logo.png

面向对象

H5 web前端

[www.bufanui.com](http://www.bufanui.com)

# 面向对象的程序设计

回顾：

1.基本数据类型和引用类型

Number Boolean string null undefined

NaN 🡺 NOT a number ;比如10/’a’== NaN , ~~NaN==NaN?~~

1. 引用类型

object array function json

1. 所有的应用类型都是object对象
2. 所有的对象都是function
3. 因为在js中是通过function去声明一个对象类型的。
4. 值类型是存储在栈（stack）中的简单数据，也就是说，它们的值直接存储在变量访问的位置。
5. 存储在堆（heap）中的对象，也就是说，存储在变量处的值是一个指针（point），指向存储对象的内存处。

思考：修改对象内容，会修改引用对象内容吗？

1. 基本数据类型不会改变
2. 引用类型会发生改变

继承 封装 多态

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，我们可以把 ECMAScript 的对象想象成散列表：无非就是一组键值对，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原生类型（Objcet Math Date Array ），也可以是开发人员定义的类型。

## 1.1 理解对象

我们曾经介绍过，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);  }; |

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。

### 1.1.1 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两对儿方括号中，例如 [[Enumerable]] 。

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。

**严格模式**：

为什么使用严格模式:

消除Javascript语法的一些不合理、不严谨之处，减少一些怪异行为;

* 消除代码运行的一些不安全之处，保证代码运行的安全；
* 提高编译器效率，增加运行速度；
* 为未来新版本的Javascript做好铺垫。

"严格模式"体现了Javascript更合理、更安全、更严谨的发展方向，包括IE 10在内的主流浏览器，都已经支持它，许多大项目已经开始全面拥抱它。

另一方面，同样的代码，在"严格模式"中，可能会有不一样的运行结果；一些在"正常模式"下可以运行的语句，在"严格模式"下将不能运行。

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" |

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（use strict），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" |

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"  }); |

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。

**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 = {  sale:.75,  \_price: 100,  edition: 1  };  Object.defineProperty(book, "newPrice", {  get: function(){  return this.\_price;  },  set: function(v){  this.\_price = v\*this.sale;  }  });  book.newPrice = 200;  alert(book.\_price); |

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，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。

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 |

案例：

商品打折方案。

### 1.1.2 定义多个属性

为对象定义多个属性的可能性很大，ECMAScript 5 又定义了一个 **Object.definePro-perties()** 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：

### 1.1.2 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 ,默认的值都为false。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;  }  }  }  });  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" |

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。

## 1.2 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。

## 1.2.1 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。

|  |
| --- |
| 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）。

## 1.2.2 构造函数模式

通过构造函数方式创建对象。

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

**区别：**

*  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。

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 instan-

ceof 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。

**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"

**举例：工具类封装**

1. **函数封装**
2. **对象封装**

**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。——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  } |

证明：

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

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"); |

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，但这里可以全局调用；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。

这些问题可以通过使用**原型模式**来解决。

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 |