**第六章面向对象（156）**

**理解对象**

理解对象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。

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。

1. 属性类型

ECMAScript 中有两种属性：数据属性和访问器属性。

**数据属性**

在这个位置可以读取和写入值。数据属性有 4 个描述其行为的特性。

q [[Configurable]]：表示能否**通过 delete 删除属性从而重新定义属性**，**能否修改属性的特性**，或者能否把属性修改为访问器属性。像前面例子中那样直接在对象上定义的属性，它们的这个特性默认值为 true。

q [[Enumerable]]：**表示能否通过 for-in 循环返回属性**。像前面例子中那样直接在对象上定义的属性，它们的这个特性默认值为 true。

q [[Writable]]：**表示能否修改属性的值**。像前面例子中那样直接在对象上定义的属性，它们的这个特性默认值为 true。

q [[Value]]：包含这个属性的数据值。读取属性值的时候，从这个位置读；写入属性值的时候，把新值保存在这个位置。这个特性的默认值为 undefined。

要修改属性默认的特性，必须使用 ECMAScript 5 的 Object.defineProperty()方法来定义属性。

var person = {};

Object.defineProperty(person, "name", {

writable: false,

value: "Nicholas"

});

这个例子创建了一个名为 name 的属性，它的值"Nicholas"是只读的。这个属性的值是不可修改的，如果尝试为它指定新值，则在非严格模式下，赋值操作将被忽略；在严格模式下，赋值操作将会导致抛出错误。

把 configurable 设置为 false，表示不能从对象中删除属性。如果对这个属性调用 delete，则在非严格模式下什么也不会发生，而在严格模式下会导致错误。而且，一旦把属性定义为不可配置的，就不能再把它变回可配置了。此时，再调用 Object.defineProperty()方法修改除 writable 之外的特性，都会导致错误。

**访问器属性**

访问器属性有如下 4 个特性。

q [[Configurable]]：表示能否通过 delete 删除属性从而重新定义属性，能否修改属性的特性，或者能否把属性修改为数据属性。对于直接在对象上定义的属性，这个特性的默认值为true。

q [[Enumerable]]：表示能否通过 for-in 循环返回属性。对于直接在对象上定义的属性，这个特性的默认值为 true。

q [[Get]]：在读取属性时调用的函数。默认值为 undefined。

q [[Set]]：在写入属性时调用的函数。默认值为 undefined。

访问器属性不能直接定义，必须使用 Object.defineProperty()来定义。

var book = {

\_year: 2004,

edition: 1

};

Object.defineProperty(book, "year", {

get: function(){

return this.\_year;

},

set: function(newValue){

if (newValue > 2004) {

this.\_year = newValue;

this.edition += newValue - 2004;

}

}

});

book.year = 2005;

alert(book.edition); //2

\_year 前面的下划线是一种常用的记号，用于表示只能通过对象方法访问的属性。这是使用访问器属性的常见方式，即设置一个属性的值会导致其他属性发生变化。

不一定非要同时指定 getter 和 setter。只指定 getter 意味着属性是不能写，只指定 setter 函数的属性也不能读，会返回 undefined。严格模式下回报错。

在不支持 Object.defineProperty() 方法的浏览器中使用下面遗留函数实现，不能修改 [[Configurable]] 和[[Enumerable]]。

//定义访问器的旧有方法

book.**\_\_defineGetter\_\_**("year", function(){

return this.\_year;

});

book.**\_\_defineSetter\_\_**("year", function(newValue){

if (newValue > 2004) {

this.\_year = newValue;

this.edition += newValue - 2004;

}

});

1. 定义多个属性

ECMAScript 5 又定义了一个 Object.defineProperties()方法。利用这个方法可以通过描述符一次定义多个属性。这个方法接收两个对象参数：第一个对象是要添加和修改其属性的对象，第二个对象的属性与第一个对象中要添加或修改的属性一一对应。例如：

var book = {};

Object.defineProperties(book, {

\_year: {

value: 2004

},

edition: {

value: 1

},

year: {

get: function(){

return this.\_year;

},

set: function(newValue){

if (newValue > 2004) {

this.\_year = newValue;

this.edition += newValue - 2004;

