javascript学习--2017-11-18：继承

笔记本：	javascript		
创建时间：	2017/11/18 9:57	**更新时间：**	2017/11/19 20:10
作者：	xiethan		

继承方式：接口继承 & 实现继承（javascript可实现）

```
//继承
function Mother(){
this.fair="red";
}
Mother.prototype.getFair=function(){
return this.fair;
}

function Son(){
this.sex="boy";
}
//继承要先继承，如果先定义getSex属性，会被覆盖。
Son.prototype=new Mother();
Son.prototype.getSex=function(){
return this.sex;
}
var human=new Son();
console.log(human.getSex());
console.log(human);
console.log(Mother);//未实例化的话，会返回function
```

```
function SuperType(){
    this.property = true;
}
```

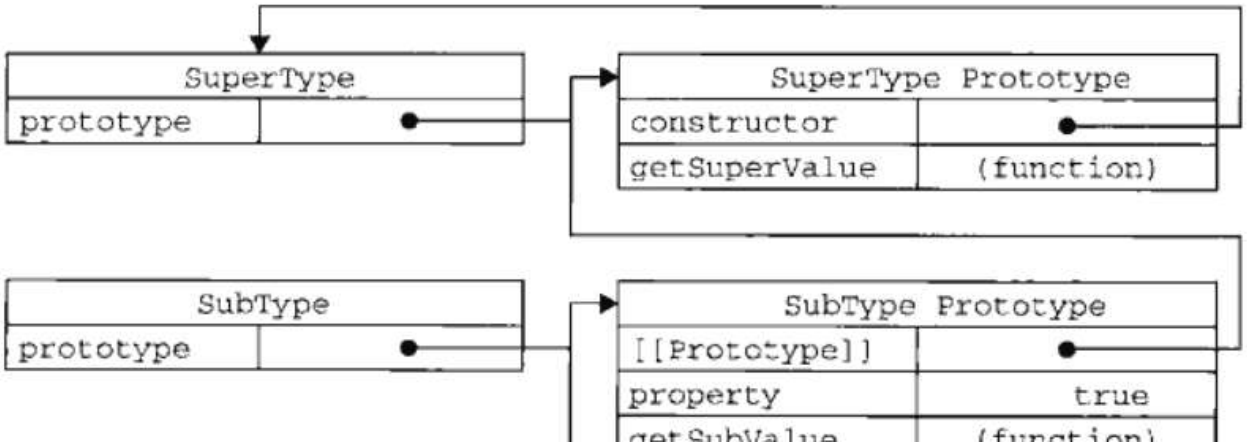
```
SuperType.prototype.getSuperValue = function(){
    return this.property;
};

function SubType(){
    this.subproperty = false;
}

//继承了 SuperType
SubType.prototype = new SuperType();

SubType.prototype.getSubValue = function (){
    return this.subproperty;
};

var instance = new SubType();
alert(instance.getSuperValue());        //true
```



