

第6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

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。

每个对象都是基于一个引用类型创建的，这个引用类型可以是第 5 章讨论的原生类型，也可以是开发人员定义的类型。

6.1 理解对象

上一章曾经介绍过，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);
};
```

CreatingObjectsExample01.htm

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。

6.1.1 属性类型

ECMA-262 第 5 版在定义只有内部才用的特性（attribute）时，描述了属性（property）的各种特征。ECMA-262 定义这些特性是为了实现 JavaScript 引擎用的，因此在 JavaScript 中不能直接访问它们。为了表示特性是内部值，该规范把它们放在了方括号中，例如 `[[Enumerable]]`。尽管 ECMA-262 第 3 版的定义有些不同，但本书只参考第 5 版的描述。

ECMAScript 中有两种属性：数据属性和访问器属性。

1. 数据属性

数据属性包含一个数据值的位置。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。

- `[[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`。
- `[[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`。
- `[[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`。
- `[[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`。

对于像前面例子中那样直接在对象上定义的属性，它们的 `[[Configurable]]`、`[[Enumerable]]` 和 `[[Writable]]` 特性都被设置为 `true`，而 `[[Value]]` 特性被设置为指定的值。例如：

```
var person = {
  name: "Nicholas"
};
```

这里创建了一个名为 `name` 的属性，为它指定的值是 `"Nicholas"`。也就是说，`[[Value]]` 特性将被设置为 `"Nicholas"`，而对这个值的任何修改都将反映在这个位置。

要修改属性默认的特性，必须使用 ECMAScript 5 的 `Object.defineProperty()` 方法。这个方法接收三个参数：属性所在的对象、属性的名字和一个描述符对象。其中，描述符（descriptor）对象的属性必须是：`configurable`、`enumerable`、`writable` 和 `value`。设置其中的一或多个值，可以修改对应的特性值。例如：

```
var person = {};  
Object.defineProperty(person, "name", {  
    writable: false,  
    value: "Nicholas"  
});  
  
alert(person.name);    //"Nicholas"  
person.name = "Greg";  
alert(person.name);    //"Nicholas"
```

DataPropertiesExample01.htm

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。

类似的规则也适用于不可配置的属性。例如：

```
var person = {};  
Object.defineProperty(person, "name", {  
    configurable: false,  
    value: "Nicholas"  
});  
  
alert(person.name);    //"Nicholas"  
delete person.name;  
alert(person.name);    //"Nicholas"
```

DataPropertiesExample02.htm

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：

```
var person = {};  
Object.defineProperty(person, "name", {  
    configurable: false,  
    value: "Nicholas"  
});  
  
//抛出错误  
Object.defineProperty(person, "name", {  
    configurable: true,  
    value: "Nicholas"  
});
```

DataPropertiesExample03.htm

也就是说，可以多次调用 `Object.defineProperty()` 方法修改同一个属性，但在把 `configurable` 特性设置为 `false` 之后就会有限制了。

在调用 `Object.defineProperty()` 方法时，如果不指定，`configurable`、`enumerable` 和 `writable` 特性的默认值都是 `false`。多数情况下，可能都没有必要利用 `Object.defineProperty()` 方法提供的这些高级功能。不过，理解这些概念对理解 JavaScript 对象却非常有用。



IE8 是第一个实现 `Object.defineProperty()` 方法的浏览器版本。然而，这个版本的实现存在诸多限制：只能在 DOM 对象上使用这个方法，而且只能创建访问器属性。由于实现不彻底，建议读者不要在 IE8 中使用 `Object.defineProperty()` 方法。

2. 访问器属性

访问器属性不包含数据值；它们包含一对儿 `getter` 和 `setter` 函数（不过，这两个函数都不是必需的）。在读取访问器属性时，会调用 `getter` 函数，这个函数负责返回有效的值；在写入访问器属性时，会调用 `setter` 函数并传入新值，这个函数负责决定如何处理数据。访问器属性有如下 4 个特性。

- ❑ `[[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`。
- ❑ `[[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`。
- ❑ `[[Get]]`：在读取属性时调用的函数。默认值为 `undefined`。
- ❑ `[[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。请看下面的例子。

