# 面向对象的程序设计

面向对象（Object-Oriented，OO）的语言有一个标志，那就是它们都有类的概念，而通过类可以创建任意多个具有相同属性和方法的对象。前面提到过，ECMAScript 中没有类的概念，因此它的对象也与基于类的语言中的对象有所不同。

ECMA-262 把对象定义为：“**无序属性的集合，其属性可以包含基本值、对象或者函数**。”严

格来讲，这就相当于说对象是一组没有特定顺序的值。对象的每个属性或方法都有一个名字，而每个名字都映射到一个值。正因为这样（以及其他将要讨论的原因），我们可以把 ECMAScript 的对象想象成散列表：无非就是一组名值对，其中值可以是数据或函数。

## 理解对象

创建自定义对象的最简单方式就是创建一个 Object 的实例，然后再为它添加属性和方法，如下所示。

var person = new Object();

person.name = "Nicholas";

person.age = 29;

person.job = "Software Engineer";

person.sayName = function(){

alert(this.name);

};

上面的例子创建了一个名为 person 的对象，并为它添加了三个属性（ name 、 age 和 job ）和一个方法（ sayName() ）。其中， sayName() 方法用于显示 this.name （将被解析为 person.name ）的值。

早期的 JavaScript开发人员经常使用这个模式创建新对象。几年后，对象字面量成为创建这种对象的首选模式。前面的例子用对象字面量语法可以写成这样：

var person = {

name: "Nicholas",

age: 29,

job: "Software Engineer",

sayName: function(){

alert(this.name);

}

};

这个例子中的 person 对象与前面例子中的 person 对象是一样的，都有相同的属性和方法。这些属性在创建时都带有一些特征值（characteristic），JavaScript通过这些特征值来定义它们的行为。

## 创建对象

虽然 Object 构造函数或对象字面量都可以用来创建单个对象，但这些方式有个明显的缺点：使用同一个接口创建很多对象，会产生大量的重复代码。为解决这个问题，人们开始使用工厂模式的一种变体。

### 工厂模式

工厂模式是软件工程领域一种广为人知的设计模式，这种模式抽象了创建具体对象的过程。考虑到在 ECMAScript5 中无法创建类，开发人员就发明了一种函数，用函数来封装以特定接口创建对象的细节，如下面的例子所示。

function createPerson(name, age, job){

var o = new Object();

o.name = name;

o.age = age;

o.job = job;

o.sayName = function(){

alert(this.name);

};

return o;

}

var person1 = createPerson("Nicholas", 29, "Software Engineer");

var person2 = createPerson("Greg", 27, "Doctor");

函数 createPerson() 能够根据接受的参数来构建一个包含所有必要信息的 Person 对象。可以无数次地调用这个函数，而每次它都会返回一个包含三个属性一个方法的对象。工厂模式虽然解决了创建多个相似对象的问题，但却没有解决对象识别的问题（即怎样知道一个对象的类型）。随着 JavaScript的发展，又一个新模式出现了。

### 构造函数模式

ECMAScript 中的构造函数可用来创建特定类型的对象。像 Object 和 Array 这样的原生构造函数，在运行时会自动出现在执行环境中。此外，也可以创建自定义的构造函数，从而定义自定义对象类型的属性和方法。例如，可以使用构造函数模式将前面的例子重写如下。

function Person(name, age, job){

this.name = name;

this.age = age;

this.job = job;

this.sayName = function(){

alert(this.name);

};

}

var person1 = new Person("Nicholas", 29, "Software Engineer");

var person2 = new Person("Greg", 27, "Doctor");

在这个例子中， Person() 函数取代了 createPerson() 函数。我们注意到， Person() 中的代码除了与 createPerson() 中相同的部分外，还存在以下不同之处：

*  没有显式地创建对象；
*  直接将属性和方法赋给了 this 对象；
*  没有 return 语句。

此外，还应该注意到函数名 Person 使用的是大写字母 P。按照惯例，构造函数始终都应该以一个大写字母开头，而非构造函数则应该以一个小写字母开头。这个做法借鉴自其他 OO 语言，主要是为了区别于 ECMAScript 中的其他函数；因为构造函数本身也是函数，只不过可以用来创建对象而已。要创建 Person 的新实例，必须使用 new 操作符。以这种方式调用构造函数实际上会经历以下 4个步骤：