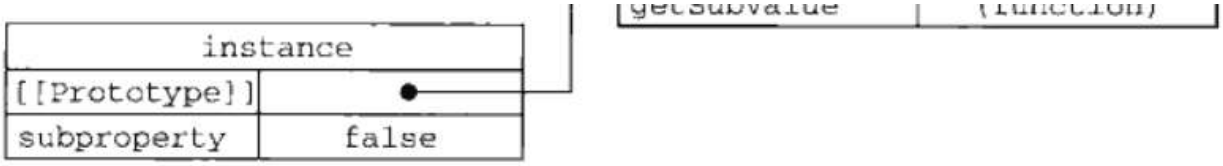
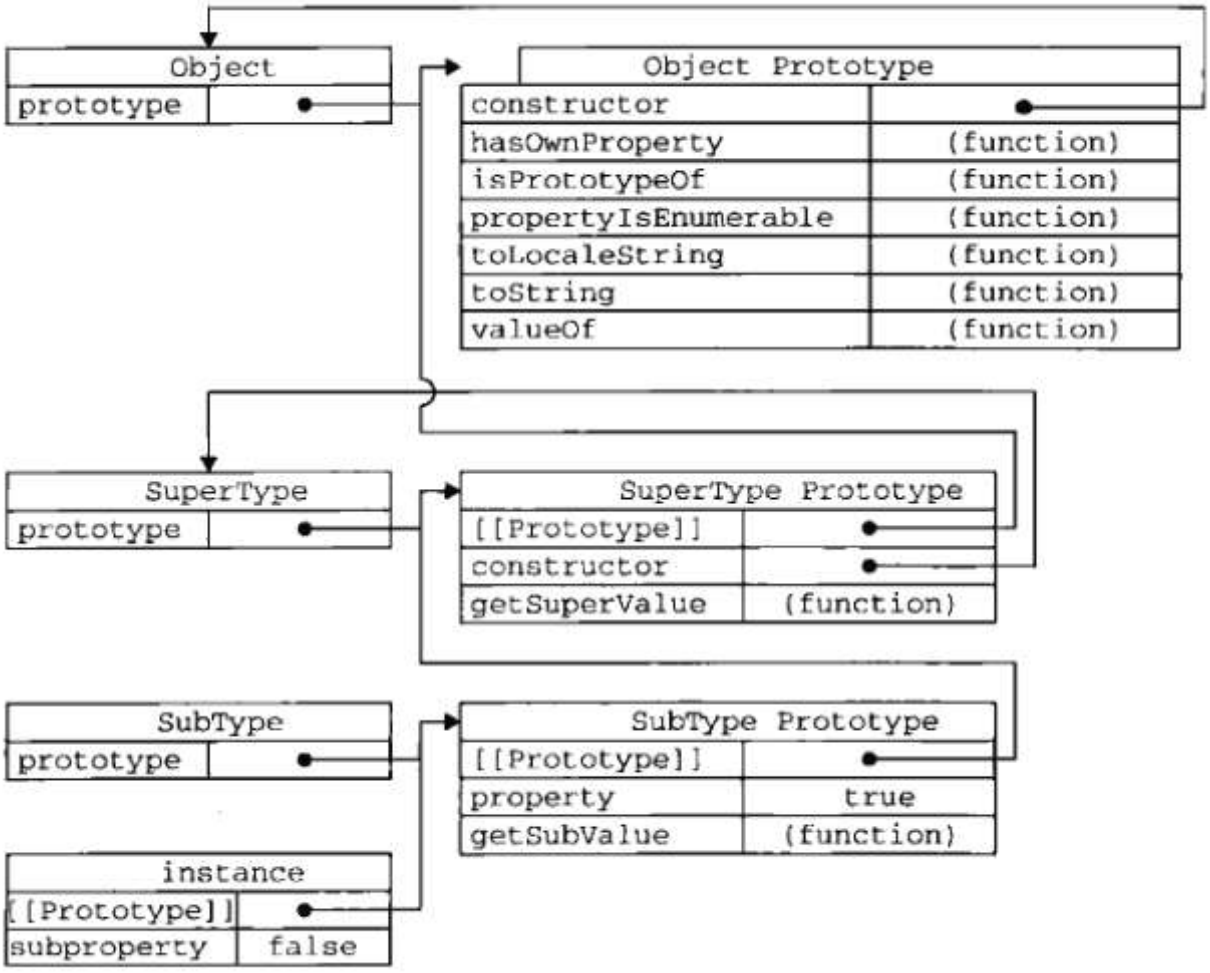


图 6-4

原型搜索机制：当我们要读取某个实例属性，搜索顺序
搜索实例
搜索son的prototype
搜索mother的prototype（包括contructor [[prototype]] 自己定义的prototype）
记住：所以函数默认原型都是object的实例




```
//确认关系
console.log(human instanceof Son);//true
console.log(human instanceof Mother);
```

