**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()方法来修改属性。

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);

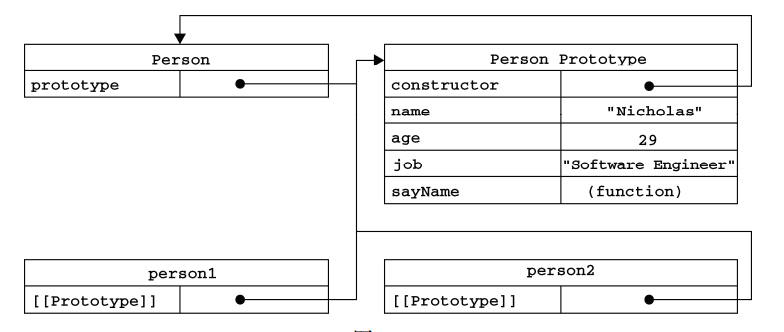
};

**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，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);**

**};**

**}**

}

1. 寄生构造函数模式

通常，在前述的几种模式都不适用的情况下，可以使用寄生（parasitic）构造函数模式。

function Person(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 friend = new Person("Nicholas", 29, "Software Engineer");

friend.sayName(); //"Nicholas"

除了使用 new 操作符并把使用的包装函数叫做构造函数之外，这个模式跟工厂模式其实是一模一样的。

1. 稳妥构造函数模式

所谓稳妥对象，指的是没有公共属性，而且其方法也不引用 this 的对象。稳妥对象最适合在一些安全的环境中（这些环境中会禁止使用 this 和 new）。稳妥构造函数遵循与寄生构造函数类似的模式，但有两点不同：一是新创建对象的实例方法不引用 this；二是不使用 new 操作符调用构造函数。按照稳妥构造函数的要求，可以将前面的 Person 构造函数重写如下。

function Person(name, age, job){

//创建要返回的对象

var o = new Object();

//可以在这里定义私有变量和函数

Var name = name;

//添加方法

o.sayName = function(){

alert(name);

};

//返回对象

return o;

}

在以这种模式创建的对象中，除了使用 sayName()方法之外，没有其他办法访问 name 的值。

**继承**

1. 原型链

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 (){

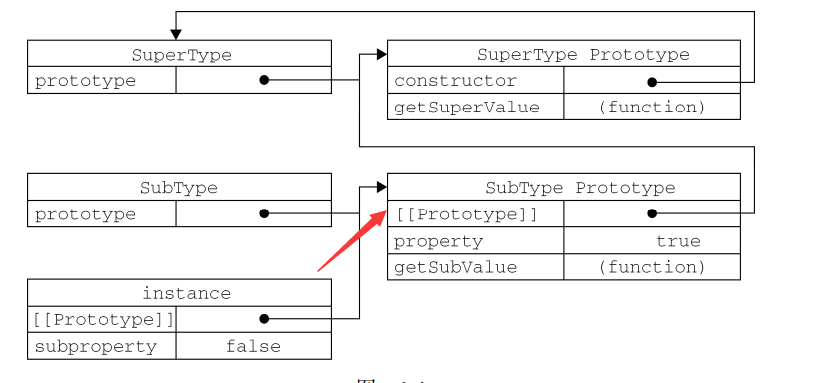
return this.subproperty;

};

var instance = new SubType();

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

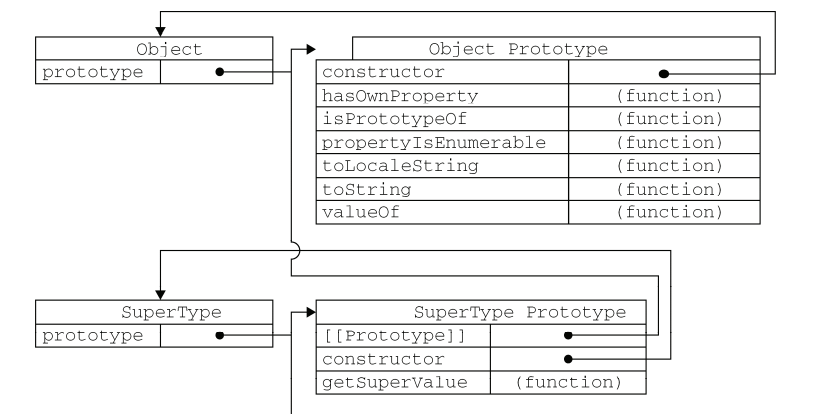
要注意 instance.constructor 现在指向的是 SuperType，这是因为原来 SubType.prototype 中的 constructor 被重写了的缘故。



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。

**别忘了默认的原型**

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。



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。

**确定原型和实例的关系**

可以通过两种方式来确定原型和实例之间的关系。第一种方式是使用 instanceof 操作符，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。

alert(instance instanceof Object); //true

alert(instance instanceof SuperType); //true

alert(instance instanceof SubType); //true

第二种方式是使用 isPrototypeOf()方法。同样，只要是原型链中出现过的原型，都可以说是该原型链所派生的实例的原型。

alert(Object.prototype.isPrototypeOf(instance)); //true

alert(SuperType.prototype.isPrototypeOf(instance)); //true

alert(SubType.prototype.isPrototypeOf(instance)); //true

**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。

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，在继承之后，否则会像替换对象一样失去关联**

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

**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。

**function SuperType(){//构造函数的原型（另一个构造函数的实例）的原型**

**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"

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。

1. 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）。

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"

**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，为了确保不会覆盖下面定义的实力属性，应在定义实例属性之前调用构造函数**

**SuperType.call(this, "Nicholas");**

//实例属性

this.age = 29;

}

var instance = new SubType();

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

alert(instance.age); //29

**借助构造函数问题**

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。

1. 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。

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

1. 原型式继承

**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。**

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。

function object(o){

function F(){}

F.prototype = o;

return new F();

}

新对象将 o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o的属性会被共享。

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"

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。

var person = {

name: "Nicholas",

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

};

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

**name: {**

**value: "Greg"**

**}**

**});**

alert(anotherPerson.name); //"Greg"

1. 寄生式继承

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。

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

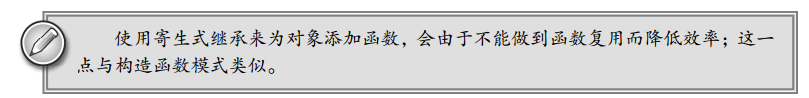
alert("hi");

};

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。



1. 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。

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);

};

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。**

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。

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

}

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 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。

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);

this.age = age;

}

**inheritPrototype(SubType, SuperType);**

SubType.prototype.sayAge = function(){

alert(this.age);

};

这个例子的高效率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。与此同时，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。

一些查看对象实例的原型和构造函数的方法[demo](demo.html)