第 15 章 面向对象与原型

学习要点：

1.学习条件

2.创建对象

3.原型

4.继承

主讲教师：李炎恢

合作网站：http://www.ibeifeng.com

讲师博客：http://hi.baidu.com/李炎恢

ECMAScript 有两种开发模式：1.函数式(过程化)，2.面向对象(OOP)。面向对象的语言有一个标志，那就是类的概念，而通过类可以创建任意多个具有相同属性和方法的对象。但是，ECMAScript 没有类的概念，因此它的对象也与基于类的语言中的对象有所不同。

一．学习条件

在 JavaScript 视频课程第一节课，就已经声明过，JavaScript 课程需要大量的基础。这里，我们再详细探讨一下：

1.xhtml 基础：JavaScript 方方面面需要用到。

2.扣代码基础：比如 XHTML,ASP,PHP 课程中的项目都有 JS 扣代码的过程。

3.面向对象基础：JS 的面向对象是非正统且怪异的，必须有正统面向对象基础。

4.以上三大基础，必须是基于项目中掌握的基础，只是学习基础知识不够牢固，必须在项目中掌握上面的基础即可。

以上基础可以推荐的教程：xhtml(83 课时)、asp(200 课时)、php 第一季(136 课时)、关于面向对象部分，可以选择 php 第二季和 php 第三季，也可以选择市面上比较优秀的 java 教程，java 教程都是面向对象的。

二．创建对象

创建一个对象，然后给这个对象新建属性和方法。

```
var box = new Object();                    //创建一个 Object 对象
box.name = 'Lee';                          //创建一个 name 属性并赋值
box.age = 100;                             //创建一个 age 属性并赋值
box.run = function () {                    //创建一个 run()方法并返回值
    return this.name + this.age + '运行中...';
};
alert(box.run());                          //输出属性和方法的值
```

上面创建了一个对象，并且创建属性和方法，在 run()方法里的 this，就是代表 box 对象本身。这种是 JavaScript 创建对象最基本的方法，但有个缺点，想创建一个类似的对象，就会产生大量的代码。

```
var box2 = box;                            //得到 box 的引用
box2.name = 'Jack';                        //直接改变了 name 属性
```

```
alert(box2.run());                    //用 box.run()发现 name 也改变了

var box2 = new Object();
box2.name = 'Jack';
box2.age = 200;
box2.run = function () {
    return this.name + this.age + '运行中...';
};
alert(box2.run());                    //这样才避免和 box 混淆，从而保持独立
```

为了解决多个类似对象声明的问题，我们可以使用一种叫做**工厂模式**的方法，这种方法就是为了解决实例化对象产生大量重复的问题。

```
function createObject(name, age) {    //集中实例化的函数
    var obj = new Object();
    obj.name = name;
    obj.age = age;
    obj.run = function () {
        return this.name + this.age + '运行中...';
    };
    return obj;
}

var box1 = createObject('Lee', 100);    //第一个实例
var box2 = createObject('Jack', 200);   //第二个实例
alert(box1.run());
alert(box2.run());                      //保持独立
```

工厂模式解决了重复实例化的问题，但还有一个问题，那就是识别问题，因为根本无法搞清楚他们到底是哪个对象的实例。

```
alert(typeof box1);                   //Object
alert(box1 instanceof Object);        //true
```

ECMAScript 中可以采用**构造函数**(构造方法)可用来创建特定的对象。类型于 Object 对象。

```
function Box(name, age) {             //构造函数模式
    this.name = name;
    this.age = age;
    this.run = function () {
        return this.name + this.age + '运行中...';
    };
}

var box1 = new Box('Lee', 100);       //new Box()即可
var box2 = new Box('Jack', 200);
```

```
alert(box1.run());
alert(box1 instanceof Box);                    //很清晰的识别他从属于 Box
```

使用构造函数的方法，即解决了重复实例化的问题，又解决了对象识别的问题，但问题是，这里并没有 new Object()，为什么可以实例化 Box()，这个是哪里来的呢？