原型链的问题：

```
//继承
function Mother(){
this.fair="red";
this.color="color";
this.arr=['one','two',"three"];
}
Mother.prototype.getFair=function(){
return this.fair;
}

function Son(){
this.sex="boy";
this.fair="black";
this.arr=['one','two'];
```

```
}
//继承要先继承，如果先定义getSex属性，会被覆盖。
Son.prototype=new Mother();
Son.prototype.getSex=function(){
return this.sex;
}
Son.prototype.arr=['one'];
var human=new Son();
console.log(human.getSex());
console.log(human);
console.log(new Mother);//未实例化的话，会返回function
//确认关系
console.log(human instanceof Son);
console.log(human instanceof Mother);

function GrandSon(){
this.sex="girl";
// this.fair="gard";
}
//继承要先继承，如果先定义getSex属性，会被覆盖。
GrandSon.prototype=new Son();
GrandSon.prototype.getSex=function(){
return this.arr;
}
var gSon=new GrandSon();
var gSon2=new GrandSon();
console.log(gSon.getSex());//["one", "two"]
gSon.arr.push("mode");
console.log(gSon2.getSex()); //["one", "two", "mode"]
```

通过以上代码可以发现：包含引用类型的值的原型属性会被所有实例共享。而也正是为什么要放在构造函数中，而不是在原型对象中定义属性的原因，在通过原型来实现继承时，原型实际上会变成另一个类型的实例，于是，原先的实例属性，也就顺利成章的变成了现在的原型属性了。

当mother构造函数定义了一个arr数组后（引用类型值），mother的每个实例都有各自各的的数组。当通过原型链继承后，son.prototype就变成了mother第一个实例，，因此它也拥有一个arr属性，，这就跟son.prototype.arr属性一样，但结果呢，son的所有实例都会共享这个一个arr属性

为了解决上述的包含引用类型带来问题时：借用一种

构造函数方法，伪造对象，经典继承

```
//借用构造函数方法，伪造对象，经典继承
function father(){
this.color=["red","black"];
}
function boy(){
//继承了father
father.call(this);
}

var boy1=new boy();
var boy2=new boy();
boy1.color.push("bark");
console.log(boy1.color);
console.log(boy2.color);
// ["red", "black", "bark"]0: "red"1: "black"2: "bark"length: 3__proto__: Array[0]
// ["red", "black"]
```

原理：那行代码“借调”了那个超类型的构造函数，通过call(或者apply）,实际上是在新创建的boy实例环境下调用了father的构造函数。
这样一来，就会在新boy对象上执行father()函数时，会执行father函数上定义了所有对象传送话代码。这个每个boy每个实例都具有color的属性了。

组合继承：最常用的继承模式

```
function father(name){
this.name=name;
this.color=["1","2"];
}
father.prototype.sayName=function(){
```

```
console.log(this.name);
}

function boy(name,age){
//继承了father
father.call(this,name);
this.age=age;
}
boy.prototype=new father();
boy.prototype.sayAge=function(){
console.log(this.age);
}
var boy1=new boy('ethan','21');
var boy2=new boy('ethan2','22');
boy1.color.push("bark");
console.log(boy1.color);
boy1.sayAge();
boy1.sayName();
console.log(boy2.color);
boy2.sayAge();
boy2.sayName();
/*结果为:
demo.html:88 ["1", "2", "bark"]
demo.html:89 21
demo.html:79 ethan
demo.html:97 ["1", "2"]0: "1"1: "2"length: 2__proto__: Array[0]
demo.html:89 22
demo.html:79 ethan2
*/
```

原型式继承

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

在object()函数内部，先创建了一个临时性的构造函数，然后将传入的对象作为这个构造函数的原型，最后返回了这个临时类型的一个新实例。从本质上讲，object()对传入其中的对象执行了一次