(1) 创建一个新对象；

(2) 将构造函数的作用域赋给新对象（因此 this 就指向了这个新对象）；

(3) 执行构造函数中的代码（为这个新对象添加属性）；

(4) 返回新对象。

alert(person1 instanceof Object); //true

alert(person1 instanceof Person); //true

alert(person2 instanceof Object); //true

alert(person2 instanceof Person); //true

创建自定义的构造函数意味着将来可以将它的实例标识为一种特定的类型；而这正是构造函数模式胜过工厂模式的地方。在这个例子中， person1 和 person2 之所以同时是 Object 的实例，是因为所有对象均继承自 Object。

**1. 将构造函数当作函数**

构造函数与其他函数的唯一区别，就在于调用它们的方式不同。不过，构造函数毕竟也是函数，不存在定义构造函数的特殊语法。任何函数，只要通过 new 操作符来调用，那它就可以作为构造函数；而任何函数，如果不通过 new 操作符来调用，那它跟普通函数也不会有什么两样。

**2. 构造函数的问题**

构造函数模式虽然好用，但也并非没有缺点。使用构造函数的主要问题，就是每个方法都要在每个实例上重新创建一遍。在前面的例子中， person1 和 person2 都有一个名为 sayName() 的方法，但那两个方法不是同一个 Function 的实例。不要忘了——ECMAScript 中的函数是对象，因此每定义一个函数，也就是实例化了一个对象。从逻辑角度讲，此时的构造函数也可以这样定义。

function Person(name, age, job){

this.name = name;

this.age = age;

this.job = job;

this.sayName = new Function("alert(this.name)"); // 与声明函数在逻辑上是等价的

}

从这个角度上来看构造函数，更容易明白每个 Person 实例都包含一个不同的 Function 实例（以显示 name 属性）的本质。说明白些，以这种方式创建函数，会导致不同的作用域链和标识符解析，但创建 Function 新实例的机制仍然是相同的。因此，不同实例上的同名函数是不相等的，以下代码可以证明这一点。

alert(person1.sayName == person2.sayName); //false

然而，创建两个完成同样任务的 Function 实例的确没有必要；况且有 this 对象在，根本不用在执行代码前就把函数绑定到特定对象上面。

### 原型模式

我们创建的每个函数都有一个 prototype （原型）属性，这个属性是一个指针，指向一个对象，而这个对象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。如果按照字面意思来理解，那么 prototype 就是通过调用构造函数而创建的那个对象实例的原型对象。使用原型对象的好处是可以让所有对象实例共享它所包含的属性和方法。换句话说，**不必在构造函数中定义对象实例的信息**，而是可以将这些信息直接添加到原型对象中，如下面的例子所示。

function Person(){

}

Person.prototype.name = "Nicholas";

Person.prototype.age = 29;

Person.prototype.job = "Software Engineer";

Person.prototype.sayName = function(){

alert(this.name);

};

var person1 = new Person();

person1.sayName(); //"Nicholas"

var person2 = new Person();

person2.sayName(); //"Nicholas"

alert(person1.sayName == person2.sayName); //true

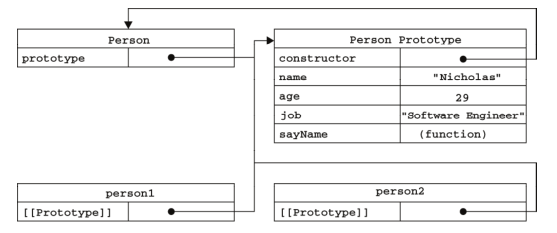
在此，我们将 sayName() 方法和所有属性直接添加到了 Person 的 prototype 属性中，构造函数变成了空函数。即使如此，也仍然可以通过调用构造函数来创建新对象，而且新对象还会具有相同的属性和方法。但与构造函数模式不同的是，新对象的这些属性和方法是由所有实例共享的。换句话说，person1 和 person2 访问的都是同一组属性和同一个 sayName() 函数。要理解原型模式的工作原理，必须先理解 ECMAScript 中原型对象的性质。

#### 1. 理解原型对象

**无论什么时候，只要创建了一个新函数，就会根据一组特定的规则为该函数创建一个 prototype属性，这个属性指向函数的原型对象**。在默认情况下，所有原型对象都会自动获得一个 constructor（构造函数）属性，这个属性包含一个指向 prototype 属性所在函数的指针。就拿前面的例子来说，Person.prototype. constructor 指向 Person 。而通过这个构造函数，我们还可继续为原型对象添加其他属性和方法。

