链滴

JavaScript 定义类和对象的几种方式

作者：zhuangyan
原文链接：https://ld246.com/article/1374897605458
来源网站：链滴

《javaScript高级程序设计》(人民邮电出版社)第三章第五节内容.

使用预定义对象只是面向对象语言的能力的一部分，它真正强大之处在于能够创建自专用的类和对象。
ECMAScript 拥有很多创建对象或类的方法。

<h2>工厂方式</h2>
<h3>原始的方式</h3>
因为对象的属性可以在对象创建后动态定义，所有许多开发者都在 JavaScript 最初引入时编写似下面的代码：
<pre>var oCar = new Object;
oCar.color = "blue";
oCar.doors = 4;
oCar.mpg = 25;
oCar.showColor = function() {
  alert(this.color);
};
</pre>
TIY
在上面的代码中，创建对象 car。然后给它设置几个属性：它的颜色是蓝色，有四个门，每加仑可以跑 25 英里。最后一个属性实际上是指向函数的指针，意味着该属性是个方法。执行这段代码后就可以使用对象 car。
不过这里有一个问题，就是可能需要创建多个 car 的实例。
<h3>解决方案：工厂方式</h3>
要解决该问题，开发者创造了能创建并返回特定类型的对象的工厂函数（factory function）。
例如，函数 createCar() 可用于封装前面列出的创建 car 对象的操作：
<pre>function createCar() {
  var oTempCar = new Object;
  oTempCar.color = "blue";
  oTempCar.doors = 4;
  oTempCar.mpg = 25;
  oTempCar.showColor = function() {
    alert(this.color);
  };
  return oTempCar;
}

var oCar1 = createCar();
var oCar2 = createCar();
</pre>

TIY
在这里，第一个例子中的所有代码都包含在 createCar() 函数中。此外，还有一行额外的代码，回 car 对象（oTempCar）作为函数值。调用此函数，将创建新对象，并赋予它所有必要的属性，复出一个我们在前面说明过的 car 对象。因此，通过这种方法，我们可以很容易地创建 car 对象的两个本（oCar1 和 oCar2），它们的属性完全一样。
<h3>为函数传递参数</h3>
我们还可以修改 createCar() 函数，给它传递各个属性的默认值，而不是简单地赋予属性默认值

```
</p>
 <pre>function createCar(sColor,iDoors,iMpg) {
  var oTempCar = new Object;
  oTempCar.color = sColor;
  oTempCar.doors = iDoors;
  oTempCar.mpg = iMpg;
  oTempCar.showColor = function() {
    alert(this.color);
  };
  return oTempCar;
}

var oCar1 = createCar(&amp;quot;red&amp;quot;
4,23);
var oCar2 = createCar(&amp;quot;blue&amp;quot;
3,25);

oCar1.showColor();       //输出 &amp;quot;red
quot;
oCar2.showColor();       //输出 &amp;quot;blue
quot;
</pre>

 <p><a href="http://www.w3school.com.cn/tiy/t.asp?f=jseg_pro_object_defining_factory_par
digm_arguments">TIY</a></p>
 <p>给 createCar() 函数加上参数，即可为要创建的 car 对象的 color、doors 和 mpg 属性赋值。
使两个对象具有相同的属性，却有不同的属性值。</p>
 <h3>在工厂函数外定义对象的方法</h3>
 <p>虽然 ECMAScript 越来越正式化，但创建对象的方法却被置之不理，且其规范化至今还遭人反
。一部分是语义上的原因（它看起来不像使用带有构造函数 new 运算符那么正规），一部分是功能
的原因。功能原因在于用这种方式必须创建对象的方法。前面的例子中，每次调用函数 createCar()
都要创建新函数 showColor()，意味着每个对象都有自己的 showColor() 版本。而事实上，每个对
都共享同一个函数。</p>
 <p>有些开发者在工厂函数外定义对象的方法，然后通过属性指向该方法，从而避免这个问题：</p>

 <pre><code>function showColor() {
  alert(this.color);
}</code>

function createCar(sColor,iDoors,iMpg) {
var oTempCar = new Object;
oTempCar.color = sColor;
oTempCar.doors = iDoors;
oTempCar.mpg = iMpg;
<code>oTempCar.showColor = showColor;</code>
return oTempCar;
}

var oCar1 = createCar(&amp;quot;red&amp;quot;;
```

```
4,23);
var oCar2 = createCar(&amp;quot;blue&amp;quot;
3,25);

oCar1.showColor();        //输出 &amp;quot;red
quot;
oCar2.showColor();        //输出 &amp;quot;blue
quot;
</pre>

 <p><a href="http://www.w3school.com.cn/tiy/t.asp?f=jseg_pro_object_defining_factory_par
digm_method">TIY</a></p>
 <p>在上面这段重写的代码中，在函数 createCar() 之前定义了函数 showColor()。在 createCar()
部，赋予对象一个指向已经存在的 showColor() 函数的指针。从功能上讲，这样解决了重复创建函数
象的问题；但是从语义上讲，该函数不太像是对象的方法。</p>
 <p>所有这些问题都引发了<em>开发者定义</em>的构造函数的出现。</p>
</div>
<div>
 <h2>构造函数方式</h2>
 <p>创建构造函数就像创建工厂函数一样容易。第一步选择类名，即构造函数的名字。根据惯例，
个名字的首字母大写，以使它与首字母通常是小写的变量名分开。除了这点不同，构造函数看起来很
工厂函数。请考虑下面的例子：</p>
 <pre>function Car(sColor,iDoors,iMpg) {
  <code>this</code>.color = sColor;
  <code>this</code>.doors = iDoors;
  <code>this</code>.mpg = iMpg;
  <code>this</code>.showColor = function() {
   alert(this.color);
  };
}

var oCar1 = <code>new</code> Car(&amp;quot;red&amp;quot;,4,23);
var oCar2 = <code>new</code> Car(&amp;quot;blue&amp;quot;,3,25);
</pre>

 <p><a href="http://www.w3school.com.cn/tiy/t.asp?f=jseg_pro_object_defining_constructor
paradigm">TIY</a></p>
 <p>下面为您解释上面的代码与工厂方式的差别。首先在构造函数内没有创建对象，而是使用 this
键字。使用 new 运算符构造函数时，在执行第一行代码前先创建一个对象，只有用 this 才能访问该
象。然后可以直接赋予 this 属性，默认情况下是构造函数的返回值（不必明确使用 return 运算符）
</p>
 <p>现在，用 new 运算符和类名 Car 创建对象，就更像 ECMAScript 中一般对象的创建方式了。</
>
 <p>你也许会问，这种方式在管理函数方面是否存在于前一种方式相同的问题呢？是的。</p>
 <p>就像工厂函数，构造函数会重复生成函数，为每个对象都创建独立的函数版本。不过，与工厂
数相似，也可以用外部函数重写构造函数，同样地，这么做语义上无任何意义。这正是下面要讲的原
方式的优势所在。</p>
</div>
<div>
 <h2>原型方式</h2>
 <p>该方式利用了对象的 prototype 属性，可以把它看成创建新对象所依赖的原型。</p>
 <p>这里，首先用空构造函数来设置类名。然后所有的属性和方法都被直接赋予 prototype 属性。
```

