6.3.6　寄生组合式继承

前面说过，组合继承是 JavaScript 最常用的继承模式；不过，它也有自己的不足。组合继承最大的问题就是无论什么情况下，都会调用两次超类型构造函数：一次是在创建子类型原型的时候，另一次是在子类型构造函数内部。没错，子类型最终会包含超类型对象的全部实例属性，但我们不得不在调用子类型构造函数时重写这些属性。再来看一看下面组合继承的例子。

```
function SuperType(name){
    this.name = name;
    this.colors = ["red", "blue", "green"];
}

SuperType.prototype.sayName = function(){
    alert(this.name);
};

function SubType(name, age){
    SuperType.call(this, name);          //第二次调用 SuperType()

    this.age = age;
}

SubType.prototype = new SuperType();     //第一次调用 SuperType()
SubType.prototype.constructor = SubType;
SubType.prototype.sayAge = function(){
    alert(this.age);
};
```

加粗字体的行中是调用 SuperType 构造函数的代码。在第一次调用 SuperType 构造函数时，SubType.prototype 会得到两个属性：name 和 colors；它们都是 SuperType 的实例属性，只不过现在位于 SubType 的原型中。当调用 SubType 构造函数时，又会调用一次 SuperType 构造函数，这一次又在新对象上创建了实例属性 name 和 colors。于是，这两个属性就屏蔽了原型中的两个同名属性。图 6-6 展示了上述过程。

如图 6-6 所示，有两组 name 和 colors 属性：一组在实例上，一组在 SubType 原型中。这就是调用两次 SuperType 构造函数的结果。好在我们已经找到了解决这个问题方法——寄生组合式继承。

所谓寄生组合式继承，即通过借用构造函数来继承属性，通过原型链的混成形式来继承方法。其背后的基本思路是：不必为了指定子类型的原型而调用超类型的构造函数，我们所需要的无非就是超类型原型的一个副本而已。本质上，就是使用寄生式继承来继承超类型的原型，然后再将结果指定给子类型的原型。寄生组合式继承的基本模式如下所示。

```
function inheritPrototype(subType, superType){
    var prototype = object(superType.prototype);      //创建对象
    prototype.constructor = subType;                  //增强对象
    subType.prototype = prototype;                    //指定对象
}
```

这个示例中的 inheritPrototype()函数实现了寄生组合式继承的最简单形式。这个函数接收两个参数：子类型构造函数和超类型构造函数。在函数内部，第一步是创建超类型原型的一个副本。第二步是为创建的副本添加 constructor 属性，从而弥补因重写原型而失去的默认的 constructor 属性。最后一步，将新创建的对象（即副本）赋值给子类型的原型。这样，我们就可以用调用 inherit-Prototype()函数的语句，去替换前面例子中为子类型原型赋值的语句了，例如：