创建了自定义的构造函数之后，其原型对象默认只会取得 constructor 属性；至于其他方法，则都是从 Object 继承而来的。当调用构造函数创建一个新实例后，该实例的内部将包含一个指针（内部属性），指向构造函数的原型对象。ECMA-262 第 5 版中管这个指针叫 [[Prototype]] 。虽然在脚本中没有标准的方式访问 [[Prototype]] ，但 Firefox、Safari 和 Chrome 在每个对象上都支持一个属性\_\_proto\_\_ ；而在其他实现中，这个属性对脚本则是完全不可见的。不过，要明确的真正重要的一点就是，这个连接存在于实例与构造函数的原型对象之间，而不是存在于实例与构造函数之间。

以前面使用 Person 构造函数和 Person.prototype 创建实例的代码为例，下图展示了各个对象之间的关系。



展示了 Person 构造函数、 Person 的原型属性以及 Person 现有的两个实例之间的关系。

在此， Person.prototype 指向了原型对象，而 Person.prototype.constructor 又指回了 Person 。原型对象中除了包含 constructor 属性之外，还包括后来添加的其他属性。 Person 的每个实例——person1 和 person2 都包含一个内部属性，该属性仅仅指向了 Person.prototype ；换句话说，它们与构造函数没有直接的关系。此外，要格外注意的是，虽然这两个实例都不包含属性和方法，但我们却可以调用 person1.sayName() 。这是通过查找对象属性的过程来实现的。

每当代码读取某个对象的某个属性时，都会执行一次搜索，目标是具有给定名字的属性。搜索首先从对象实例本身开始。如果在实例中找到了具有给定名字的属性，则返回该属性的值；如果没有找到，则继续搜索指针指向的原型对象，在原型对象中查找具有给定名字的属性。如果在原型对象中找到了这个属性，则返回该属性的值。也就是说，在我们调用 person1.sayName() 的时候，会先后执行两次搜索。首先，解析器会问：“实例 person1 有 sayName 属性吗？”答：“没有。”然后，它继续搜索，再问：“ person1 的原型有 sayName 属性吗？”答：“有。”于是，它就读取那个保存在原型对象中的函数。当我们调用 person2.sayName() 时，将会重现相同的搜索过程，得到相同的结果。而这正是多个对象实例共享原型所保存的属性和方法的基本原理。

前面提到过，原型最初只包含 constructor 属性，而该属性也是共享的，因此可以通过对象实例访问。虽然可以通过对象实例访问保存在原型中的值，但却不能通过对象实例重写原型中的值。如果我们在实例中添加了一个属性，而该属性与实例原型中的一个属性同名，那我们就在实例中创建该属性，该属性将会屏蔽原型中的那个属性。来看下面的例子。

function Person(){

}

Person.prototype.name = "Nicholas";

Person.prototype.age = 29;

Person.prototype.job = "Software Engineer";

Person.prototype.sayName = function(){

alert(this.name);

};

var person1 = new Person();

var person2 = new Person();

person1.name = "Greg";

alert(person1.name); //"Greg" —— 来自实例

alert(person2.name); //"Nicholas" —— 来自原型

在这个例子中， person1 的 name 被一个新值给屏蔽了。但无论访问 person1.name 还是访问person2.name 都能够正常地返回值，即分别是 "Greg" （来自对象实例）和 "Nicholas" （来自原型）。当在 alert() 中访问 person1.name 时，需要读取它的值，因此就会在这个实例上搜索一个名为 name的属性。这个属性确实存在，于是就返回它的值而不必再搜索原型了。当以同样的方式访问 person2.name 时，并没有在实例上发现该属性，因此就会继续搜索原型，结果在那里找到了 name 属性。

**当为对象实例添加一个属性时，这个属性就会屏蔽原型对象中保存的同名属性；换句话说，添加这个属性只会阻止我们访问原型中的那个属性，但不会修改那个属性**。即使将这个属性设置为 null ，也只会在实例中设置这个属性，而不会恢复其指向原型的连接。不过，使用 delete 操作符则可以完全删除实例属性，从而让我们能够重新访问原型中的属性，如下所示。

function Person(){

}

Person.prototype.name = "Nicholas";

Person.prototype.age = 29;

Person.prototype.job = "Software Engineer";

Person.prototype.sayName = function(){

alert(this.name);