们重写了前面的例子，代码如下：
<pre>function Car() {
}

Car.prototype.color = "blue";
Car.prototype.doors = 4;
Car.prototype.mpg = 25;
Car.prototype.showColor = function() {
alert(this.color);
};

var oCar1 = new Car();
var oCar2 = new Car();
</pre>

TIY
在这段代码中，首先定义构造函数（Car），其中无任何代码。接下来的几行代码，通过给 Car prototype 属性添加属性去定义 Car 对象的属性。调用 new Car() 时，原型的所有属性都被立即赋要创建的对象，意味着所有 Car 实例存放的都是指向 showColor() 函数的指针。从语义上讲，所有性看起来都属于一个对象，因此解决了前面两种方式存在的问题。
此外，使用这种方式，还能用 instanceof 运算符检查给定变量指向的对象的类型。因此，下面代码将输出 TRUE：
<pre>alert(oCar1 instanceof Car); //输出 "true"</pre>
<h3>原型方式的问题</h3>
原型方式看起来是个不错的解决方案。遗憾的是，它并不尽如人意。
首先，这个构造函数没有参数。使用原型方式，不能通过给构造函数传递参数来初始化属性的，因为 Car1 和 Car2 的 color 属性都等于 "blue"，doors 属性都等于 4，mpg 属性都于 25。这意味着必须在对象创建后才能改变属性的默认值，这点很令人讨厌，但还没完。真正的问出现在属性指向的是对象，而不是函数时。函数共享不会造成问 题，但对象却很少被多个实例共享。思考下面的例子：
<pre>function Car() {
}

Car.prototype.color = "blue";
Car.prototype.doors = 4;
Car.prototype.mpg = 25;
<code>Car.prototype.drivers = new Array("Mike
quot;,"John");   <
code>
Car.prototype.showColor = function() {
alert(this.color);
};

var oCar1 = new Car();
var oCar2 = new Car();

<code>oCar1.drivers.push("Bill"
;</code>

alert(oCar1.drivers); //输出 "Mike,John,Bill
quot;

alert(oCar2.drivers); //输出 "Mike,John,Bill
quot;

