# 面向对象的小总20170513

面向对象：

面向对象的语言有一个标志，那就是他们都有类的概念，通过类可以创建任意多个具有相同属性和方法的对象；

构造函数，原型和对象实例的关系哦：

每个构造函数都有一个原型对象，原型对象都包含一个指向构造函数的指针，而对象实例都包含一个指向原型对象的内部指针

---

**Javascript 面向对象编程（一）：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？

---

一、 生成实例对象的原始模式

假定我们把猫看成一个对象，它有"名字"和"颜色"两个属性。

```
var Cat = {
    name : '',
}
```

现在，我们需要根据这个原型对象的规格（schema），生成两个实例对象。

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

好了，这就是最简单的封装了，把两个属性封装在一个对象里面。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。

---

二、 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。

```
function Cat(name,color) {
    return {
        name:name,
        color:color
    }
}
```

然后生成实例对象，就等于是在调用函数：

```
var cat1 = Cat("大毛","黄色");
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。

---

三、 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，

```
function Cat(name,color){
    this.name=name;
    this.color=color;
}
```

我们现在就可以生成实例对象了。

```
var cat1 = new Cat("大毛","黄色");
var cat2 = new Cat("二毛","黑色");
alert(cat1.name); // 大毛
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。

```
alert(cat1.constructor == Cat); //true
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。

```
alert(cat1 instanceof Cat); //true
alert(cat2 instanceof Cat); //true
```

---

## 四、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）。那么，原型对象Cat就变成了下面这样：

```
function Cat(name,color){
    this.name = name;
    this.color = color;
    this.type = "猫科动物";
    this.eat = function(){alert("吃老鼠");};
}
```

还是采用同样的方法，生成实例：

```
var cat1 = new Cat("大毛","黄色");
var cat2 = new Cat ("二毛","黑色");
alert(cat1.type); // 猫科动物
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。

```
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。

---

## 五、 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。

```
function Cat(name,color){
    this.name = name;
    this.color = color;
```

```
}
Cat.prototype.type = "猫科动物";
Cat.prototype.eat = function(){alert("吃老鼠")};
```

然后，生成实例。

```
var cat1 = new Cat("大毛","黄色");
var cat2 = new Cat("二毛","黑色");
alert(cat1.type); // 猫科动物
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。

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

---

## 六、 Prototype模式的验证方法

为了配合prototype属性，Javascript定义了一些辅助方法，帮助我们使用它。，

### 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。

```
alert(Cat.prototype.isPrototypeOf(cat1)); //true
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。

```
alert(cat1.hasOwnProperty("name")); // true
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。

```
alert("name" in cat1); // true
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。

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

---

**Javascript面向对象编程（二）：构造函数的继承象生成实例：**

今天要介绍的是，对象之间的"继承"的五种方法。

比如，现在有一个"动物"对象的构造函数。

```
function Animal(){
    this.species = "动物";
}
```

还有一个"猫"对象的构造函数。

```
function Cat(name,color){
    this.name = name;
    this.color = color;
}
```

怎样才能使"猫"继承"动物"呢？

---

## 一、 构造函数绑定

第一种方法也是最简单的方法，使用call或apply方法，将父对象的构造函数绑定在子对象上，即在子对象构造函数中加一行：

```
function Cat(name,color){
    Animal.apply(this, arguments);
    this.name = name;
    this.color = color;
}
var cat1 = new Cat("大毛","黄色");
alert(cat1.species); // 动物
```

---

## 二、 prototype模式

第二种方法更常见，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。

```
Cat.prototype = new Animal();
Cat.prototype.constructor = Cat;
var cat1 = new Cat("大毛","黄色");
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。

```
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？

```
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。如果没有"Cat.prototype = new Animal();"这一行，Cat.prototype.constructor是指向Cat的；加了这一行以后，Cat.prototype.constructor指向Animal。

```
alert(Cat.prototype.constructor == Animal); //true
```

更重要的是，每一个实例也有一个constructor属性，默认调用prototype对象的constructor属性。

```
alert(cat1.constructor == Cat.prototype.constructor); // true
```

因此，在运行"Cat.prototype = new Animal();"这一行之后，cat1.constructor也指向Animal!

```
alert(cat1.constructor == Animal); // true
```

这显然会导致继承链的紊乱（cat1明明是用构造函数Cat生成的），因此我们必须手动纠正，将Cat.prototype对象的constructor值改为Cat。这就是第二行的意思。

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，

```
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。

```
o.prototype.constructor = o;
```

---

## 三、 直接继承prototype

第三种方法是对第二种方法的改进。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：

```
function Animal(){ }
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。

```
Cat.prototype = Animal.prototype;
Cat.prototype.constructor = Cat;
var cat1 = new Cat("大毛","黄色");
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

```
Cat.prototype.constructor = Cat;
```

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

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

---

## 四、 利用空对象作为中介

由于"直接继承prototype"存在上述的缺点，所以就有第四种方法，利用一个空对象作为中介。

```
var F = function(){};
F.prototype = Animal.prototype;
Cat.prototype = new F();
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。

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

我们将上面的方法，封装成一个函数，便于使用。

```
function extend(Child, Parent) {

    var F = function(){};
    F.prototype = Parent.prototype;
    Child.prototype = new F();
    Child.prototype.constructor = Child;
    Child.uber = Parent.prototype;
}
```

使用的时候，方法如下

```
extend(Cat,Animal);
var cat1 = new Cat("大毛","黄色");
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

```
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。（uber是一个德语词，意思是"向上"、"上一层"。）这等于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。

---

## 五、 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？这样我们就有了第五种方法。

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

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

然后，再写一个函数，实现属性拷贝的目的。

```
function extend2(Child, Parent) {
    var p = Parent.prototype;
    var c = Child.prototype;
    for (var i in p) {
        c[i] = p[i];
        }
    c.uber = p;
}
```

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：

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

---

**Javascript面向对象编程（三）：非构造函数的继承**

今天是最后一个部分，介绍不使用构造函数实现"继承"。

## 一、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"。

```
var Chinese = {
    nation:'中国'
};
```

还有一个对象，叫做"医生"。

```
var Doctor ={
    career:'医生'
}
```

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"。

---

## 二、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。

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

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：

```
var Doctor = object(Chinese);
```

然后，再加上子对象本身的属性：

```
Doctor.career = '医生';
```

这时，子对象已经继承了父对象的属性了。

```
alert(Doctor.nation); //中国
```

---

## 三、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：

```
function extendCopy(p) {
    var c = {};
    for (var i in p) {
        c[i] = p[i];
    }
    c.uber = p;
    return c;
}
```

使用的时候，这样写：

```
var Doctor = extendCopy(Chinese);
Doctor.career = '医生';
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。

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

通过extendCopy()函数，Doctor继承了Chinese。

```
var Doctor = extendCopy(Chinese);
```

然后，我们为Doctor的"出生地"添加一个城市：

```
Doctor.birthPlaces.push('厦门');
```

发生了什么事？Chinese的"出生地"也被改掉了！

```
alert(Doctor.birthPlaces); //北京, 上海, 香港, 厦门
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。

---

## 四、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。

```
function deepCopy(p, c) {
    var c = c || {};
```

```
        for (var i in p) {
            if (typeof p[i] === 'object') {
                c[i] = (p[i].constructor === Array) ? [] : {};
                deepCopy(p[i], c[i]);
            } else {
                  c[i] = p[i];
            }
        }
        return c;
    }
```

使用的时候这样写：

```
    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：

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

这时，父对象就不会受到影响了。

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

---

摘抄：阮一峰