};

var person1 = new Person();

var person2 = new Person();

person1.name = "Greg";

alert(person1.name); //"Greg"——来自实例

alert(person2.name); //"Nicholas"——来自原型

delete person1.name;

alert(person1.name); //"Nicholas" —— 来自原型

使用 **hasOwnProperty()**方法可以检测一个属性是存在于实例中，还是存在于原型中。这个方法（不要忘了它是从 Object 继承来的）只在给定属性存在于对象实例中时，才会返回 true。

#### 2. 原型与 in 操作符

有两种方式使用 in 操作符：单独使用和在 for-in 循环中使用。在单独使用时， 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。看一看下面的例子。

function Person(){

}

Person.prototype.name = "Nicholas";

Person.prototype.age = 29;

Person.prototype.job = "Software Engineer";

Person.prototype.sayName = function(){

alert(this.name);

};

var person1 = new Person();

var person2 = new Person();

alert(person1.hasOwnProperty("name")); //false

alert("name" in person1); //true

person1.name = "Greg";

alert(person1.name); //"Greg" ——来自实例

alert(person1.hasOwnProperty("name")); //true

alert("name" in person1); //true

alert(person2.name); //"Nicholas" ——来自原型

alert(person2.hasOwnProperty("name")); //false

alert("name" in person2); //true

delete person1.name;

alert(person1.name); //"Nicholas" ——来自原型

alert(person1.hasOwnProperty("name")); //false

alert("name" in person1); //true

在以上代码执行的整个过程中， name 属性要么是直接在对象上访问到的，要么是通过原型访问到的。因此，调用 "name" in person1 始终都返回 true ，无论该属性存在于实例中还是存在于原型中。

同时使用 hasOwnProperty() 方法和 in 操作符，就可以确定该属性到底是存在于对象中，还是存在于原型中。

function hasPrototypeProperty(object, name){

return !object.hasOwnProperty(name) && (name in object);

}

由于 in 操作符只要通过对象能够访问到属性就返回 true ， hasOwnProperty() 只在属性存在于实例中时才返回 true ，因此只要 in 操作符返回 true 而 hasOwnProperty() 返回 false ，就可以确定属性是原型中的属性。

#### 3. 原生对象的原型

原型模式的重要性不仅体现在创建自定义类型方面，就连所有原生的引用类型，都是采用这种模式创建的。所有原生引用类型（ Object 、 Array 、 String ，等等）都在其构造函数的原型上定义了方法。

例如，在 Array.prototype 中可以找到 sort() 方法，而在 String.prototype 中可以找到substring() 方法，如下所示。

alert(typeof Array.prototype.sort); //"function"

alert(typeof String.prototype.substring); //"function"

通过原生对象的原型，不仅可以取得所有默认方法的引用，而且也可以定义新方法。可以像修改自定义对象的原型一样修改原生对象的原型，因此可以随时添加方法。下面的代码就给基本包装类型String 添加了一个名为 startsWith() 的方法。

String.prototype.startsWith = function (text) {

return this.indexOf(text) == 0;

};

var msg = "Hello world!";

alert(msg.startsWith("Hello")); //true

这里新定义的 startsWith() 方法会在传入的文本位于一个字符串开始时返回 true 。既然方法被添加给了 String.prototype ，那么当前环境中的所有字符串就都可以调用它。由于 msg 是字符串，而且后台会调用 String 基本包装函数创建这个字符串，因此通过 msg 就可以调用 startsWith() 方法。

尽管可以这样做，但我们不推荐在产品化的程序中修改原生对象的原型。如果因某个实现中缺少某个方法，就在原生对象的原型中添加这个方法，那么当在另一个支持该方法的实现中运行代码时，就可能会导致命名冲突。而且，这样做也可能会意外地重写原生方法。

#### 4. 原型对象的问题

原型模式也不是没有缺点。首先，它省略了为构造函数传递初始化参数这一环节，结果所有实例在默认情况下都将取得相同的属性值。虽然这会在某种程度上带来一些不方便，但还不是原型的最大问题。原型模式的最大问题是由其共享的本性所导致的。

function Person(){

}

Person.prototype = {

constructor: Person,

name : "Nicholas",

age : 29,

job : "Software Engineer",

friends : ["Shelby", "Court"],

sayName : function () {

alert(this.name);