</pre>

TIY
上面的代码中，属性 drivers 是指向 Array 对象的指针，该数组中包含两个名字 "Mike" 和 "John"。由于 drivers 是引用值，Car 的两个实例都指向同一个数组。这意味着 oCar1.drivers 添加值 "Bill"，在 oCar2.drivers 中也能看到。输出这两个指针中的任何个，结果都是显示字符串 "Mike,John,Bill"。
由于创建对象时有这么多问题，你一定会想，是否有种合理的创建对象的方法呢？答案是有，要联合使用构造函数和原型方式。

<h2>混合的构造函数/原型方式</h2>
联合使用构造函数和原型方式，就可像用其他程序设计语言一样创建对象。这种概念非常简单即用构造函数定义对象的所有非函数属性，用原型方式定义对象的函数属性（方法）。结果是，所有数都只创建一次，而每个对象都具有自己的对象属性实例。
我们重写了前面的例子，代码如下：
<pre>function Car(sColor,iDoors,iMpg) {
 this.color = sColor;
 this.doors = iDoors;
 this.mpg = iMpg;
 this.drivers = new Array("Mike","John");
}

Car.prototype.showColor = function() {

alert(this.color);

};

var oCar1 = new Car("red"
4,23);

var oCar2 = new Car("blue"
3,25);

oCar1.drivers.push("Bill"
;

alert(oCar1.drivers); //输出 "Mike,John,Bill
quot;

alert(oCar2.drivers); //输出 "Mike,John
quot;

</pre>

TIY

现在就更像创建一般对象了。所有的非函数属性都在构造函数中创建，意味着又能够用构造函的参数赋予属性默认值了。因为只创建 showColor() 函数的一个实例，所以没有内存浪费。此外，给 Car1 的 drivers 数组添加 "Bill" 值，不会影响到 oCar2 的数组，所以输出这些数组的时，oCar1.drivers 显示的是 "Mike,John,Bill"，而 oCar2.drivers 显示的是 "Mike John"。因为使用了原型方式，所以仍然能利用 instanceof 运算符来判断对象的类型。
这种方式是 ECMAScript 采用的主要方式，它具有其他方式的特性，却没有他们的副作用。不，有些开发者仍觉得这种方法不够完美。

<h2>动态原型方法</h2>
对于习惯使用其他语言的开发者来说，使用混合的构造函数/原型方式感觉不那么和谐。毕竟，义类时，大多数面向对象语言都对属性和方法进行了视觉上的封装。请考虑下面的 Java 类：
<pre>class Car {
public String color = "blue";
public int doors = 4;
public int mpg = 25;

public Car(String color, int doors, int mpg) {
this.color = color;
this.doors = doors;
this.mpg = mpg;
}

public void showColor() {
System.out.println(color);
}
}
</pre>

Java 很好地打包了 Car 类的所有属性和方法，因此看见这段代码就知道它要实现什么功能，它义了一个对象的信息。批评混合的构造函数/原型方式的人认为，在构造函数内部找属性， 在其外部方法的做法不合逻辑。因此，他们设计了动态原型方法，以提供更友好的编码风格。
动态原型方法的基本想法与混合的构造函数/原型方式相同，即在构造函数内定义非函数属性，函数属性则利用原型属性定义。唯一的区别是赋予对象方法的位置。下面是用动态原型方法重写的 Car 类：
<pre>function Car(sColor,iDoors,iMpg) {
this.color = sColor;
this.doors = iDoors;
this.mpg = iMpg;
this.drivers = new Array("Mike","John");

if (<code>typeof Car._initialized == "undefined
quot;</code>) {
Car.prototype.showColor = function() {
alert(this.color);
};

<code>Car._initialized = true;</code>

}

}

</pre>

TIY
直到检查 typeof Car._initialized 是否等于 "undefined" 之前，这个构造函数都发生变化。这行代码是动态原型方法中最重要的部分。如果这个值未定义，构造函数将用原型方式继定义对象的方法，然后把 Car._initialized 设置为 true。如果这个值定义了（它的值为 true 时，typeo 的值为 Boolean），那么就不再创建该方法。简而言之，该方法使用标志（_initialized）来判断是已给原型赋予了任何方法。该方法只创建并赋值 一次，传统的 OOP 开发者会高兴地发现，这段代码起来更像其他语言中的类定义了。

<h2>混合工厂方式</h2>
这种方式通常是在不能应用前一种方式时的变通方法。它的目的是创建假构造函数，只返回另种对象的新实例。
这段代码看起来与工厂函数非常相似：
<pre>function Car() {
<code>var oTempCar = new Object;</code>
oTempCar.color = "blue";
oTempCar.doors = 4;
oTempCar.mpg = 25;
oTempCar.showColor = function() {
alert(this.color);
};

return oTempCar;