使用了构造函数的方法，和使用工厂模式的方法他们不同之处如下：

1.构造函数方法没有显示的创建对象(new Object())；

2.直接将属性和方法赋值给 this 对象；

3.没有 renturn 语句。

构造函数的方法有一些规范：

1.函数名和实例化构造名相同且大写，(PS：非强制，但这么写有助于区分构造函数和普通函数)；

2.通过构造函数创建对象，必须使用 new 运算符。

既然通过构造函数可以创建对象，那么这个对象是哪里来的，new Object()在什么地方执行了？执行的过程如下：

1.当使用了构造函数，并且 new 构造函数()，那么就后台执行了 new Object()；

2.将构造函数的作用域给新对象，(即 new Object()创建出的对象)，而函数体内的 this 就代表 new Object()出来的对象。

3.执行构造函数内的代码；

4.返回新对象(后台直接返回)。

关于 this 的使用，this 其实就是代表当前作用域对象的引用。如果在全局范围 this 就代表 window 对象，如果在构造函数体内，就代表当前的构造函数所声明的对象。

```
var box = 2;
alert(this.box);                               //全局，代表 window
```

构造函数和普通函数的唯一区别，就是他们调用的方式不同。只不过，构造函数也是函数，必须用 new 运算符来调用，否则就是普通函数。

```
var box = new Box('Lee', 100);                 //构造模式调用
alert(box.run());

Box('Lee', 20);                                //普通模式调用，无效

var o = new Object();
Box.call(o, 'Jack', 200)                       //对象冒充调用
alert(o.run());
```

探讨构造函数内部的方法(或函数)的问题，首先看下两个实例化后的属性或方法是否相等。

```
var box1 = new Box('Lee', 100);                //传递一致
var box2 = new Box('Lee', 100);                //同上
```

```
alert(box1.name == box2.name);          //true，属性的值相等
alert(box1.run == box2.run);            //false，方法其实也是一种引用地址
alert(box1.run() == box2.run());        //true，方法的值相等，因为传参一致
```

可以把构造函数里的方法(或函数)用 new Function()方法来代替，得到一样的效果，更加证明，他们最终判断的是引用地址，唯一性。

```
function Box(name, age) {                   //new Function()唯一性
    this.name = name;
    this.age = age;
    this.run = new Function("return this.name + this.age + '运行中...'");
}
```

我们可以通过构造函数外面绑定同一个函数的方法来保证引用地址的一致性，但这种做法没什么必要，只是加深学习了解：

```
function Box(name, age) {
    this.name = name;
    this.age = age;
    this.run = run;
}

function run() {                            //通过外面调用，保证引用地址一致
    return this.name + this.age + '运行中...';
}
```

虽然使用了全局的函数 run()来解决了保证引用地址一致的问题，但这种方式又带来了一个新的问题，全局中的 this 在对象调用的时候是 Box 本身，而当作普通函数调用的时候，this 又代表 window。

三．原型

我们创建的每个函数都有一个 prototype(原型)属性，这个属性是一个对象，它的用途是包含可以由特定类型的所有实例共享的属性和方法。逻辑上可以这么理解：prototype 通过调用构造函数而创建的那个对象的原型对象。使用原型的好处可以让所有对象实例共享它所包含的属性和方法。也就是说，不必在构造函数中定义对象信息，而是可以直接将这些信息添加到原型中。

```
function Box() {}                           //声明一个构造函数

Box.prototype.name = 'Lee';                 //在原型里添加属性
Box.prototype.age = 100;
Box.prototype.run = function () {           //在原型里添加方法
    return this.name + this.age + '运行中...';
};
```

