Javascript 创建对象的几种方式？

1. **使用Object构造函数来创建一个对象**
2. **工厂模式**
3. **构造函数模式**
4. **原型模式**
5. **组合使用构造函数模式和原型模式（最常用）**
6. **动态原型模式**
7. **寄生构造函数模式**
8. **稳妥构造函数模式**

**1. 使用Object构造函数来创建一个对象**

下面代码创建了一个person对象，并用两种方式打印出了Name的属性值。

[?](https://www.jb51.net/article/75167.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5 | **var** person = **new** Object();   person.name="kevin";   person.age=31;   alert(person.name);   alert(person["name"]) |

上述写法的另外一种表现形式是使用对象字面量创建一个对象，不要奇怪person[“5”],这里是合法的；另外使用这种加括号的方式字段之间是可以有空格的如person[“my age”].

[?](https://www.jb51.net/article/75167.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | **var** person =    {      name:"Kevin",      age:31,      5:"Test"    };    alert(person.name);    alert(person["5"]); |

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。

**2、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，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。

[?](https://www.jb51.net/article/75167.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | **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"); |

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。

**3. 构造函数模式**

像Object 和Array 这样构造函数，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。

[?](https://www.jb51.net/article/75167.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **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"); |

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。

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。

[?](https://www.jb51.net/article/75167.htm)

|  |  |
| --- | --- |
| 1  2 | 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。

[?](https://www.jb51.net/article/75167.htm)

|  |  |
| --- | --- |
| 1  2  3  4 | 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。

**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。

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。

[?](https://www.jb51.net/article/75167.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **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。

[?](https://www.jb51.net/article/75167.htm)

|  |  |
| --- | --- |
| 1 | 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。

[?](https://www.jb51.net/article/75167.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **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"); |

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。

**4、原型模式**

[?](https://www.jb51.net/article/75167.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **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 |

要理解原型对象，可见我的另一篇博客：JavaScript prototype详解

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。

[?](https://www.jb51.net/article/75167.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | **function** Person(){  }  Person.prototype = {    name : "Nicholas",    age : 29,    job: "Software Engineer",    sayName : **function** () {      alert(**this**.name);    }  }; |

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。

[?](https://www.jb51.net/article/75167.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5 | **var** friend = **new** Person();  alert(friend **instanceof** Object); //true  alert(friend **instanceof** Person); //true  alert(friend.constructor == Person); //false  alert(friend.constructor == Object); //true |

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。

[?](https://www.jb51.net/article/75167.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **function** Person(){  }    Person.prototype = {    constructor : Person,    name : "Nicholas",    age : 29,    job: "Software Engineer",    sayName : **function** () {      alert(**this**.name);    }  }; |

需要注意一点就是：实例中的指针仅指向原型，而不指向构造函数。

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。

[?](https://www.jb51.net/article/75167.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | **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 |

**5、组合使用构造函数模式和原型模式（最常用）**

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。

[?](https://www.jb51.net/article/75167.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | **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 |

**6、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的一个方案，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。

[?](https://www.jb51.net/article/75167.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | **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(); |

**7、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。

[?](https://www.jb51.net/article/75167.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | **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" |

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。

**8、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），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。

[?](https://www.jb51.net/article/75167.htm)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **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;  } |