}

};

var person1 = new Person();

var person2 = new Person();

person1.friends.push("Van");

alert(person1.friends); //"Shelby,Court,Van"

alert(person2.friends); //"Shelby,Court,Van"

alert(person1.friends === person2.friends); //true

在此， Person.prototype 对象有一个名为 friends 的属性，该属性包含一个字符串数组。然后，创建了 Person 的两个实例。接着，修改了 person1.friends 引用的数组，向数组中添加了一个字符串。由于 friends 数组存在于 Person.prototype 而非 person1 中，所以刚刚提到的修改也会通过person2.friends （与 person1.friends 指向同一个数组）反映出来。假如我们的初衷就是像这样在所有实例中共享一个数组，那么对这个结果我没有话可说。可是，实例一般都是要有属于自己的全部属性的。而这个问题正是我们很少看到有人单独使用原型模式的原因所在。

### 组合使用构造函数模式和原型模式

创建自定义类型的最常见方式，就是组合使用构造函数模式与原型模式。**构造函数模式用于定义实例属性，而原型模式用于定义方法和共享的属性**。结果，每个实例都会有自己的一份实例属性的副本，但同时又共享着对方法的引用，最大限度地节省了内存。另外，这种混成模式还支持向构造函数传递参数；可谓是集两种模式之长。

function Person(name, age, job){

this.name = name;

this.age = age;

this.job = job;

}

Person.prototype = {

constructor : Person,

sayName : function(){

alert(this.name);

}

}

var person1 = new Person("Nicholas", 29, "Software Engineer");

var person2 = new Person("Greg", 27, "Doctor");

person1.friends.push("Van");

alert(person1.friends); //"Shelby,Count,Van"

alert(person2.friends); //"Shelby,Count"

alert(person1.friends === person2.friends); //false

alert(person1.sayName === person2.sayName); //true

在这个例子中，实例属性都是在构造函数中定义的，而由所有实例共享的属性 constructor 和方法 sayName() 则是在原型中定义的。而修改了 person1.friends （向其中添加一个新字符串），并不会影响到 person2.friends ，因为它们分别引用了不同的数组。

**这种构造函数与原型混成的模式，是目前在 ECMAScript中使用最广泛、认同度最高的一种创建自定义类型的方法。可以说，这是用来定义引用类型的一种默认模式。**

## 继承

继承是 OO 语言中的一个最为人津津乐道的概念。许多 OO 语言都支持两种继承方式：接口继承和实现继承。接口继承只继承方法签名，而实现继承则继承实际的方法。如前所述，由于函数没有签名，在 ECMAScript 中无法实现接口继承。ECMAScript 只支持实现继承，而且其实现继承主要是依靠原型链来实现的。

### 原型链

ECMAScript 中描述了原型链的概念，并将原型链作为实现继承的主要方法。其基本思想是利用原型让一个引用类型继承另一个引用类型的属性和方法。简单回顾一下构造函数、原型和实例的关系：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，而实例都包含一个指向原型对象的内部指针。那么，假如我们让原型对象等于另一个类型的实例，结果会怎么样呢？显然，此时的原型对象将包含一个指向另一个原型的指针，相应地，另一个原型中也包含着一个指向另一个构造函数的指针。假如另一个原型又是另一个类型的实例，那么上述关系依然成立，如此层层递进，就构成了实例与原型的链条。这就是所谓原型链的基本概念。

实现原型链有一种基本模式，其代码大致如下。

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 (){

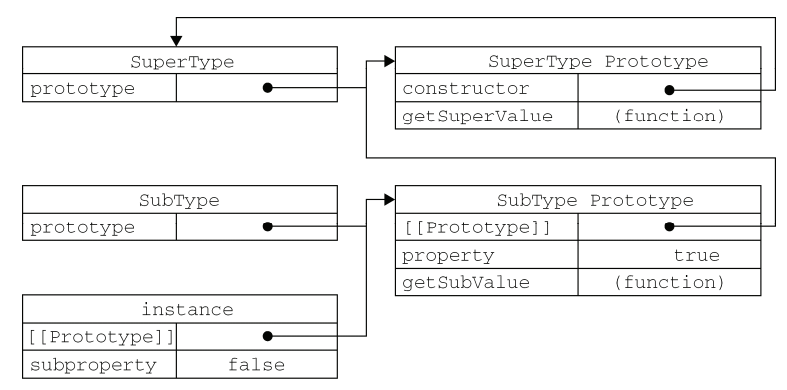
return this.subproperty;