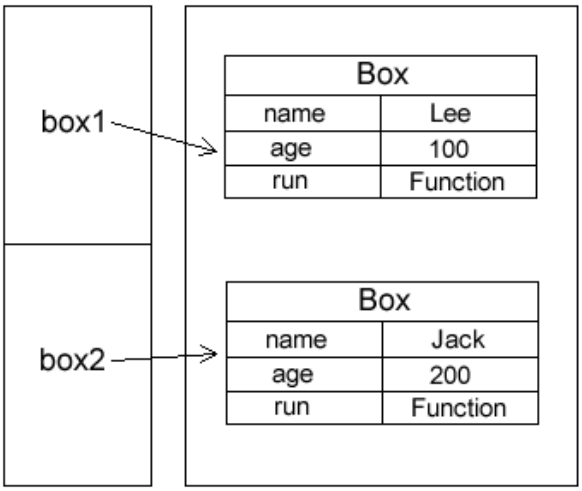
比较一下原型内的方法地址是否一致：

```
var box1 = new Box();
```

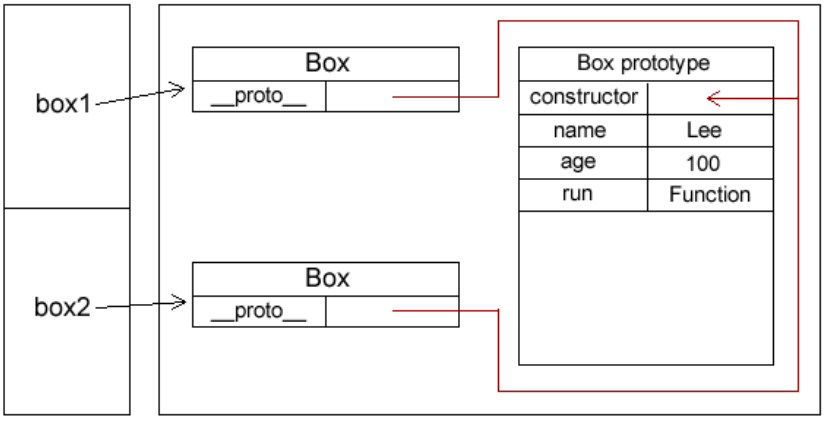
```
var box2 = new Box();
alert(box1.run == box2.run);                    //true，方法的引用地址保持一致
```

为了更进一步了解构造函数的声明方式和原型模式的声明方式，我们通过图示来了解一下：

构造函数方式



原型模式方式



在原型模式声明中，多了两个属性，这两个属性都是创建对象时自动生成的。__proto__属性是实例指向原型对象的一个指针，它的作用就是指向构造函数的原型属性 constructor。通过这两个属性，就可以访问到原型里的属性和方法了。

PS：IE 浏览器在脚本访问__proto__会不能识别，火狐和谷歌浏览器及其他某些浏览器均能识别。虽然可以输出，但无法获取内部信息。

```
alert(box1.__proto__);                          //[object Object]
```

判断一个对象是否指向了该构造函数的原型对象，可以使用 isPrototypeOf()方法来测试。

```
alert(Box.prototype.isPrototypeOf(box));        //只要实例化对象，即都会指向
```

原型模式的执行流程：

1.先查找构造函数实例里的属性或方法，如果有，立刻返回；

2.如果构造函数实例里没有，则去它的原型对象里找，如果有，就返回；

虽然我们可以通过对象实例访问保存在原型中的值，但却不能访问通过对象实例重写原型中的值。

```
var box1 = new Box();
alert(box1.name);                    //Lee，原型里的值
box1.name = 'Jack';
alert(box.1name);                    //Jack，就近原则，

var box2 = new Box();
alert(box2.name);                    //Lee，原型里的值，没有被 box1 修改
```