```
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
```

[AccessorPropertiesExample01.htm](#)

以上代码创建了一个 `book` 对象，并给它定义两个默认的属性：`_year` 和 `edition`。`_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。而访问器属性 `year` 则包含一个 `getter` 函数和一个 `setter` 函数。`getter` 函数返回 `_year` 的值，`setter` 函数通过计算来确定正确的版本。因此，把 `year` 属性修改为 2005 会导致 `_year` 变成 2005，而 `edition` 变为 2。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，尝试写入属性会被忽略。在严格模式下，尝试写入只指定了 `getter` 函数的属性会抛出错误。类似地，没有指定 `setter` 函数的属性也不能读，否则在非严格模式下会返回 `undefined`，而在严格模式下会抛出错误。

支持 ECMAScript 5 的这个方法的浏览器有 IE9+（IE8 只是部分实现）、Firefox 4+、Safari 5+、Opera 12+ 和 Chrome。在这个方法之前，要创建访问器属性，一般都使用两个非标准的方法：`__defineGetter__()` 和 `__defineSetter__()`。这两个方法最初是由 Firefox 引入的，后来 Safari 3、Chrome 1 和 Opera 9.5 也给出了相同的实现。使用这两个遗留的方法，可以像下面这样重写前面的例子。



```
var book = {
  _year: 2004,
  edition: 1
};

//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;
  }
});

book.year = 2005;
alert(book.edition); //2
```

[AccessorPropertiesExample02.htm](#)

在不支持 `Object.defineProperty()` 方法的浏览器中不能修改 `[[Configurable]]` 和 `[[Enumerable]]`。

6.1.2 定义多个属性

由于为对象定义多个属性的可能性很大，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;
        }
    }
}
});

```


MultiplePropertiesExample01.htm

以上代码在 book 对象上定义了两个数据属性（_year 和 edition）和一个访问器属性（year）。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。

支持 Object.defineProperty() 方法的浏览器有 IE9+、Firefox 4+、Safari 5+、Opera 12+ 和 Chrome。

6.1.3 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：属性所在的对象和要读取其描述符的属性名称。返回值是一个对象，如果是访问器属性，这个对象的属性有 configurable、enumerable、get 和 set；如果是数据属性，这个对象的属性有 configurable、enumerable、writable 和 value。例如：



```

var book = {};

Object.defineProperty(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");
alert(descriptor.value);           //2004
alert(descriptor.configurable);    //false

```

```
alert(typeof descriptor.get); // "undefined"

var descriptor = Object.getOwnPropertyDescriptor(book, "year");
alert(descriptor.value); // undefined
alert(descriptor.enumerable); // false
alert(typeof descriptor.get); // "function"
```

GetPropertyDescriptorExample01.htm

对于数据属性 `year`, `value` 等于最初的值, `configurable` 是 `false`, 而 `get` 等于 `undefined`。对于访问器属性 `year`, `value` 等于 `undefined`, `enumerable` 是 `false`, 而 `get` 是一个指向 `getter` 函数的指针。


在 JavaScript 中, 可以针对任何对象——包括 DOM 和 BOM 对象, 使用 `Object.getOwnPropertyDescriptor()` 方法。支持这个方法的浏览器有 IE9+、Firefox 4+、Safari 5+、Opera 12+ 和 Chrome。

6.2 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。

6.2.1 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 (本书后面还将讨论其他设计模式及其在 JavaScript 中的实现)。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, 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");
```

FactoryPatternExample01.htm

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。

6.2.2 构造函数模式

前几章介绍过, 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");
```

ConstructorPatternExample01.htm

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的大写字母开头，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。

在前面例子的最后，`person1` 和 `person2` 分别保存着 `Person` 的一个不同的实例。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
```

对象的 `constructor` 属性最初是用来标识对象类型的。但是，提到检测对象类型，还是 `instanceof` 操作符要更可靠一些。我们在这个例子中创建的所有对象既是 `Object` 的实例，同时也是 `Person` 的实例，这一点通过 `instanceof` 操作符可以得到验证。

```
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`（详细内容稍后讨论）。



以这种方式定义的构造函数是定义在 Global 对象(在浏览器中是 window 对象)中的。第 8 章将详细讨论浏览器对象模型(BOM)。

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。例如,前面例子中定义的 Person() 函数可以通过下列任何一种方式来调用。

```
// 当作构造函数使用
var person = new Person("Nicholas", 29, "Software Engineer");
person.sayName(); // "Nicholas"

// 作为普通函数调用
Person("Greg", 27, "Doctor"); // 添加到 window
window.sayName(); // "Greg"

// 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"
```

[*ConstructorPatternExample02.htm*](#)

这个例子中的前两行代码展示了构造函数的典型用法,即使用 new 操作符来创建一个新对象。接下来的两行代码展示了不使用 new 操作符调用 Person() 会出现什么结果:属性和方法都被添加给 window 对象了。有读者可能还记得,当在全局作用域中调用一个函数时, this 对象总是指向 Global 对象(在浏览器中就是 window 对象)。因此,在调用完函数之后,可以通过 window 对象来调用 sayName() 方法,并且还返回了 "Greg"。最后,也可以使用 call() (或者 apply()) 在某个特殊对象的作用域中调用 Person() 函数。这里是在对象 o 的作用域中调用的,因此调用后 o 就拥有了所有属性和 sayName() 方法。