浅复制。来看下面的例子。

```
var person = {
    name: "Nicholas",
    friends: ["Shelby", "Court", "Van"]
};

var anotherPerson = object(person);
anotherPerson.name = "Greg";
anotherPerson.friends.push("Rob");

var yetAnotherPerson = object(person);
yetAnotherPerson.name = "Linda";
yetAnotherPerson.friends.push("Barbie");

alert(person.friends);   //"Shelby,Court,Van,Rob,Barbie"
```

es5,通过使用object.create规范化了的原型式继承。

ECMAScript 5 通过新增 Object.create()方法规范化了原型式继承。这个方法接收两个参数：一个用作新对象原型的对象和（可选的）一个为新对象定义额外属性的对象。在传入一个参数的情况下，Object.create()与 object()方法的行为相同。

```
var person = {
    name: "Nicholas",
    friends: ["Shelby", "Court", "Van"]
};

var anotherPerson = Object.create(person);
anotherPerson.name = "Greg";
anotherPerson.friends.push("Rob");

var yetAnotherPerson = Object.create(person);
yetAnotherPerson.name = "Linda";
yetAnotherPerson.friends.push("Barbie");

alert(person.friends); //"Shelby,Court,Van,Rob,Barbie"
```

寄生继承：

6.3.5 寄生式继承

寄生式（parasitic）继承是与原型式继承紧密相关的一种思路，并且同样也是由克罗克福德推而广之的。寄生式继承的思路与寄生构造函数和工厂模式类似，即创建一个仅用于封装继承过程的函数，该函数在内部以某种方式来增强对象，最后再像真地是它做了所有工作一样返回对象。以下代码示范了寄生式继承模式。

```
function createAnother(original){
    var clone = object(original);   //通过调用函数创建一个新对象
    clone.sayHi = function(){       //以某种方式来增强这个对象
        alert("hi");
    };
    return clone;                   //返回这个对象
}
```

在这个例子中，createAnother()函数接收了一个参数，也就是将要作为新对象基础的对象。然后，把这个对象（original）传递给 object()函数，将返回的结果赋值给 clone。再为 clone 对象添加一个新方法 sayHi()，最后返回 clone 对象。可以像下面这样来使用 createAnother()函数：

```
var person = {
    name: "Nicholas",
    friends: ["Shelby", "Court", "Van"]
};

var anotherPerson = createAnother(person);
anotherPerson.sayHi(); //"hi"
```

这个例子中的代码基于 person 返回了一个新对象——anotherPerson。新对象不仅具有 person 的所有属性和方法，而且还有自己的 sayHi()方法。