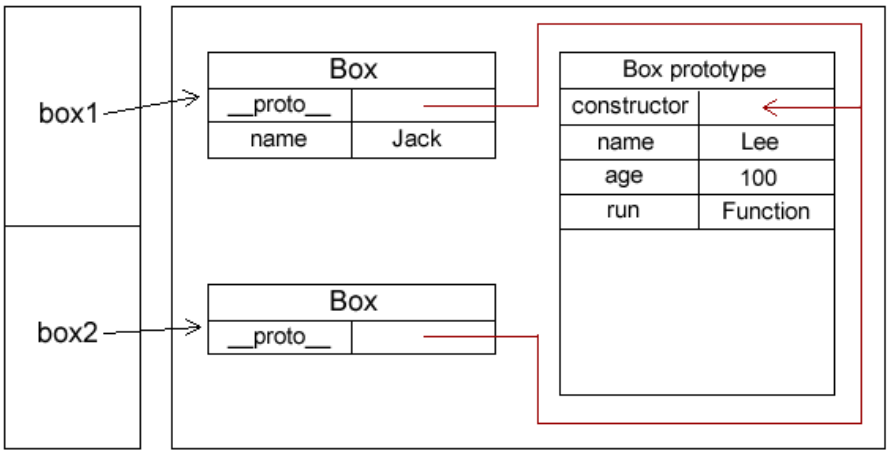
如果想要 box1 也能在后面继续访问到原型里的值，可以把构造函数里的属性删除即可，具体如下：

```
delete box1.name;                    //删除属性
alert(box1.name);
```

如何判断属性是在构造函数的实例里，还是在原型里？可以使用 hasOwnProperty()函数来验证：

```
alert(box.hasOwnProperty('name'));   //实例里有返回 true，否则返回 false
```

构造函数实例属性和原型属性示意图



in 操作符会在通过对象能够访问给定属性时返回 true，无论该属性存在于实例中还是原型中。

```
alert('name' in box);                //true，存在实例中或原型中
```

我们可以通过 **hasOwnProperty()**方法检测属性是否存在实例中，也可以通过 in 来判断实例或原型中是否存在属性。那么结合这两种方法，可以判断原型中是否存在属性。

```
function isProperty(object, property) {          //判断原型中是否存在属性
    return !object.hasOwnProperty(property) && (property in object);
}

var box = new Box();
alert(isProperty(box, 'name'))                   //true，如果原型有
```

为了让属性和方法更好的体现封装的效果，并且减少不必要的输入，原型的创建可以使用字面量的方式：

```
function Box() {};
Box.prototype = {                                //使用字面量的方式
    name : 'Lee',
    age : 100,
    run : function () {
        return this.name + this.age + '运行中...';
    }
};
```

使用构造函数创建原型对象和使用字面量创建对象在使用上基本相同，但还是有一些区别，字面量创建的方式使用 constructor 属性不会指向实例，而会指向 Object，构造函数创建的方式则相反。

```
var box = new Box();
alert(box instanceof Box);
alert(box instanceof Object);
alert(box.constructor == Box);                   //字面量方式，返回 false，否则，true
alert(box.constructor == Object);                //字面量方式，返回 true，否则，false
```

如果想让字面量方式的 constructor 指向实例对象，那么可以这么做：

```
Box.prototype = {
    constructor : Box,                           //直接强制指向即可
};
```

PS：字面量方式为什么 constructor 会指向 Object？因为 Box.prototype={};这种写法其实就是创建了一个新对象。而每创建一个函数，就会同时创建它 prototype，这个对象也会自动获取 constructor 属性。所以，新对象的 constructor 重写了 Box 原来的 constructor，因此会指向新对象，那个新对象没有指定构造函数，那么就默认为 Object。

原型的声明是有先后顺序的，所以，重写的原型会切断之前的原型。

```
function Box() {};

Box.prototype = {                                //原型被重写了
    constructor : Box,
    name : 'Lee',
    age : 100,
```

```
        run : function () {
            return this.name + this.age + '运行中...';
        }
    };

    Box.prototype = {
        age = 200
    };

    var box = new Box();                    //在这里声明
    alert(box.run());                       //box 只是最初声明的原型
```

原型对象不仅仅可以在自定义对象的情况下使用，而 ECMAScript 内置的引用类型都可以使用这种方式，并且内置的引用类型本身也使用了原型。

```
    alert(Array.prototype.sort);            //sort 就是 Array 类型的原型方法
    alert(String.prototype.substring);      //substring 就是 String 类型的原型方法

    String.prototype.addstring = function () {      //给 String 类型添加一个方法
        return this + '，被添加了！ ';              //this 代表调用的字符串
    };

    alert('Lee'.addstring());               //使用这个方法
```