};

var instance = new SubType();

alert(instance.getSuperValue()); //true

以上代码定义了两个类型： SuperType 和 SubType 。每个类型分别有一个属性和一个方法。它们的主要区别是 SubType 继承了 SuperType ，而继承是通过创建 SuperType 的实例，并将该实例赋给SubType.prototype 实现的。实现的本质是重写原型对象，代之以一个新类型的实例。换句话说，原来存在于 SuperType 的实例中的所有属性和方法，现在也存在于 SubType.prototype 中了。在确立了继承关系之后，我们给 SubType.prototype 添加了一个方法，这样就在继承了 SuperType 的属性和方法的基础上又添加了一个新方法。这个例子中的实例以及构造函数和原型之间的关系如图所示。

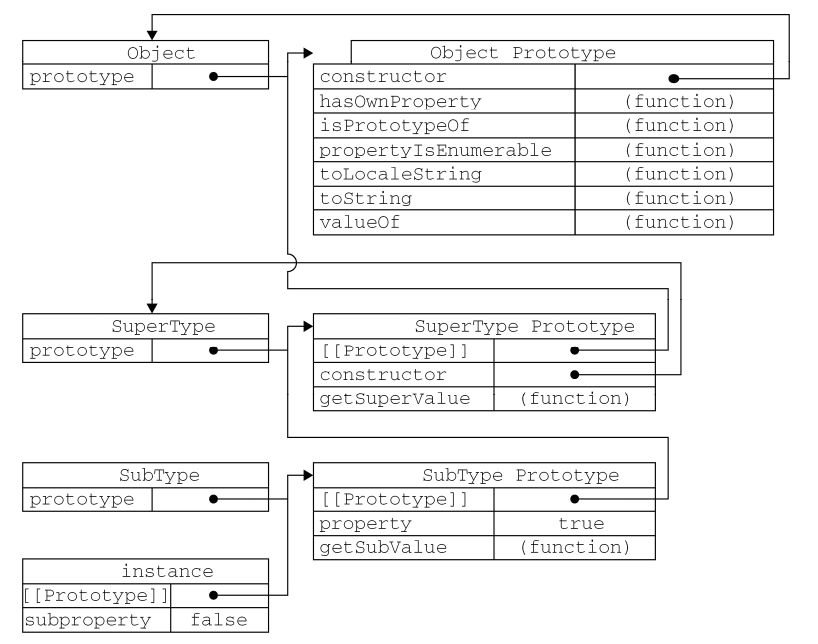


在上面的代码中，我们没有使用 SubType 默认提供的原型，而是给它换了一个新原型；这个新原型就是 SuperType 的实例。于是，新原型不仅具有作为一个 SuperType 的实例所拥有的全部属性和方法，而且其内部还有一个指针，指向了 SuperType 的原型。最终结果就是这样的： instance 指向 SubType的 原 型 ， SubType 的 原 型 又 指 向 SuperType 的 原 型 。 getSuperValue() 方 法 仍 然 还 在SuperType.prototype 中，但 property 则位于 SubType.prototype 中。这是因为 property 是一个实例属性，而 getSuperValue() 则是一个原型方法。既然 SubType.prototype 现在是 SuperType的实例，那么 property 当然就位于该实例中了。此外，要注意 instance.constructor 现在指向的是 SuperType ，这是因为原来 SubType.prototype 中的 constructor 被重写了的缘故 。

当以读取模式访问一个实例属性时，首先会在实例中搜索该属性。如果没有找到该属性，则会继续搜索实例的原型。在通过原型链实现继承的情况下，搜索过程就得以沿着原型链继续向上。就拿上面的例子来说，调用instance.getSuperValue() 会经历三个搜索步骤：1）搜索实例；2）搜索 SubType.prototype ；3）搜索 SuperType.prototype ，最后一步才会找到该方法。在找不到属性或方法的情况下，搜索过程总是要一环一环地前行到原型链末端才会停下来。

**1. 别忘记默认的原型**

事实上，前面例子中展示的原型链还少一环。我们知道，所有引用类型默认都继承了 Object ，而这个继承也是通过原型链实现的。大家要记住，所有函数的默认原型都是 Object 的实例，因此默认原型都会包含一个内部指针，指向 Object.prototype 。这也正是所有自定义类型都会继承 toString() 、valueOf() 等默认方法的根本原因。所以，我们说上面例子展示的原型链中还应该包括另外一个继承层次。下图为我们展示了该例子中完整的原型链。



一句话， SubType 继承了 SuperType ，而 SuperType 继承了 Object 。当调用 instance.toString()时，实际上调用的是保存在 Object.prototype 中的那个方法。

