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

|  |
| --- |
| var person = new Object();  person.name = "Nicholas";  person.age = 29;  person.job = "Software Engineer";  person.sayName = function(){  alert(this.name);  }; |
| 对象字面量成为创建这种对象的首选模式。  var person = {  name: "Nicholas",  age: 29,  job: "Software Engineer",  sayName: function(){  alert(this.name);  }  }; |

## 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

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。

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"  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"  });  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。 |

### 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 |

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，只指定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。

## 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

|  |
| --- |
| 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;  }  }  }  }); |

## 读取属性特性

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。

|  |
| --- |
| 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");  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" |

# 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。

## 工厂模式

|  |
| --- |
| 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

|  |
| --- |
| 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");  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。  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：创建一个新对象；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）； 执行构造函数中的代码（为这个新对象添加属性）；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 |

### 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"

### 构造函数的问题

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  } |

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。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");  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 |

## 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。

|  |
| --- |
| 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  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。 |

### 理解原型对象