PS：尽管给原生的内置引用类型添加方法使用起来特别方便，但我们不推荐使用这种方法。因为它可能会导致命名冲突，不利于代码维护。

原型模式创建对象也有自己的缺点，它省略了构造函数传参初始化这一过程，带来的缺点就是初始化的值都是一致的。而原型最大的缺点就是它最大的优点，那就是共享。

原型中所有属性是被很多实例共享的，共享对于函数非常合适，对于包含基本值的属性也还可以。但如果属性包含引用类型，就存在一定的问题：

```
    function Box() {};
    Box.prototype = {
        constructor : Box,
        name : 'Lee',
        age : 100,
        family : ['父亲', '母亲', '妹妹'],          //添加了一个数组属性
        run : function () {
            return this.name + this.age + this.family;
        }
    };

    var box1 = new Box();
```

```
box1.family.push('哥哥');                    //在实例中添加'哥哥'
alert(box1.run());

var box2 = new Box();
alert(box2.run());                           //共享带来的麻烦，也有'哥哥'了
```

PS：数据共享的缘故，导致很多开发者放弃使用原型，因为每次实例化出的数据需要保留自己的特性，而不能共享。

为了解决构造传参和共享问题，可以**组合构造函数+原型模式**：

```
function Box(name, age) {                    //不共享的使用构造函数
    this.name = name;
    this.age = age;
    this. family = ['父亲', '母亲', '妹妹'];
};
Box.prototype = {                            //共享的使用原型模式
    constructor : Box,
    run : function () {
        return this.name + this.age + this.family;
    }
};
```

PS：这种混合模式很好的解决了传参和引用共享的大难题。是创建对象比较好的方法。

原型模式，不管你是否调用了原型中的共享方法，它都会初始化原型中的方法，并且在声明一个对象时，构造函数+原型部分让人感觉又很怪异，最好就是把构造函数和原型封装到一起。为了解决这个问题，我们可以使用**动态原型模式**。

```
function Box(name ,age) {                    //将所有信息封装到函数体内
    this.name = name;
    this.age = age;

    if (typeof this.run != 'function') {     //仅在第一次调用的初始化
        Box.prototype.run = function () {
            return this.name + this.age + '运行中...';
        };
    }
}

var box = new Box('Lee', 100);
alert(box.run());
```

当第一次调用构造函数时，run()方法发现不存在，然后初始化原型。当第二次调用，就不会初始化，并且第二次创建新对象，原型也不会再初始化了。这样及得到了封装，又实现了原型方法共享，并且属性都保持独立。

```
if (typeof this.run != 'function') {
    alert('第一次初始化');                    //测试用
    Box.prototype.run = function () {
        return this.name + this.age + '运行中...';
    };
}

var box = new Box('Lee', 100);                //第一次创建对象
alert(box.run());                             //第一次调用
alert(box.run());                             //第二次调用

var box2 = new Box('Jack', 200);              //第二次创建对象
alert(box2.run());
alert(box2.run());
```

PS：使用动态原型模式，要注意一点，不可以再使用字面量的方式重写原型，因为会切断实例和新原型之间的联系。

以上讲解了各种方式对象创建的方法，如果这几种方式都不能满足需求，可以使用一开始那种模式：寄生构造函数。

```
function Box(name, age) {
    var obj = new Object();
    obj.name = name;
    obj.age = age;
    obj.run = function () {
        return this.name + this.age + '运行中...';
    };
    return obj;
}
```

寄生构造函数，其实就是工厂模式+构造函数模式。这种模式比较通用，但不能确定对象关系，所以，在可以使用之前所说的模式时，不建议使用此模式。

在什么情况下使用寄生构造函数比较合适呢？假设要创建一个具有额外方法的引用类型。由于之前说明不建议直接 String.prototype.addstring，可以通过寄生构造的方式添加。

```
function myString(string) {
    var str = new String(string);
    str.addstring = function () {
        return this + '，被添加了！';
    };
    return str;
}

var box = new myString('Lee');                //比直接在引用原型添加要繁琐好多
alert(box.addstring());
```

