**第六章 面向对象的程序设计**

本章内容

* 理解对象属性
* 理解并创建对象
* 理解继承

代码中加粗的那一行代码“借调”了超类型的构造函数。通过使用call()方法（或apply()方法也可以），我们实际上是在（未来将要）新创建的SubType实例的环境下调用了SuperType构造函数。这样一来，就会在新SubType对象上执行SuperType()函数中定义的所有对象初始化代码。结果，SubType的每个实例就都会具有自己的colors属性的副本了。

1.传递参数

相对于原型链而言，借用构造函数有一个很大的优势，即可以在子类型构造函数中向超类型构造函数传递参数。看下面这个例子。

1.         function SuperType(name) {

2.             this.name = name;

3.         }

4.

5.         function SubType() {

6.             SuperType.call(this, "xieshengyun");

7.             this.age = 33;

8.         }

9.

10.         var instance = new SubType();

11.

12.         alert(instance.name);

13.         alert(instance.age);

ConstructorStealingExample01.html

在以上代码中的SuperType只接受一个参数name，该参数会直接赋给一个属性。在SubType构造函数内部调用SuperType构造函数时，实际上是为SubType的实力设置了name属性。为了确保SuperType构造函数不会重写子类型的属性，可以在调用超类构造函数后，再添加应该在子类型中定义的属性。

2.借用构造函数的问题

如果仅仅是借用构造函数，那么也将无法避免构造函数模式存在的问题---方法都在构造函数中定义，因此函数复用就无从谈起了。而且，在超类型的原型中定义的方法，对子类而言也是不可见的，结果所有类型都只能使用构造函数模式。考虑到这些问题，借用构造函数的技术也是很少单独使用的。

6.3.3 组合继承

组合继承（combination inheritance），有时候也叫做伪经典继承，指的是将原型链与借用构造函数的技术组合到一块，从而发挥二者之长的一种继承模式。其背后的思路是使用原型链实现对原型属性和方法的继承，而通过借用构造函数来实现实例属性的继承。这样，即通过在原型上定义方法实现了函数服用，又能够保证每个实例都有它自己的属性。下面来看一下例子：

1.         function SuperType(name) {

2.             this.name = name;

3.             this.colors = ["read", "blue", "green"];

4.         }

5.

6.         SuperType.prototype.sayName = function () {

7.             alert(this.name);

8.         };

9.

10.         function SubType(name, age) {

11.             // 继承属性

12.             SuperType.call(this, "inheritence property:" + name);

13.

14.             this.age = age;

15.         }

16.

17.         // 继承方法

18.         SubType.prototype = new SuperType();

19.         SubType.prototype.constructor = SubType;

20.         SubType.prototype.sayAge = function () {

21.             alert(this.age);

22.         }

23.

24.         var instance1 = new SubType("Nicholas", 33);

25.         instance1.colors.push("yellow");

26.         alert(instance1.colors); // "red,blue,green,yellow"

27.         instance1.sayName(); // Nicholas

28.         instance1.sayAge(); // 33

29.

30.         var instance2 = new SubType("Greg", 33);

31.         alert(instance2.colors); // "red,blue,green"

32.         instance2.sayName(); // "Greg"

33.         instance2.sayAge(); // 33

CombinationInheritenceExample01.html

在这个例子中，SuperType构造函数定义了两个属性：name和colors。SuperType的原型定义了一个方法sayName()。SubType构造函数在调用SuperType构造函数时传入了name参数，紧接着又定义了它自己的属性age。然后，将SuperType的实例赋值给SubType的原型，然后又在该新原型上定义了方法sayAge()。这样一来，就可以让两个不同的SubType实例即分别拥有自己的属性---包括colors属性，又可以使用相同的方法了。

组合继承避免了原型链和借用构造函数的缺陷，融合了它们的优点，成为JavaScript中最常用的集成模式。而且，instanceof和isPrototypeOf()也能够识别基于组合继承创建的对象。

6.3.4 原型式继承

道格拉斯.可罗克福德在2006年写了一篇文章，题为Prototypal Inheritance in JavaScript（JavaScript中的原型继承）。在这片文章中，他介绍了一种实现继承的方法，这种方法并没有使用严格意义上的构造函数。他的想法是借助原型可以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。为了达到这个目的，他给出了如下函数。

1.         function object(o) {

2.             function F() {}

3.             F.prototype = o;

4.             return new F();

5.         }

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。

1.         function object(o) {

2.             function F() {}

3.             F.prototype = o;

4.             return new F();

5.         }

6.