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。因此，大可像下面这样，通过把函数定义转移到构造函数外部来解决这个问题。

```
function Person(name, age, job){
    this.name = name;
    this.age = age;
    this.job = job;
    this.sayName = sayName;
}

function sayName(){
    alert(this.name);
}

var person1 = new Person("Nicholas", 29, "Software Engineer");
var person2 = new Person("Greg", 27, "Doctor");
```

ConstructorPatternExample03.htm

在这个例子中，我们把 sayName() 函数的定义转移到了构造函数外部。而在构造函数内部，我们将 sayName 属性设置成等于全局的 sayName 函数。这样一来，由于 sayName 包含的是一个指向函数的指针，因此 person1 和 person2 对象就共享了在全局作用域中定义的同一个 sayName() 函数。这样做确实解决了两个函数做同一件事的问题，可是新问题又来了：在全局作用域中定义的函数实际上只能被某个对象调用，这让全局作用域有点名不副实。而更让人无法接受的是：如果对象需要定义很多方法，那么就要定义很多个全局函数，于是我们这个自定义的引用类型就丝毫没有封装性可言了。好在，这些问题可以通过使用原型模式来解决。

6.2.3 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

```

PrototypePatternExample01.htm

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,图 6-1 展示了各个对象之间的关系。

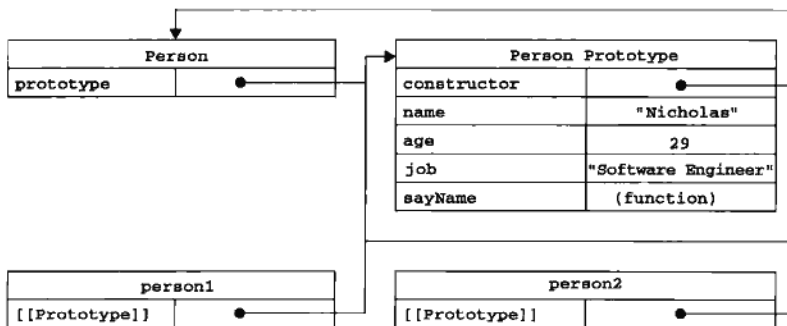


图 6-1

图 6-1 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。

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`，如下所示：

```
alert(Person.prototype.isPrototypeOf(person1)); //true
alert(Person.prototype.isPrototypeOf(person2)); //true
```

这里，我们用原型对象的 `isPrototypeOf()` 方法测试了 `person1` 和 `person2`。因为它们内部都有一个指向 `Person.prototype` 的指针，因此都返回了 `true`。

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。例如：

```
alert(Object.getPrototypeOf(person1) == Person.prototype); //true
alert(Object.getPrototypeOf(person1).name); //"Nicholas"
```

这里的第一行代码只是确定 `Object.getPrototypeOf()` 返回的对象实际就是这个对象的原型。第二行代码取得了原型对象中 `name` 属性的值，也就是“Nicholas”。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。支持这个方法的浏览器有 IE9+、Firefox 3.5+、Safari 5+、Opera 12+ 和 Chrome。

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
```

PrototypePatternExample02.htm

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
```

PrototypePatternExample03.htm

在这个修改后的例子中，我们使用 delete 操作符删除了 person1.name，之前它保存的"Greg"值屏蔽了同名的原型属性。把它删除以后，就恢复了对原型中 name 属性的连接。因此，接下来再调用 person1.name 时，返回的就是原型中 name 属性的值了。

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。来看下面这个例子。

```
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

person1.name = "Greg";
alert(person1.name); // "Greg"——来自实例
alert(person1.hasOwnProperty("name")); //true