}

}

}

});

最终的对象与上一节中定义的对象相同。唯一的区别是这里的属性都是在同一时间创建的。

1. 读取属性的特性

使用 ECMAScript 5 的 Object.getOwnPropertyDescriptor()方法，可以取得给定属性的描述符。这个方法接收两个参数：对象和要读取其描述符的属性名称。返回值是一个对象。

var book = {};

Object.defineProperties(book, {

\_year: {

value: 2004

},

edition: {

value: 1

},

year: {

get: function(){

return this.\_year;

},

set: function(newValue){

if (newValue > 2004) {

this.\_year = newValue;

this.edition += newValue - 2004;

}

}

}

});

var descriptor = Object.getOwnPropertyDescriptor(book, "\_year");

**创建对象**

1. 工厂模式

考虑到在 ECMAScript 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。

考虑到在 ECMAScript 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;**

}

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）。

1. 构造函数模式

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。

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");

与工厂模式比较，还存在以下不同之处：

q 没有显式地创建对象；

q 直接将属性和方法赋给了 this 对象；

q 没有 return 语句。

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。

这两个对象都有一个 constructor（构造函数）属性，该属性指向 Person，如下所示。

alert(person1.constructor == Person); //true

alert(person2.constructor == Person); //true

提到检测对象类型，还是 instanceof操作符要更可靠一些。有对象既是 Object 的实例，同时也是 Person的实例。

alert(person1 instanceof Object); //true

alert(person1 instanceof Person); //true

**将构造函数当成函数**

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。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。

// 在另一个对象的作用域中调用

var o = new Object();

Person.call(o, "Kristen", 25, "Nurse");

o.sayName(); //"Kristen"

**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。

1. 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。

function Person(){

}

Person.prototype.name = "Nicholas";

Person.prototype.age = 29;

Person.prototype.job = "Software Engineer";

Person.prototype.sayName = function(){

alert(this.name);

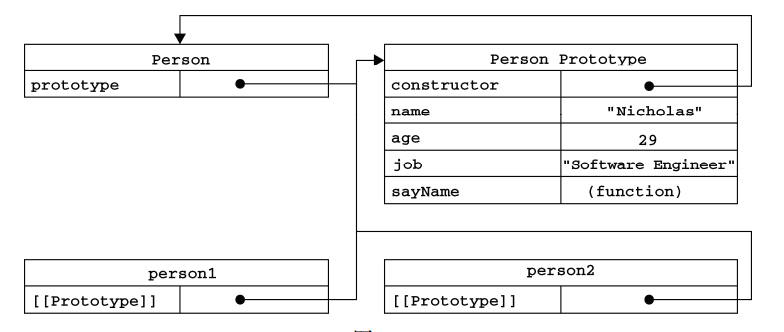
};

**理解原型对象**

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。

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。Firefox、Safari 和 Chrome 在每个对象上都支持一个属性

\_\_proto\_\_来访问这个指针。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。



虽然在所有实现中都无法访问到[[Prototype]]，但可以通过 isPrototypeOf()方法来确定对象之间是否存在这种关系。从本质上讲，如果[[Prototype]]指向调用 isPrototypeOf()方法的对象（Person.prototype），那么这个方法就返回 true

alert(Person.prototype.isPrototypeOf(person1)); //true，表示原型对象是person1的

ECMAScript 5 增加了一个新方法，叫 Object.getPrototypeOf()，在所有支持的实现中，这个方法返回[[Prototype]]的值。使用 Object.getPrototypeOf()可以方便地取得一个对象的原型，而这在利用原型实现继承（本章稍后会讨论）的情况下是非常重要的。例如：

alert(Object.getPrototypeOf(person1) == Person.prototype); //true

每当代码读取某个对象的某个属性时，都会执行搜索。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。

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。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。

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。

person1.name = "Greg";

alert(person1.name); //"Greg"——来自实例

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

delete person1.name;

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

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

**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。

person1.name = "Greg";

alert(person1.name); //"Greg" ——来自实例

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

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

delete person1.name;

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

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

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

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){//只有当属性是原型对象（prototype）的属性时返回true

