# 一、原型

## 1. 理解原型对象

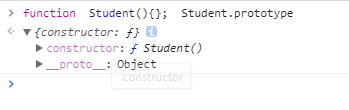
* **prototype**

无论什么时候，只要创建了一个新函数，js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。

* **constructor**

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。也就是说该构造函数指向我们创建的构造函数本身。

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，都是从 Object 继承而来的。



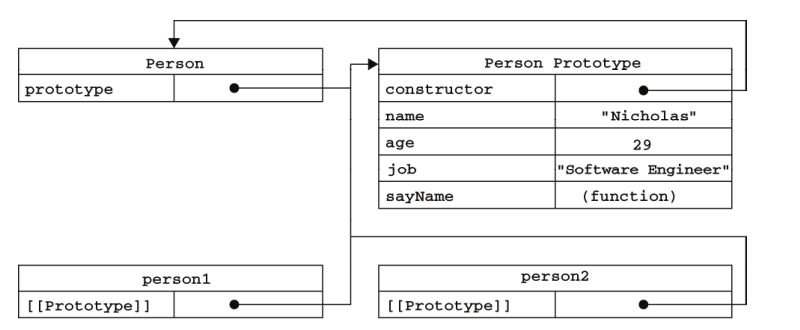
* **\_\_proto\_\_**

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**。

自定义构造函数与实例对象直接的关系是什么？

自定义构造函数的原型对象 == 实例对象的\_\_proto\_\_属性

**关系：**



* **isPrototypeOf()**

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。

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

* **getPrototypeOf()**

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。

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

* **重写属性值：**

每当代码读取某个对象的某个属性时，都会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。如果在原型对象中找到了这个属性，则返回该属性的值。

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。

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。

|  |
| --- |
| 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" —— |

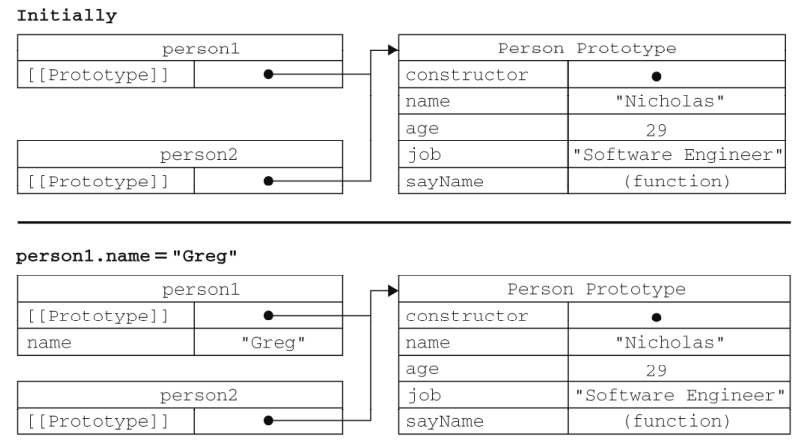
换句话说，添加这个属性只会阻止我们访问原型中的那个属性，但不会修改那个属性。

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。

|  |
| --- |
| 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" —— |

* **hasOwnProperty()**

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 。



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。

## 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。

|  |
| --- |
| 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 |

问题：1.怎么判断一个属性是属于示例对象还是原型对象的？

2.怎么判断一个属性是否是属于原型对象的？

|  |
| --- |
| function hasPrototypeProperty(object, name){  return !object.hasOwnProperty(name) && (name in object);  } |

for-in循环

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。

For(var i in obj) …

* keys()

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。

|  |
| --- |
| 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" |

* Object.getOwnPropertyNames()

Object.getOwnPropertyNames()可以得到所有的实例属性，无论它是否可枚举。

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

## 3. 简单的原型语法

为了不重复添加属性和方法，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);  }  }; |

问题：查看结果，思考原因？

|  |
| --- |
| var friend = new Person();  alert(friend instanceof Object); // true  alert(friend instanceof Person);  alert(friend.constructor == Person);  alert(friend.constructor == Object); |

重写了prototype属性，而constructor属性没定义。

可以重新设定constructor : Person，以这种方式重设 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，可以通过

|  |
| --- |
| // 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

**看以下代码：**

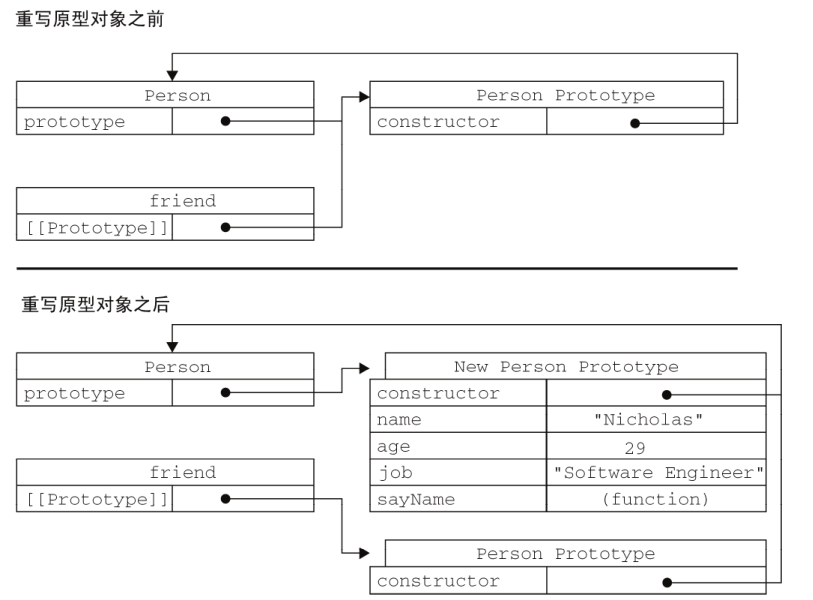
|  |
| --- |
| var friend = new Person();  Person.prototype.sayHi = function(){  alert("hi");  };  friend.sayHi(); |

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。

**看代码：**

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

注意：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。



## 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。

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。

|  |
| --- |
| String.prototype.startsWith = function (text) {  return this.indexOf(text) == 0;  };  var msg = "Hello world!";  alert(msg.startsWith("Hello")); |

注意：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

1.无法传递初始化参数

2.共享的本性导致对于包含引用类型值的属性会出问题

|  |
| --- |
| 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 |

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。

## 7. 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。

|  |
| --- |
| 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); / |

好处：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。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。

## 8. 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。

|  |
| --- |
| 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(); |

这段代码只会在初次调用构造函数时才会执行。

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。

还可以使用 instanceof 操作符确定它的类型。

注意：使用动态原型模式时，不能使用对象字面量重写原型。

## 9. ~~寄生构造函数模式(了解)~~

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。

|  |
| --- |
| 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" |

## 10. ~~稳妥构造函数模式（了解）~~

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。

|  |
| --- |
| 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;  } |

变量 friend 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，使得它非常适合安全执行环境。