7.         var person = {

8.             name: "Nichols",

9.             friends: ["Shelly", "Court", "Van"]

10.         }

11.

12.         var anotherPerson = object(person);

13.         anotherPerson.name = "Greg";

14.         anotherPerson.friends.push("Rob");

15.

16.         var yetAnotherPerson = object(person);

17.         yetAnotherPerson.name = "Linda";

18.         yetAnotherPerson.friends.push("Barbie");

19.

20.         alert(person.friends);

PrototypalInheritanceExample01.html

克罗克福德主张的这种原型式继承，要求你必须有一个对象作为另一个对象的基础。如果有这么一个对象的话，可以把它传递给object()函数，然后再根据具体需求对得到的对象加以修改即可。在这个例子中，可以作为另一个对象基础的是person对象，于是我们把它传入到object()函数中，然后该函数就会返回一个新对象。这个新对象将person作为原型。所以它的原型中就包含一个基本类型值属性和一个引用类型值属性。这意味着person.friends不仅属于person所有，而且也会被anotherPerson及yetAnotherPerson共享。实际上，这就相当于又创建了person对象的两个副本。

ECMAScript 5通过新增Object.create()方法规范化了原型式继承。这个方法接受两个参数：一个用作新对象原型的对象（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。

  1.     var person = {

2.         name: "Nichols",

3.         friends: ["Shelly", "Court", "Van"],

4.       };

5.

6.       var anotherPerson = Object.create(person);

7.       anotherPerson.name = "Greg";

8.       anotherPerson.friends.push("Rob");

9.

10.       var yetAnotherPerson = Object.create(person);

11.       yetAnotherPerson.name = "Linda";

12.       yetAnotherPerson.friends.push("Barbie");

13.

14.       alert(person.friends);

PrototypalInheritanceExample02.html

Object.create()方法的第二个参数与Object.defineProperties()方法的第二个参数格式相同：每个属性都是通过自己的描述符定义的。以这种方式指定的任何属性都会覆盖原型对象上的同名属性。例如：

1.       var person = {

2.         name: "Nichols",

3.         friends: ["Shelly", "Court", "Van"],

4.       };

5.

6.       var anotherPerson = Object.create(person, {

7.         name: {

8.           value: "Greg"

9.         }

10.       });

11.

12.       alert(anotherPerson.name);

PrototypalInheritanceExample03.html

支持Object.create()方法的浏览器有IE 9+、Firefox 4+、Safari 5+、Opera 12+和chrome。

在没有必要兴师动众地创建构造函数，而只想让一个对象与另一个对象保持类似的情况下，原型式继承是完全可以胜任的。不过别忘了，包含引用类型值的属性始终都会共享相应的值，就像使用原型模式一样。

6.3.5 寄生式继承

寄生式继承（parasitic）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，最后再像真的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。

1.       function createAnother(original) {

2.         var clone = object(original); // （原型式继承）通过调用函数创建一个新对象

3.         clone.sayHi = function () {   // 以某种方式来增强这个对象

5.           alert("Hi");

6.         };

7.         return clone; // 返回这个对象

8.       }

在这个例子中，createAnother()函数接受了一个参数，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：

1.       function object(o) {

2.         function F() {}

3.         F.prototype = o;

4.         return new F();

5.       }

6.

7.       function createAnother(original) {

8.         var clone = object(original); // 通过调用函数创建一个新对象

9.         clone.sayHi = function () {

10.           // 以某种方式来增强这个对象

11.           alert("Hi");

12.         };

13.         return clone; // 返回这个对象

14.       }

15.

16.       var person = {

17.         name: "Nichols",

18.         friends: ["Shelly", "Court", "Van"]

19.       }

20.

21.       var anotherPerson = createAnother(person);

22.       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()函数不是必须的；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。

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。

1.       function SuperType(name) {

2.         this.name = name;

3.         this.colors = ["red", "blue", "green"];

4.       }

5.

6.       SuperType.prototype.sayName = function () {

7.         alert(this.name);

8.       }

9.

10.       function SubType(name, age) {

11.         **SuperType.call(this, name);** // 第二次调用 SuperType

12.         this.age = age;

13.       }

14.