在一些安全的环境中，比如禁止使用 this 和 new，这里的 this 是构造函数里不使用 this，这里的 new 是在外部实例化构造函数时不使用 new。这种创建方式叫做稳妥构造函数。

```
function Box(name , age) {
    var obj = new Object();
    obj.run = function () {
        return name + age + '运行中...';        //直接打印参数即可
    };
    return obj;
}

var box = Box('Lee', 100);                      //直接调用函数
alert(box.run());
```

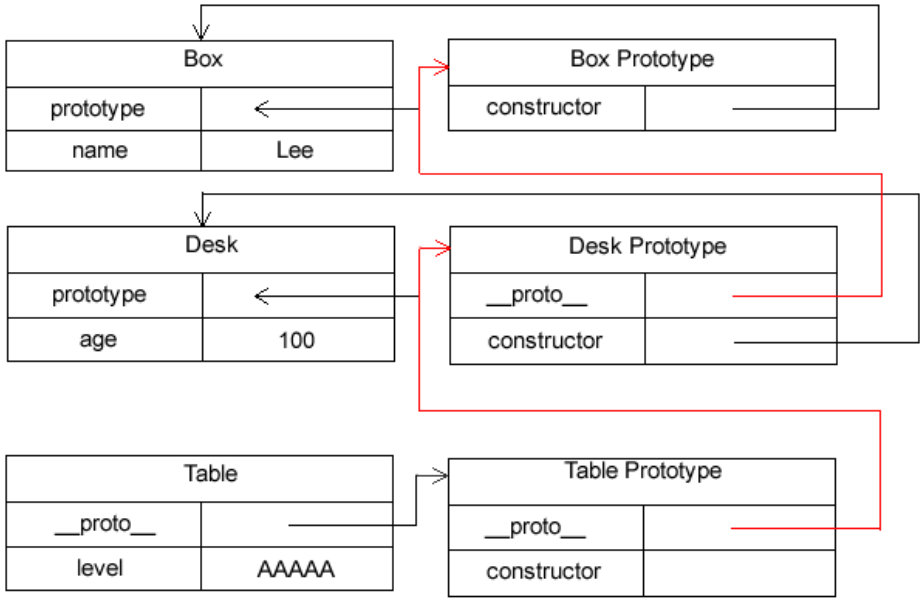
PS：稳妥构造函数和寄生类似。

四．继承

继承是面向对象中一个比较核心的概念。其他正统面向对象语言都会用两种方式实现继承：一个是接口实现，一个是继承。而 ECMAScript 只支持继承，不支持接口实现，而实现继承的方式依靠原型链完成。

```
function Box() {                                //Box 构造
    this.name = 'Lee';
}

function Desk() {                               //Desk 构造
    this.age = 100;
}

Desk.prototype = new Box();                     //Desc 继承了 Box，通过原型，形成链条

var desk = new Desk();
alert(desk.age);
alert(desk.name);                               //得到被继承的属性

function Table() {                              //Table 构造
    this.level = 'AAAAA';
}

Table.prototype = new Desk();                   //继续原型链继承

var table = new Table();
alert(table.name);                              //继承了 Box 和 Desk
```

原型链继承流程图



如果要实例化 table，那么 Desk 实例中有 age=100，原型中增加相同的属性 age=200，最后结果是多少呢？

```
Desk.prototype.age = 200;                    //实例和原型中均包含 age
```

PS：以上原型链继承还缺少一环，那就是 Obejct，所有的构造函数都继承自 Obejct。而继承 Object 是自动完成的，并不需要程序员手动继承。

经过继承后的实例，他们的从属关系会怎样呢？

```
alert(table instanceof Object);              //true
alert(desk instanceof Table);                //false，desk 是 table 的超类
alert(table instanceof Desk);                //true
alert(table instanceof Box);                 //true
```