}

</pre>

TIY
与经典方式不同，这种方式使用 new 运算符，使它看起来像真正的构造函数：
<pre>var car = new Car();</pre>
由于在 Car() 构造函数内部调用了 new 运算符，所以将忽略第二个 new 运算符（位于构造函之外），在构造函数内部创建的对象被传递回变量 car。
这种方式在对象方法的内部管理方面与经典方式有着相同的问题。强烈建议：除非万不得已，是避免使用这种方式。

<h2>采用哪种方式</h2>
如前所述，目前使用最广泛的是混合的构造函数/原型方式。此外，动态原始方法也很流行，在能上与构造函数/原型方式等价。可以采用这两种方式中的任何一种。不过不要单独使用经典的构造数或原型方式，因为这样会给代码引入问题。

<h2>实例</h2>
对象令人感兴趣的一点是用它们解决问题的方式。ECMAScript 中最常见的一个问题是字符串连的性能。与其他语言类似，ECMAScript 的字符串是不可变的，即它们的值不能改变。请考虑下面的

码：
<pre>var str = "hello ";
str += "world";
</pre>
实际上，这段代码在幕后执行的步骤如下：

创建存储 "hello " 的字符串。
创建存储 "world" 的字符串。
创建存储连接结果的字符串。
把 str 的当前内容复制到结果中。
把 "world" 复制到结果中。
更新 str，使它指向结果。

每次完成字符串连接都会执行步骤 2 到 6，使得这种操作非常消耗资源。如果重复这一过程几次，甚至几千次，就会造成性能问题。解决方法是用 Array 对象存储字符串，然后用 join() 方法（参是空字符串）创建最后的字符串。想象用下面的代码代替前面的代码：
<pre>var arr = new Array();
arr[0] = "hello ";
arr[1] = "world";
var str = arr.join("");
</pre>
这样，无论数组中引入多少字符串都不成问题，因为只在调用 join() 方法时才会发生连接操作此时，执行的步骤如下：

创建存储结果的字符串
把每个字符串复制到结果中的合适位置

虽然这种解决方案很好，但还有更好的方法。问题是，这段代码不能确切反映出它的意图。要它更容易理解，可以用 StringBuffer 类打包该功能：
<pre>function StringBuffer () {
this._strings_ = new Array();
}

StringBuffer.prototype.append = function(str) {
this.strings.push(str);
};

StringBuffer.prototype.toString = function() {
return this.strings.join("");
};
</pre>

这段代码首先要注意的是 strings 属性，本意是私有属性。它只有两个方法，即 append() 和 toString() 方法。append() 方法有一个参数，它把该参数附加到字符串数组中，toString() 方法调用数组 join 方法，返回真正连接成的字符串。要用 StringBuffer 对象连接一组字符串，可以用下面的代码
<pre>var buffer = new StringBuffer ();
buffer.append("hello ");
buffer.append("world");
var result = buffer.toString();
</pre>
<a href="http://www.w3school.com.cn/tiy/t.asp?f=jseg_pro_object_defining_example">T

```
Y</a></p>
 <p>可用下面的代码测试 StringBuffer 对象和传统的字符串连接方法的性能：</p>
 <pre>var d1 = new Date();
var str = &quot;&quot;;
for (var i=0; i &lt; 10000; i++) {
    str += &quot;text&quot;;
}
var d2 = new Date();

document.write(&quot;Concatenation with plus: &qu
t;

● (d2.getTime() - d1.getTime()) + &quot; milliseconds
quot;);

var buffer = new StringBuffer();

d1 = new Date();

for (var i=0; i &lt; 10000; i++) {

buffer.append(&quot;text&quot;
;

}

var result = buffer.toString();

d2 = new Date();

document.write(&quot;&lt;br /
gt;Concatenation with StringBuffer: &quot;

● (d2.getTime() - d1.getTime()) + &quot; milliseconds
quot;);

</pre>

 <p><a href="http://www.w3school.com.cn/tiy/t.asp?f=jseg_pro_object_defining_example_pe
formance_test">TIY</a></p>
 <p>这段代码对字符串连接进行两个测试，第一个使用加号，第二个使用 StringBuffer 类。每个操
都连接 10000 个字符串。日期值 d1 和 d2 用于判断完成操作需要的时间。请注意，创建 Date 对象
，如果没有参数，赋予对象的是当前的日期和时间。要计算连接操作历经多少时间，把日期的毫秒表
（用 getTime() 方法的返回值）相减即可。这是衡量 JavaScript 性能的常见方法。该测试的结果可
帮助您比较使用 StringBuffer 类与使用加号的效率差异。</p>
</div>
```