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

}

在使用 for-in 循环时，返回的是所有能够通过对象访问的、可枚举的（enumerated）属性，其中既包括存在于实例中的属性，也包括存在于原型中的属性。**屏蔽了原型中不可枚举属性（即原型中[[Enumerable]]标记为 false 的属性被实例中同名属性替代的情况）的实例属性也会在 for-in 循环中返回**，因为根据规定，所有开发人员定义的属性都是可枚举的——**只有在 IE8 及更早版本中例外**。

Object对象中不可枚举的包括：hasOwnProperty()、propertyIsEnumerable()、toLocaleString()、toString()和 valueOf()。ECMAScript 5 也将 constructor 和 prototype 属性的[[Enumerable]]特性设置为 false，但并不是所有浏览器都照此实现。

要取得对象上所有可枚举的实例属性，可以使用 ECMAScript 5 的 Object.keys()方法。

如果你想要得到所有实例属性，无论它是否可枚举，都可以使用 Object.getOwnPropertyNames()方法，结果中包含了不可枚举的 constructor 属性。

**更简单的原型语法**

读者大概注意到了，前面例子中每添加一个属性和方法就要敲一遍 Person.prototype。为减少不必要的输入，也为了从视觉上更好地封装原型的功能，如下写原型对象：

function Person(){

}

Person.prototype = {

**constructor : Person,//** **每创建一个函数，就会同时创建它的 prototype 对象，这个对象也会自动获得 constructor 属性。而我们在这里使用的语法，本质上完全重写了默认的 prototype 对象，因此 constructor 属性也就变成了新对象的 constructor 属性（指向 Object 构造函数），不再指向 Person 函数。**

name : "Nicholas",

age : 29,

job: "Software Engineer",

sayName : function () {

alert(this.name);

}

};

如果不改变constructor将会出现下面结果，instanceof返回还是true，而constructor属性值为Object：

var friend = new Person();

alert(friend instanceof Object); //true

alert(friend instanceof Person); //true

alert(friend.constructor == Person); //false

alert(friend.constructor == Object); //true

以这种方式重设 constructor 属性会导致它的[[Enumerable]]特性被设置为 true。默认情况下，原生的 constructor 属性是不可枚举的，因此如果你使用兼容 ECMAScript 5 的 JavaScript 引擎，可以试一试 Object.defineProperty()。

//重设构造函数，只适用于 ECMAScript 5 兼容的浏览器

Object.defineProperty(Person.prototype, "constructor", {

enumerable: false,

value: Person

});

**原型的动态性**

由于在原型中查找值的过程是一次搜索，因此我们对原型对象所做的任何修改都能够立即从实例上反映出来——即使是**先创建了实例后修改原型**也照样如此。

var friend = new Person();

Person.prototype.sayHi = function(){

alert("hi");

};

friend.sayHi(); //"hi"（没有问题！）

但如果是重写整个原型对象，那么情况就不一样了。我们知道，调用构造函数时会为实例添加一个指向最初原型的[[Prototype]]指针，而把原型修改为另外一个对象就等于切断了构造函数与最初原型之间的联系。

function Person(){

}

var friend = new Person();

Person.prototype = {

sayName : function () {

alert(this.name);

}

};

friend.sayName(); //error

**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。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。

String.prototype.startsWith = function (text) {

return this.indexOf(text) == 0;

};

var msg = "Hello world!";

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

**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。

1. 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。

function Person(name, age, job){

this.name = name;

this.age = age;

this.job = job;

this.friends = ["Shelby", "Court"];

}

Person.prototype = {

constructor : Person,

sayName : function(){

alert(this.name);

}

}

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。

1. 动态原型模式

这里只在 sayName()方法不存在的情况下，才会将它添加到原型中。这段代码只会在初次调用构造函数时才会执行。此后，原型已经完成初始化，不需要再做什么修改了。不过要记住，这里对原型所做的修改，能够立即在所有实例中得到反映。

function Person(name, age, job){

//属性

this.name = name;

this.age = age;

this.job = job;

//方法

**if (typeof this.sayName != "function"){**

**Person.prototype.sayName = function(){**

**alert(this.name);**

**};**

**}**

}