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

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

```

通过使用 `hasOwnProperty()` 方法，什么时候访问的是实例属性，什么时候访问的是原型属性就一清二楚了。调用 `person1.hasOwnProperty("name")` 时，只有当 `person1` 重写 `name` 属性后才会返回 `true`，因为只有这时候 `name` 才是一个实例属性，而非原型属性。图 6-2 展示了上面例子在不同情况下的实现与原型的关系（为了简单起见，图中省略了与 `Person` 构造函数的关系）。

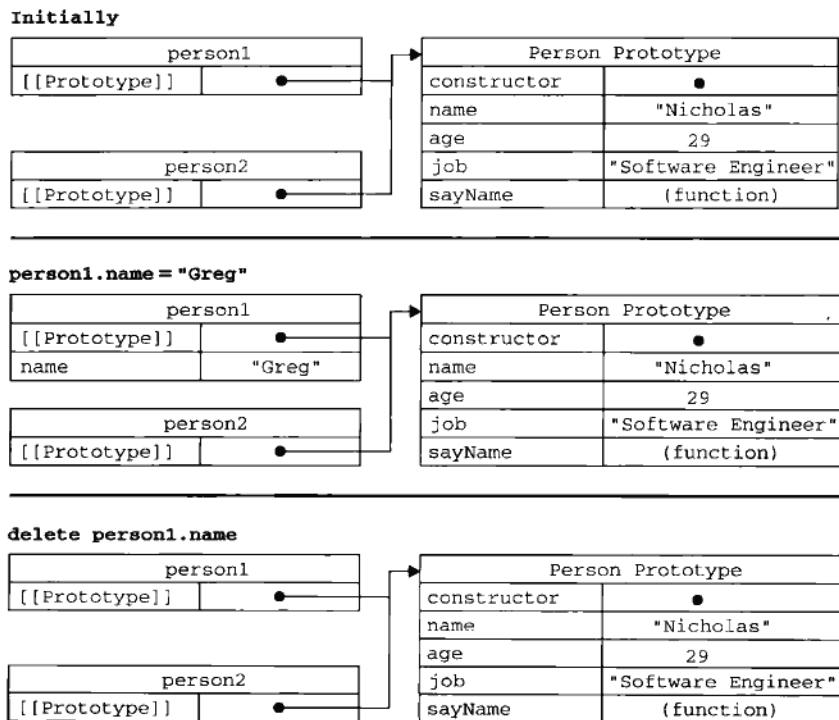


图 6-2



ECMAScript 5 的 `Object.getOwnPropertyDescriptor()` 方法只能用于实例属性,要取得原型属性的描述符,必须直接在原型对象上调用 `Object.getOwnPropertyDescriptor()` 方法。

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
```

PrototypePatternExample04.htm

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, 如下所示。



```
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。下面来看一看上面定义的函数 `hasPrototypeProperty()` 的用法。


```

function Person(){
}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function(){
    alert(this.name);
};

var person = new Person();
alert(hasPrototypeProperty(person, "name")); //true

person.name = "Greg";
alert(hasPrototypeProperty(person, "name")); //false


```

PrototypePatternExample05.htm

在这里，name 属性先是存在于原型中，因此 hasPrototypeProperty() 返回 true。当在实例中重写 name 属性后，该属性就存在于实例中了，因此 hasPrototypeProperty() 返回 false。即使原型中仍然有 name 属性，但由于现在实例中也有了这个属性，因此原型中的 name 属性就用不到了。

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（即将 [[Enumerable]] 标记的属性）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。

IE 早期版本的实现中存在一个 bug，即屏蔽不可枚举属性的实例属性不会出现在 for-in 循环中。例如：



```

var o = {
    toString : function(){
        return "My Object";
    }
};

for (var prop in o){
    if (prop == "toString"){
        alert("Found toString");    //在 IE 中不会显示
    }
}

```

PrototypePatternExample06.htm

当以上代码运行时，应该会显示一个警告框，表明找到了 toString() 方法。这里的对象 o 定义了一个名为 toString() 的方法，该方法屏蔽了原型中（不可枚举）的 toString() 方法。在 IE 中，由于其实现认为原型的 toString() 方法被打上了 [[Enumerable]] 标记就应该跳过该属性，结果我们就不会看到警告框。该 bug 会影响默认不可枚举的所有属性和方法，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。这个方法接收一个对象作为参数,返回一个包含所有可枚举属性的字符串数组。例如:

```
function Person(){
}

Person.prototype.name = "Nicholas";
Person.prototype.age = 29;
Person.prototype.job = "Software Engineer";
Person.prototype.sayName = function(){
    alert(this.name);
};

var keys = Object.keys(Person.prototype);
alert(keys);           //"name,age,job,sayName"

var p1 = new Person();
p1.name = "Rob";
p1.age = 31;
var p1keys = Object.keys(p1);
alert(p1keys);        //"name,age"
```

ObjectKeysExample01.htm

这里,变量 `keys` 中将保存一个数组,数组中是字符串 "name"、"age"、"job" 和 "sayName"。这个顺序也是它们在 `for-in` 循环中出现的顺序。如果是通过 `Person` 的实例调用,则 `Object.keys()` 返回的数组只包含 "name" 和 "age" 这两个实例属性。

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。

