在指向了同一个对象，那么任何对Cat.prototype的修改，都会反映到Animal.prototype。

所以，上面这一段代码其实是有问题的。请看第二行

1. Cat.prototype.constructor = Cat;

这一句实际上把Animal.prototype对象的constructor属性也改掉了!

1. alert(Animal.prototype.constructor); // Cat

**4. 利用空对象作为中介**

由于"直接继承prototype"存在上述的缺点，所以可以利用一个空对象作为中介。

1. var F = function(){};
2. F.prototype = Animal.prototype;
3. Cat.prototype = new F();
4. Cat.prototype.constructor = Cat;

F是空对象，所以几乎不占内存。这时，修改Cat的prototype对象，就不会影响到Animal的prototype对象。

1. alert(Animal.prototype.constructor); // Animal

**5. prototype模式的封装函数**

我们将上面的方法，封装成一个函数，便于使用。

1. function extend(Child, Parent) {
3. var F = function(){};
4. F.prototype = Parent.prototype;
5. Child.prototype = new F();
6. Child.prototype.constructor = Child;
7. Child.uber = Parent.prototype;
8. }

使用的时候，方法如下

1. extend(Cat,Animal);
2. var cat1 = new Cat("大毛","黄色");
3. alert(cat1.species); // 动物

这个extend函数，就是YUI库如何实现继承的方法。

另外，说明一点。函数体最后一行

1. Child.uber = Parent.prototype;

意思是为子对象设一个uber属性，这个属性直接指向父对象的prototype属性。这等于是在子对象上打开一条通道，可以直接调用父对象的方法。这一行放在这里，只是为了实现继承的完备性，纯属备用性质。

**6. 拷贝继承**

上面是采用prototype对象，实现继承。我们也可以换一种思路，纯粹采用"拷贝"方法实现继承。简单说，如果把父对象的所有属性和方法，拷贝进子对象，不也能够实现继承吗?

首先，还是把Animal的所有不变属性，都放到它的prototype对象上。

1. function Animal(){}
2. Animal.prototype.species = "动物";

然后，再写一个函数，实现属性拷贝的目的。

1. function extend2(Child, Parent) {
2. var p = Parent.prototype;
3. var c = Child.prototype;
4. for (var i in p) {
5. c[i] = p[i];
6. }
7. c.uber = p;
8. }

这个函数的作用，就是将父对象的prototype对象中的属性，一一拷贝给Child对象的prototype对象。

使用的时候，这样写：

1. extend2(Cat, Animal);
2. var cat1 = new Cat("大毛","黄色");
3. alert(cat1.species); // 动物

[**JavaScript面向对象编程深入分析(3)**](http://www.cnblogs.com/jenry/archive/2013/01/01/2841414.html)

**三. Javascript面向对象编程：非构造函数的继承**

本节介绍不使用构造函数实现"继承"。

**1. 什么是"非构造函数"的继承?**

比如，现在有一个对象，叫做"中国人"。

1. var Chinese = {
2. nation:'中国'
3. };

还有一个对象，叫做"医生"。

1. var Doctor ={
2. career:'医生'
3. }

请问怎样才能让"医生"去继承"中国人"，也就是说，我怎样才能生成一个"中国医生"的对象?

这里要注意，这两个对象都是普通对象，不是构造函数，无法使用构造函数方法实现"继承"。

**2. object()方法**

json格式的发明人Douglas Crockford，提出了一个object()函数，可以做到这一点。

1. function object(o) {
2. function F() {}
3. F.prototype = o;
4. return new F();
5. }

这个object()函数，其实只做一件事，就是把子对象的prototype属性，指向父对象，从而使得子对象与父对象连在一起。

使用的时候，第一步先在父对象的基础上，生成子对象：

1. var Doctor = object(Chinese);

然后，再加上子对象本身的属性：

1. Doctor.career = '医生';

这时，子对象已经继承了父对象的属性了。

1. alert(Doctor.nation); //中国

**3. 浅拷贝**

除了使用"prototype链"以外，还有另一种思路：把父对象的属性，全部拷贝给子对象，也能实现继承。

下面这个函数，就是在做拷贝：

1. function extendCopy(p) {
2. var c = {};
3. for (var i in p) {
4. c[i] = p[i];
5. }
6. c.uber = p;
7. return c;
8. }

使用的时候，这样写：

1. var Doctor = extendCopy(Chinese);
2. Doctor.career = '医生';
3. alert(Doctor.nation); // 中国

但是，这样的拷贝有一个问题。那就是，如果父对象的属性等于数组或另一个对象，那么实际上，子对象获得的只是一个内存地址，而不是真正拷贝，因此存在父对象被篡改的可能。

请看，现在给Chinese添加一个"出生地"属性，它的值是一个数组。

1. Chinese.birthPlaces = ['北京','上海','香港'];

通过extendCopy()函数，Doctor继承了Chinese。

1. var Doctor = extendCopy(Chinese);

然后，我们为Doctor的"出生地"添加一个城市：

1. Doctor.birthPlaces.push('厦门');

发生了什么事?Chinese的"出生地"也被改掉了!

1. alert(Doctor.birthPlaces); //北京, 上海, 香港, 厦门
2. alert(Chinese.birthPlaces); //北京, 上海, 香港, 厦门

所以，extendCopy()只是拷贝基本类型的数据，我们把这种拷贝叫做"浅拷贝"。这是早期jQuery实现继承的方式。

**4. 深拷贝**

所谓"深拷贝"，就是能够实现真正意义上的数组和对象的拷贝。它的实现并不难，只要递归调用"浅拷贝"就行了。

1. function deepCopy(p, c) {
2. var c = c || {};
3. for (var i in p) {
4. if (typeof p[i] === 'object') {
5. c[i] = (p[i].constructor === Array) ? [] : {};
6. deepCopy(p[i], c[i]);
7. } else {
8. c[i] = p[i];
9. }
10. }
11. return c;
12. }

使用的时候这样写：

1. var Doctor = deepCopy(Chinese);

现在，给父对象加一个属性，值为数组。然后，在子对象上修改这个属性：

1. Chinese.birthPlaces = ['北京','上海','香港'];
2. Doctor.birthPlaces.push('厦门');

这时，父对象就不会受到影响了。

1. alert(Doctor.birthPlaces); //北京, 上海, 香港, 厦门
2. alert(Chinese.birthPlaces); //北京, 上海, 香港

目前，jQuery库使用的就是这种继承方法。