在 JavaScript 里，被继承的函数称为超类型(父类，基类也行，其他语言叫法)，继承的函数称为子类型(子类，派生类)。继承也有之前问题，比如字面量重写原型会中断关系，使用引用类型的原型，并且子类型还无法给超类型传递参数。

为了解决引用共享和超类型无法传参的问题，我们采用一种叫借用构造函数的技术，或者成为对象冒充(伪造对象、经典继承)的技术来解决这两种问题。

```
function Box(age) {
    this.name = ['Lee', 'Jack', 'Hello']
    this.age = age;
}

function Desk(age) {
    Box.call(this, age);                     //对象冒充，给超类型传参
```

```
}

var desk = new Desk(200);
alert(desk.age);
alert(desk.name);
desk.name.push('AAA');                    //添加的新数据，只给 desk
alert(desk.name);
```

借用构造函数虽然解决了刚才两种问题，但没有原型，复用则无从谈起。所以，我们需要**原型链+借用构造函数**的模式，这种模式成为**组合继承**。

```
function Box(age) {
    this.name = ['Lee', 'Jack', 'Hello']
    this.age = age;
}

Box.prototype.run = function () {
    return this.name + this.age;
};

function Desk(age) {
    Box.call(this, age);                  //对象冒充
}

Desk.prototype = new Box();               //原型链继承

var desk = new Desk(100);
alert(desk.run());
```

还有一种继承模式叫做：**原型式继承**；这种继承借助原型并基于已有的对象创建新对象，同时还不必因此创建自定义类型。

```
function obj(o) {                         //传递一个字面量函数
    function F() {}                       //创建一个构造函数
    F.prototype = o;                      //把字面量函数赋值给构造函数的原型
    return new F();                       //最终返回出实例化的构造函数
}

var box = {                               //字面量对象
    name : 'Lee',
    arr : ['哥哥','妹妹','姐姐']
};

var box1 = obj(box);                      //传递
alert(box1.name);
```

```
box1.name = 'Jack';
alert(box1.name);

alert(box1.arr);
box1.arr.push('父母');
alert(box1.arr);

var box2 = obj(box);                    //传递
alert(box2.name);
alert(box2.arr);                        //引用类型共享了
```

寄生式继承把原型式+工厂模式结合而来，目的是为了封装创建对象的过程。

```
function create(o) {                    //封装创建过程
    var f= obj(o);
    f.run = function () {
        return this.arr;                //同样，会共享引用
    };
    return f;
}
```

组合式继承是 JavaScript 最常用的继承模式；但，组合式继承也有一点小问题，就是超类型在使用过程中会被调用两次：一次是创建子类型的时候，另一次是在子类型构造函数的内部。

```
function Box(name) {
    this.name = name;
    this.arr = ['哥哥','妹妹','父母'];
}

Box.prototype.run = function () {
    return this.name;
};

function Desk(name, age) {
    Box.call(this, name);               //第二次调用 Box
    this.age = age;
}

Desk.prototype = new Box();             //第一次调用 Box
```

以上代码是之前的组合继承，那么**寄生组合继承**，解决了两次调用的问题。

```
function obj(o) {
    function F() {}
    F.prototype = o;
```

```
        return new F();
}

function create(box, desk) {
        var f = obj(box.prototype);
        f.constructor = desk;
        desk.prototype = f;
}

function Box(name) {
        this.name = name;
        this.arr = ['哥哥','妹妹','父母'];
}

Box.prototype.run = function () {
        return this.name;
};

function Desk(name, age) {
        Box.call(this, name);
        this.age = age;
}

inPrototype(Box, Desk);                    //通过这里实现继承

var desk = new Desk('Lee',100);
desk.arr.push('姐姐');
alert(desk.arr);
alert(desk.run());                         //只共享了方法

var desk2 = new Desk('Jack', 200);
alert(desk2.arr);                          //引用问题解决
```

感谢收看本次教程！

本课程是由北风网(ibeifeng.com)

瓢城 Web 俱乐部(yc60.com)联合提供：

本次主讲老师：李炎恢

我的博客：hi.baidu.com/李炎恢/

我的邮件：yc60.com@gmail.com