```
var keys = Object.getOwnPropertyNames(Person.prototype);
alert(keys);    //"constructor,name,age,job,sayName"
```

ObjectPropertyNamesExample01.htm

注意结果中包含了不可枚举的 `constructor` 属性。`Object.keys()` 和 `Object.getOwnPropertyNames()` 方法都可以用来替代 `for-in` 循环。支持这两个方法的浏览器有 IE9+、Firefox 4+、Safari 5+、Opera 12+ 和 Chrome。

3. 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,更常见的做法是用一个包含所有属性和方法的对象字面量来重写整个原型对象,如下面的例子所示。

```
function Person(){
}

Person.prototype = {
    name : "Nicholas",
    age : 29,
    job: "Software Engineer",
    sayName : function () {
        alert(this.name);
    }
};
```

PrototypePatternExample07.htm

在上面的代码中，我们将 `Person.prototype` 设置为等于一个以对象字面量形式创建的新对象。最终结果相同，但有一个例外：`constructor` 属性不再指向 `Person` 了。前面曾经介绍过，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。此时，尽管 `instanceof` 操作符还能返回正确的结果，但通过 `constructor` 已经无法确定对象的类型了，如下所示。

```
var friend = new Person();

alert(friend instanceof Object);    //true
alert(friend instanceof Person);    //true
alert(friend.constructor == Person); //false
alert(friend.constructor == Object); //true
```

PrototypePatternExample07.htm

在此，用 `instanceof` 操作符测试 `Object` 和 `Person` 仍然返回 `true`，但 `constructor` 属性则等于 `Object` 而不等于 `Person` 了。如果 `constructor` 的值真的很重要，可以像下面这样特意将它设置回适当的值。

```
function Person(){
}

Person.prototype = {
  constructor : Person,
  name : "Nicholas",
  age : 29,
  job : "Software Engineer",
  sayName : function () {
    alert(this.name);
  }
};
```

PrototypePatternExample07.htm

以上代码特意包含了一个 `constructor` 属性，并将它的值设置为 `Person`，从而确保了通过该属性能够访问到适当的值。

注意，以这种方式重置 `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()`。

```
function Person(){
}

Person.prototype = {
  name : "Nicholas",
  age : 29,
  job : "Software Engineer",
  sayName : function () {
    alert(this.name);
  }
};
```

//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
});
```

4. 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。请看下面的例子。

```
var friend = new Person();

Person.prototype.sayHi = function(){
    alert("hi");
};

friend.sayHi();    //"hi" (没有问题!)
```

PrototypePatternExample09.htm

以上代码先创建了 `Person` 的一个实例，并将其保存在 `person` 中。然后，下一条语句在 `Person.prototype` 中添加了一个方法 `sayHi()`。即使 `person` 实例是在添加新方法之前创建的，但它仍然可以访问这个新方法。其原因可以归结为实例与原型之间的松散连接关系。当我们调用 `person.sayHi()` 时，首先会在实例中搜索名为 `sayHi` 的属性，在没找到的情况下，会继续搜索原型。因为实例与原型之间的连接只不过是一个指针，而非一个副本，因此就可以在原型中找到新的 `sayHi` 属性并返回保存在那里的函数。

尽管可以随时为原型添加属性和方法，并且修改能够立即在所有对象实例中反映出来，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。请记住：实例中的指针仅指向原型，而不指向构造函数。看下面的例子。

```
function Person(){
}

var friend = new Person();

Person.prototype = {
    constructor: Person,
    name: "Nicholas",
    age: 29,
    job: "Software Engineer",
    sayName: function () {
        alert(this.name);
    }
};

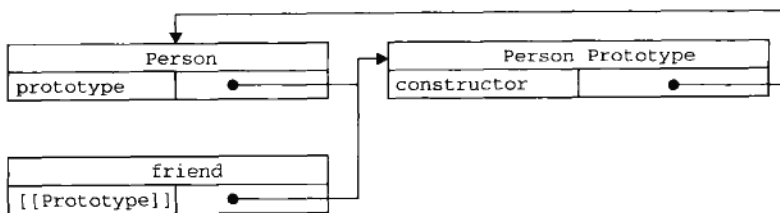
friend.sayName();    //error
```

PrototypePatternExample10.htm

在这个例子中，我们先创建了 `Person` 的一个实例，然后又重写了其原型对象。然后在调用 `friend.sayName()` 时发生了错误，因为 `friend` 指向的原型中不包含以该名字命名的属性。图 6-3 展