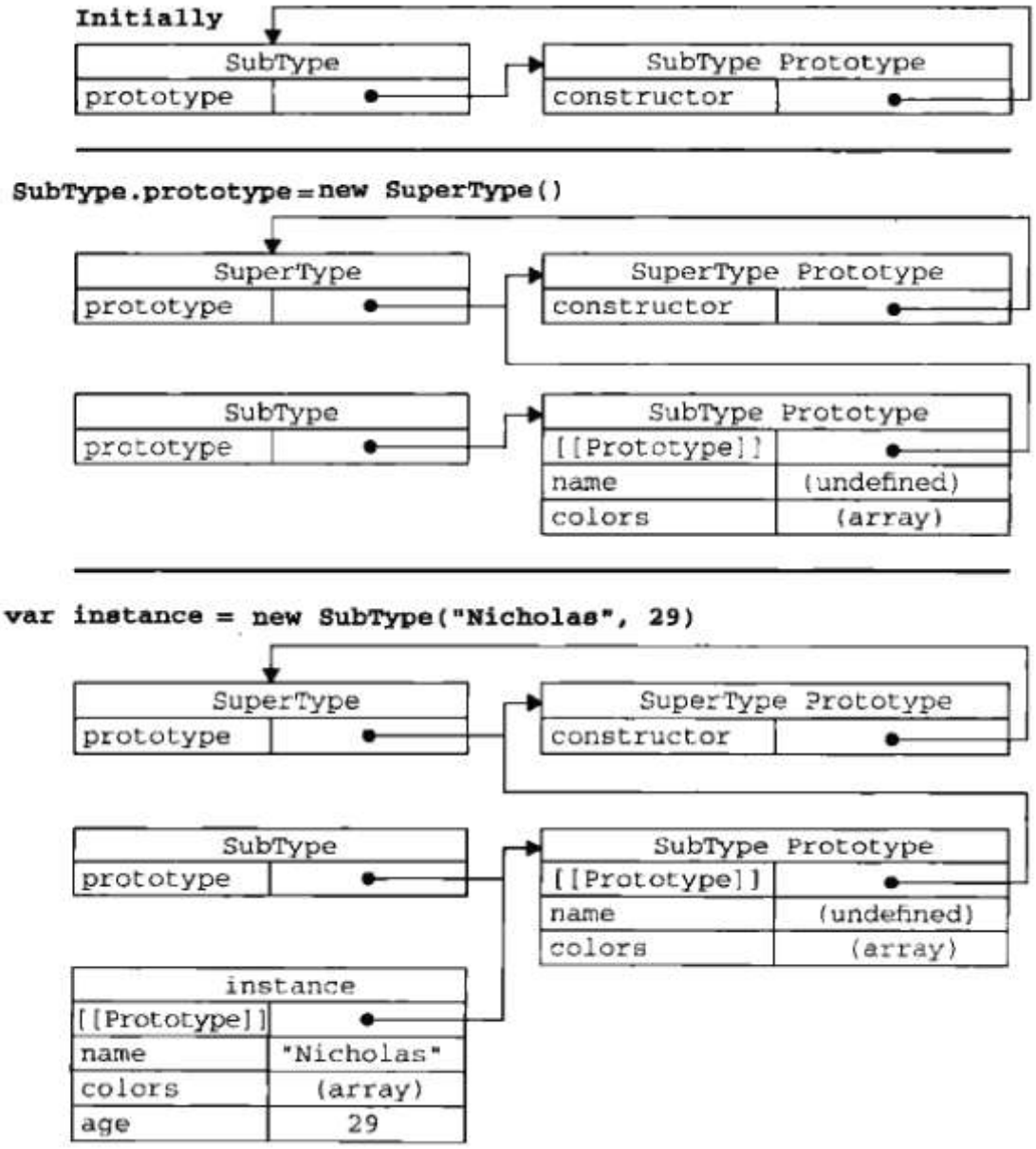
在主要考虑对象而不是自定义类型和构造函数的情况下，寄生式继承也是一种有用的模式。前面示范继承模式时使用的 object()函数不是必需的；任何能够返回新对象的函数都适用于此模式。

使用寄生式继承来为对象添加函数，会由于不能做到函数复用而降低效率；这一点与构造函数模式类似。

```
//寄生组合式继承
function Super(name){
this.name=name;
```

```
this.colors=['cat','dog'];
}
Super.prototype.sayName=function(){
console.log(this.colors);
}
function Sub(name,age){
Super.call(this,name);//调用了一次super的构造代码2
thisf.age=age;
}
Sub.prototype=new Super();//调用了一次super的构造代码1
sub.prototype.constructor=Sub;
Sub.prototype.age=function(){
console.log(this.age);
```

第一次调用是构造函数时，sub.prototype会得到两个属性：name colors 它们都是super的实例属性。
当调用sub的构造函数时，又会调用一次super的构造函数。这一次又在新对象上创建实例属性name& colors
原理图：



寄生组合式继承：通过借用构造函数来继承属性，通过原型链的混成方法来继承方法

```
// 寄生组合式的继承，最简单的实现形势。
function inheritPrototype(son,father){
var prototype=object(father.prototype);
prototype.constructor=son;
son.prototype=prototype;
}


function Super(name){
this.name=name;
```

```
this.colors=['cat','dog'];
}
Super.prototype.sayName=function(){
console.log(this.colors);
}
function Sub(name,age){
Super.call(this,name);//调用了一次super的构造代码2
thisf.age=age;
}
// Sub.prototype=new Super();//调用了一次super的构造代码1
//sub.prototype.constructor=Sub;
inheritPrototype(Sub,Super);
Sub.prototype.age=function(){
console.log(this.age);
```

上面的函数接收两个参数，子类型构造函数和超类型的构造函数。在函数内部，第一步是创建超类型的一个原型副本，第二步是为创建的副本添加添加constructor属性，从而补因重写原型而失去的construct属性，最后一步，将新建的副本赋予到子类型的原型，这样，我们就可以调用inheritPrototype()函数的语句，去替换前面例子中为子类型原型赋值的语句