15.       **SubType.prototype = new SuperType();** // 第一次调用SuperType

16.       SubType.prototype.constructor = SubType;

17.       SubType.prototype.sayAge = function () {

18.         alert(this.age);

19.       }

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。

如果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的方法---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，通过原型链的混合形式来继承方法。其背后的基本思路是：不必为了指定子类型的2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。

1.       function inheritPrototype(subType, superType) {

2.         var prototype = Object(superType.prototype); // 创建对象

3.         prototype.constructor = subType;  // 增强对象

4.         subType.prototype = prototype;  // 指定对象

5.       }

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。这个函数接受两个参数：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了，例如：

1.       function inheritPrototype(subType, superType) {

2.         var prototype = new Object(superType.prototype);

3.

4.         prototype.constructor = subType;

5.

6.         subType.prototype = prototype;

7.       }

8.

9.       function SuperType(name) {

10.         this.name = name;

11.         this.colors = ["red", "blue", "green"];

12.       }

13.

14.       SuperType.prototype.sayName = function () {

15.         alert(this.name);

16.       }

17.

18.       function SubType(name, age) {

19.         SuperType.call(this, name); // 第二次调用 SuperType

20.         this.age = age;

21.       }

22.

23.       inheritPrototype(SubType, SuperType);

24.

25.       SubType.prototype.sayAge = function () {

26.         alert(this.age);

27.       }

28.

29.       var instance = new SubType("Nichole", 39);

30.       instance.sayAge();

ParasiticCombinationInheritanceExample01.html

图示

描述已自动生成

这个例子的高效体现在它只调用了一次SuperType构造函数，并且因此避免了在SubType.prototype上面创建不必要的，多余的属性。于此同时，原型链还能保持不变；因此，还能够正常使用instanceof和isPrototypeOf()。开发人员普遍认为寄生组合式继承是引用类型最理想的继承范式。

YUI的YAHOO.lang.extend()方法采用了寄生组合继承，从而让这种模式首次出现在了一个应用非常广泛的JavaScript库中。要了解有关YUI的更多信息，请访问<http://developer.yahoo.com/yui/>。

6.4 小结

ECMAScript支持面向对象（OO）编程，但不能使用类或者接口。对象可以在代码执行过程中创建和增强，因此具有动态性而非严格定义的实体。在没有类的情况下，可以采用下列模式创建对象。

* 工厂模式，使用简单的函数创建对象，为对象添加属性和方法，然后返回对象。这个模式后来被构造函数模式所取代。
* 构造函数模式，可以创建自定义引用类型，可以像创建内置对象实例一样使用new操作符。不过，构造函数模式也有缺点，即它的每个成员都无法得到复用，包括函数。由于函数可以不局限于任何对象（即与对象具有松散耦合的特点），因此没有理由不在多个对象间共享函数。
* 原型模式，使用构造函数的prototype属性来指定那些因该共享的属性和方法。组合使用构造函数模式和原型模式时，使用构造函数定义实例属性，而使用原型定义共享的属性和方法。

JavaScript主要通过原型链实现继承。原型链的构建是通过将一个类型的实例赋值给另一个构造函数的原型实现的。这样，子类型就能够访问超类型的所有属性和方法，这一点与基于类的继承很相似。原型链的问题是对象实例共享所有继承的属性和方法，因此不适宜单独使用。解决这个问题的技术是“借用构造函数”，即在子类型构造函数的内部调用超类型构造函数。这样就可以做到每个实例都具有自己的属性，同时还能保证只使用构造函数模式来定义类型。使用最多的继承模式是组合继承，这种模式使用原型链继承共享的属性和方法，而通过借用构造函数继承实例属性。

此外，还存在下列可供选择的继承模式。

* 原型式继承，可以在不必预先定义构造函数的情况下实现继承，其本质是执行对给定对象的浅复制。而复制得到的副本还可以得到进一步改造。
* 寄生式继承，与原型式继承非常相似，也是基于某个对象或某些信息创建一个对象，然后增强对象，最后返回对象。为了解决组合继承模式由于多次调用超类型构造函数而导致的低效率问题，可以将这个模式与组合继承一起使用。
* 寄生组合式继承，集寄生式继承和组合继承的有点与一身，是实现基于类型继承的最有效方法。