示了这个过程的内幕。

重写原型对象之前



重写原型对象之后

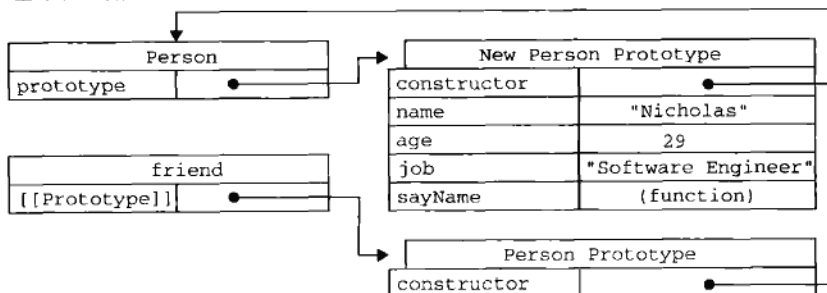


图 6-3

从图 6-3 可以看出，重写原型对象切断了现有原型与任何之前已经存在的对象实例之间的联系；它们引用的仍然是最初的原型。

5. 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。

6. 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。

原型中所有属性是被很多实例共享的，这种共享对于函数非常合适。对于那些包含基本值的属性倒也说得过去，毕竟（如前面的例子所示），通过在实例上添加一个同名属性，可以隐藏原型中的对应属性。然而，对于包含引用类型值的属性来说，问题就比较突出了。来看下面的例子。

```
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
```

PrototypePatternExample12.htm

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。

6.2.4 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。下面的代码重写了前面的例子。

```
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);
    }
}

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
```

HybridPatternExample01.htm

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。

6.2.5 动态原型模式

有其他 OO 语言经验的开发人员在看到独立的构造函数和原型时，很可能会感到非常困惑。动态原型模式正是致力于解决这个问题一个方案，它把所有信息都封装在了构造函数中，而通过在构造函数中初始化原型（仅在必要的情况下），又保持了同时使用构造函数和原型的优点。换句话说，可以通过检查某个应该存在的方法是否有效，来决定是否需要初始化原型。来看一个例子。

```
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);
    };

}

var friend = new Person("Nicholas", 29, "Software Engineer");
friend.sayName();
```

DynamicPrototypeExample01.htm

注意构造函数代码中加粗的部分。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。因此，这种方法确实可以说非常完美。其中，`if` 语句检查的可以是初始化之后应该存在的任何属性或方法——不必用一大堆 `if` 语句检查每个属性和每个方法；只要检查其中一个即可。对于采用这种模式创建的对象，还可以使用 `instanceof` 操作符确定它的类型。



使用动态原型模式时，不能使用对象字面量重写原型。前面已经解释过了，如果在已经创建了实例的情况下重写原型，那么就会切断现有实例与新原型之间的联系。

6.2.6 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。这种模式的基本思想是创建一个函数，该函数的作用仅仅是封装创建对象的代码，然后再返回新创建的对象；但从表面上看，这个函数又很像是典型的构造函数。下面是一个例子。

```
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"
```

HybridFactoryPatternExample01.htm

在这个例子中，`Person` 函数创建了一个新对象，并以相应的属性和方法初始化该对象，然后又返回了这个对象。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。构造函数在不返回值的情况下，默认会返回新对象实例。而通过在构造函数的末尾添加

一个 return 语句，可以重写调用构造函数时返回的值。

这个模式可以在特殊的情况下用来为对象创建构造函数。假设我们想创建一个具有额外方法的特殊数组。由于不能直接修改 Array 构造函数，因此可以使用这个模式。



```
function SpecialArray(){

    //创建数组
    var values = new Array();

    //添加值
    values.push.apply(values, arguments);

    //添加方法
    values.toPipedString = function(){
        return this.join("|");
    };

    //返回数组
    return values;
}

var colors = new SpecialArray("red", "blue", "green");
alert(colors.toPipedString()); // "red|blue|green"
```

HybridFactoryPatternExample02.htm

在这个例子中，我们创建了一个名叫 SpecialArray 的构造函数。在这个函数内部，首先创建了一个数组，然后 push() 方法（用构造函数接收到的所有参数）初始化了数组的值。随后，又给数组实例添加了一个 toPipedString() 方法，该方法返回以竖线分割的数组值。最后，将数组以函数值的形式返回。接着，我们调用了 SpecialArray 构造函数，向其中传入了用于初始化数组的值，此后又调用了 toPipedString() 方法。

关于寄生构造函数模式，有一点需要说明：首先，返回的对象与构造函数或者与构造函数的原型属性之间没有关系；也就是说，构造函数返回的对象与在构造函数外部创建的对象没有什么不同。为此，不能依赖 instanceof 操作符来确定对象类型。由于存在上述问题，我们建议在可以使用其他模式的情况下，不要使用这种模式。

6.2.7 稳妥构造函数模式

道格拉斯·克罗斯福德（Douglas Crockford）发明了 JavaScript 中的稳妥对象（durable objects）这个概念。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），或者在防止数据被其他应用程序（如 Mashup 程序）改动时使用。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