**2. 确定原型和实例的关系**

可以使用 instanceof 操作符来确定原型和实例之间的关系，只要用这个操作符来测试实例与原型链中出现过的构造函数，结果就会返回 true 。以下几行代码就说明了这一点。

alert(instance instanceof Object); //true

alert(instance instanceof SuperType); //true

alert(instance instanceof SubType); //true

由于原型链的关系，我们可以说 instance 是 Object 、 SuperType 或 SubType 中任何一个类型的实例。因此，测试这三个构造函数的结果都返回了 true 。

**3. 谨慎地定义方法**

子类型有时候需要重写超类型中的某个方法，或者需要添加超类型中不存在的某个方法。但不管怎样，给原型添加方法的代码一定要放在替换原型的语句之后。来看下面的例子。

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();

// 添加新方法

SubType.prototype.getSubValue = function (){

return this.subproperty;

};

// 重写超类型中的方法

SubType.prototype.getSuperValue = function (){

return false;

};

var instance = new SubType();

alert(instance.getSuperValue()); //false

在以上代码中，加粗的部分是两个方法的定义。第一个方法 getSubValue() 被添加到了 SubType中。第二个方法 getSuperValue() 是原型链中已经存在的一个方法，但重写这个方法将会屏蔽原来的那个方法。换句话说，当通过 SubType 的实例调用 getSuperValue() 时，调用的就是这个重新定义的方法；但通过 SuperType 的实例调用 getSuperValue() 时，还会继续调用原来的那个方法。这里要格外注意的是，必须在用 SuperType 的实例替换原型之后，再定义这两个方法。

还有一点需要提醒，即在通过原型链实现继承时，不能使用对象字面量创建原型方法。因为这样做就会重写原型链，如下面的例子所示。

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();

// 使用字面量添加新方法，会导致上一行代码无效

SubType.prototype = {

getSubValue : function (){

return this.subproperty;

},

someOtherMethod : function (){

return false;

}

};

var instance = new SubType();

alert(instance.getSuperValue()); //error!

以上代码展示了刚刚把 SuperType 的实例赋值给原型，紧接着又将原型替换成一个对象字面量而导致的问题。由于现在的原型包含的是一个 Object 的实例，而非 SuperType 的实例，因此我们设想中的原型链已经被切断—— SubType 和 SuperType 之间已经没有关系了。

**4. 原型链的问题**

原型链虽然很强大，可以用它来实现继承，但它也存在一些问题。其中，最主要的问题来自包含引用类型值的原型。包含引用类型值的原型属性会被所有实例共享；而这也正是为什么要在构造函数中，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理成章地变成了现在的原型属性了。

下列代码可以用来说明这个问题。

function SuperType(){

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

}

function SubType(){

}

//继承了 SuperType

SubType.prototype = new SuperType();

var instance1 = new SubType();

instance1.colors.push("black");

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

var instance2 = new SubType();

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

这个例子中的 SuperType 构造函数定义了一个 colors 属性，该属性包含一个数组（引用类型值）。SuperType 的每个实例都会有各自包含自己数组的 colors 属性。当 SubType 通过原型链继承了SuperType 之后， SubType.prototype 就变成了 SuperType 的一个实例，因此它也拥有了一个它自己的 colors 属性——就跟专门创建了一个 SubType.prototype.colors 属性一样。但结果是什么呢？结果是 SubType 的所有实例都会共享这一个 colors 属性。而我们对 instance1.colors 的修改能够通过 instance2.colors 反映出来，就已经充分证实了这一点。

原型链的第二个问题是：在创建子类型的实例时，不能向超类型的构造函数中传递参数。实际上，应该说是没有办法在不影响所有对象实例的情况下，给超类型的构造函数传递参数。有鉴于此，再加上前面刚刚讨论过的由于原型中包含引用类型值所带来的问题，实践中很少会单独使用原型链。

### 借用构造函数

在解决原型中包含引用类型值所带来问题的过程中，开发人员开始使用一种叫做借用构造函数（constructor stealing）的技术（有时候也叫做伪造对象或经典继承）。这种技术的**基本思想即在子类型构造函数的内部调用超类型构造函数**。别忘了，函数只不过是在特定环境中执行代码的对象，因此通过使用 apply() 和 call() 方法也可以在（将来）新创建的对象上执行构造函数，如下所示：

function SuperType(){

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

}

function SubType(){

// 继承了 SuperType

SuperType.call(this);

}

var instance1 = new SubType();

instance1.colors.push("black");

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

var instance2 = new SubType();

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

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。

function SuperType(name){

this.name = name;