//添加方法
o.sayName = function(){
    alert(name);
};

//返回对象
return o;
}

```

注意,在以这种模式创建的对象中,除了使用 `sayName()` 方法之外,没有其它办法访问 `name` 的值。可以像下面使用稳妥的 `Person` 构造函数。

```

var friend = Person("Nicholas", 29, "Software Engineer");
friend.sayName(); // "Nicholas"

```

这样,变量 `person` 中保存的是一个稳妥对象,而除了调用 `sayName()` 方法外,没有别的方式可以访问其数据成员。即使有其他代码会给这个对象添加方法或数据成员,但也不可能有别的办法访问传入到构造函数中的原始数据。稳妥构造函数模式提供的这种安全性,使得它非常适合在某些安全执行环境——例如, `ADsafe` (www.adsafe.org) 和 `Caja` (<http://code.google.com/p/google-caja/>) 提供的环境下使用。



与寄生构造函数模式类似,使用稳妥构造函数模式创建的对象与构造函数之间也没有什么关系,因此 `instanceof` 操作符对这种对象也没有意义。

6.3 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。

6.3.1 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

```

PrototypeChainingExample01.htm

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 6-4 所示。

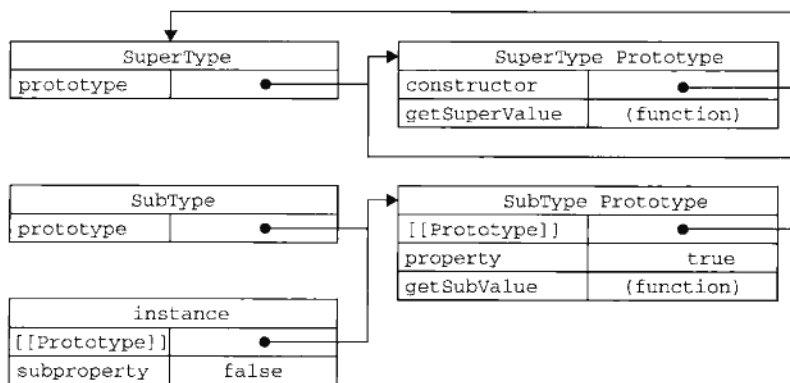


图 6-4

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^①。

通过实现原型链，本质上扩展了本章前面介绍的原型搜索机制。读者大概还记得，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。图 6-5 为我们展示了该例子中完整的原型链。

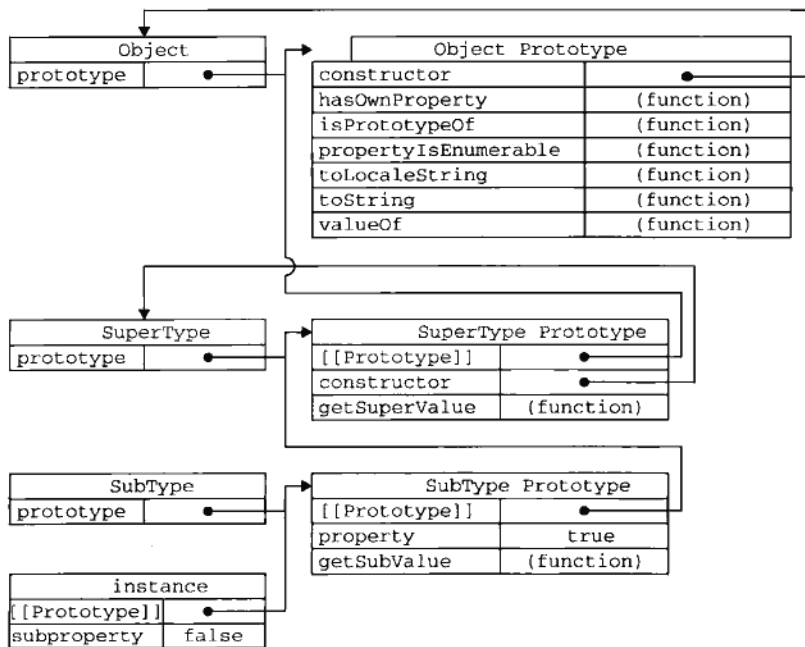



图 6-5

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

^① 实际上，不是 `SubType` 的原型的 `constructor` 属性被重写了，而是 `SubType` 的原型指向了另一个对象——`SuperType` 的原型，而这个原型对象的 `constructor` 属性指向的是 `SuperType`。

2. 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`。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
```

PrototypeChainingExample01.htm

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`。


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，因此 `isPrototypeOf()` 方法也会返回 `true`，如下所示。

```
alert(Object.prototype.isPrototypeOf(instance));           //true
alert(SuperType.prototype.isPrototypeOf(instance));         //true
alert(SubType.prototype.isPrototypeOf(instance));           //true
```

PrototypeChainingExample01.htm

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
```

PrototypeChainingExample02.htm

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。

还有一点需要提醒读者，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!
```

PrototypeChainingExample03.htm

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。想必大家还记得，我们前面介绍过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"

```

PrototypeChainingExample04.htm

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。

6.3.2 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相当简单,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"

```

ConstructorStealingExample01.htm

代码中加背景的那一行代码“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。

```
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
```

[ConstructorStealingExample02.htm](#)

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。

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例化对实例属性的继承。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.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

```

CombinationInheritanceExample01.htm

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。而且，instanceof 和 isPrototypeOf() 也能够用于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·克洛克福德在 2006 年写了一篇文章，题为 Prototypal Inheritance in JavaScript（JavaScript 中的原型式继承）。在这篇文章中，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。

```

function object(o){
    function F(){}
    F.prototype = o;
    return new F();
}

```

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。

```
var person = {
    name: "Nicholas",
    friends: ["Shelby", "Court", "Van"]
};

var anotherPerson = object(person);
anotherPerson.name = "Greg";
anotherPerson.friends.push("Rob");


var yetAnotherPerson = object(person);
yetAnotherPerson.name = "Linda";
yetAnotherPerson.friends.push("Barbie");

alert(person.friends); // "Shelby,Court,Van,Rob,Barbie"
```

PrototypallInheritanceExample01.htm

克洛克福德主张的这种原型式继承，要求你必须有一个对象可以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` 以及 `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"
```

PrototypallInheritanceExample02.htm

`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：

```
var person = {
    name: "Nicholas",
    friends: ["Shelby", "Court", "Van"]
};
```

```
var anotherPerson = Object.create(person, {
    name: {
        value: "Greg"
    }
});

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

PrototypalInheritanceExample03.htm

支持 `Object.create()` 方法的浏览器有 IE9+、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 (parasitic) 继承是与原型式继承紧密相关的一种思路，并且同样也是由克洛克福德推而广之的。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，最后再像真地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。

```
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
}
```

在这个例子中，`createAnother()` 函数接收了一个参数，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：

```
var person = {
    name: "Nicholas",
    friends: ["Shelby", "Court", "Van"]
};

var anotherPerson = createAnother(person);
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()` 函数不是必需的；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。再来看一下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 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：



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

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

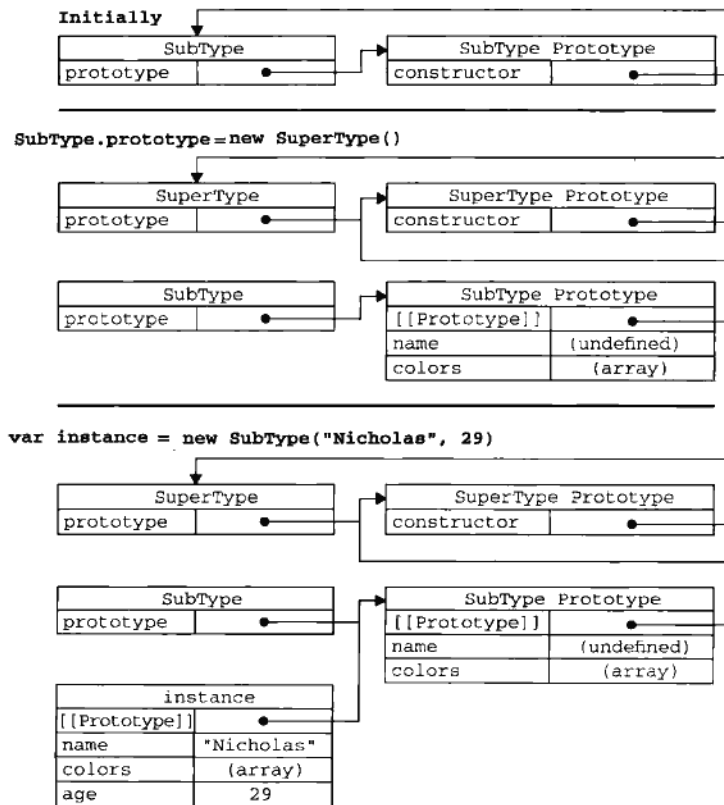
function SubType(name, age){
    SuperType.call(this, name);

    this.age = age;
}

inheritPrototype(SubType, SuperType);

SubType.prototype.sayAge = function(){
    alert(this.age);
};
```

ParasiticCombinationInheritanceExample01.htm



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。



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` 属性来指定那些应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优点与一身，是实现基于类型继承的最有效方式。