}

function SubType(){

//继承了 SuperType，同时还传递了参数

SuperType.call(this, "Nicholas");

//实例属性

this.age = 29;

}

var instance = new SubType();

alert(instance.name); //"Nicholas";

alert(instance.age); //29

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 的实例设置了 name 属性。为了确保SuperType 构造函数不会重写子类型的属性，可以在调用超类型构造函数后，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，对子类型而言也是不可见的，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。

### 组合继承

组合继承（combination inheritance），有时候也叫做伪经典继承，**指的是将原型链和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对实例属性的继承。这样，既通过在原型上定义方法实现了函数复用，又能够保证每个实例都有它自己的属性。下面来看一个例子。

function SuperType(name){

this.name = name;

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

}

SuperType.prototype.sayName = function(){

alert(this.name);

};

function SubType(name, age){

//继承属性

SuperType.call(this, name);

this.age = age;

}

//继承方法

SubType.prototype = new SuperType();

SubType.prototype.constructor = SubType;

SubType.prototype.sayAge = function(){

alert(this.age);

};

var instance1 = new SubType("Nicholas", 29);

instance1.colors.push("black");

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

instance1.sayName(); //"Nicholas";

instance1.sayAge(); //29

var instance2 = new SubType("Greg", 27);

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

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

instance2.sayAge(); //27

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 实例既分别拥有自己属性——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 中最常用的继承模式。

**回顾apply和call方法：**

每个函数都包含两个非继承而来的方法： apply() 和 call() 。这两个方法的用途都是在特定的作用域中调用函数，实际上等于设置函数体内 this 对象的值。首先， apply() 方法接收两个参数：一个是在其中运行函数的作用域，另一个是参数数组。其中，第二个参数可以是 Array 的实例，也可以是arguments 对象。例如：

function sum(num1, num2){

return num1 + num2;

}

function callSum1(num1, num2){

return sum.apply(this, arguments); // 传入 arguments 对象

}

function callSum2(num1, num2){

return sum.apply(this, [num1, num2]); // 传入数组

}

alert(callSum1(10,10)); //20

alert(callSum2(10,10)); //20

在上面这个例子中， callSum1() 在执行 sum() 函数时传入了 this 作为 this 值（因为是在全局作用域中调用的，所以传入的就是 window 对象）和 arguments 对象。而 callSum2 同样也调用了sum() 函数，但它传入的则是 this 和一个参数数组。这两个函数都会正常执行并返回正确的结果。

call() 方法与 apply() 方法的作用相同，它们的区别仅在于接收参数的方式不同。对于 call()方法而言，第一个参数是 this 值没有变化，变化的是其余参数都直接传递给函数。换句话说，在使用call() 方法时，传递给函数的参数必须逐个列举出来，如下面的例子所示。

function sum(num1, num2){

return num1 + num2;

}

function callSum(num1, num2){

return sum.call(this, num1, num2);

}

alert(callSum(10,10)); //20

在使用 call() 方法的情况下， callSum() 必须明确地传入每一个参数。结果与使用 apply() 没有什么不同。至于是使用 apply() 还是 call() ，完全取决于你采取哪种给函数传递参数的方式最方便。

如果你打算直接传入 arguments 对象，或者包含函数中先接收到的也是一个数组，那么使用 apply()肯定更方便；否则，选择 call() 可能更合适。（在不给函数传递参数的情况下，使用哪个方法都无所谓。）

事实上，传递参数并非 apply() 和 call() 真正的用武之地；它们真正强大的地方是能够扩充函数赖以运行的作用域。下面来看一个例子。

window.color = "red";

var o = { color: "blue" };

function sayColor(){

alert(this.color);

}

sayColor(); //red

sayColor.call(this); //red

sayColor.call(window); //red

sayColor.call(o); //blue

这个例子是在前面说明 this 对象的示例基础上修改而成的。这一次， sayColor() 也是作为全局函数定义的，而且当在全局作用域中调用它时，它确实会显示 "red" ——因为对 this.color 的求值会转换成对 window.color 的求值。而 sayColor.call(this) 和 sayColor.call(window) ，则是两种显式地在全局作用域中调用函数的方式，结果当然都会显示 "red" 。但是，当运行 sayColor.call(o)时，函数的执行环境就不一样了，因为此时函数体内的 this 对象指向了 o ，于是结果显示的是 "blue" 。

使用 call() （或 apply() ）来扩充作用域的最大好处，就是对象不需要与方法有任何耦合关系。