**Table of contents**

* [基于原型链的继承](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#%E5%9F%BA%E4%BA%8E%E5%8E%9F%E5%9E%8B%E9%93%BE%E7%9A%84%E7%BB%A7%E6%89%BF)
* [在 JavaScript 中使用原型](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#%E5%9C%A8_javascript_%E4%B8%AD%E4%BD%BF%E7%94%A8%E5%8E%9F%E5%9E%8B)
* [使用不同的方法来创建对象和生成原型链](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#%E4%BD%BF%E7%94%A8%E4%B8%8D%E5%90%8C%E7%9A%84%E6%96%B9%E6%B3%95%E6%9D%A5%E5%88%9B%E5%BB%BA%E5%AF%B9%E8%B1%A1%E5%92%8C%E7%94%9F%E6%88%90%E5%8E%9F%E5%9E%8B%E9%93%BE)
* [prototype 和 Object.getPrototypeOf](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#prototype_%E5%92%8C_object.getprototypeof)
* [结论](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#%E7%BB%93%E8%AE%BA)
* [示例](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#%E7%A4%BA%E4%BE%8B)

**继承与原型链**

对于使用过基于类的语言 (如 Java 或 C++) 的开发人员来说，JavaScript 有点令人困惑，因为它是动态的，并且本身不提供一个 class 实现。（在 ES2015/ES6 中引入了 class 关键字，但那只是语法糖，JavaScript 仍然是基于原型的）。

当谈到继承时，JavaScript 只有一种结构：对象。每个实例对象（ object ）都有一个私有属性（称之为 \_\_proto\_\_ ）指向它的构造函数的原型对象（**prototype**）。该原型对象也有一个自己的原型对象( \_\_proto\_\_ ) ，层层向上直到一个对象的原型对象为 null。根据定义，null 没有原型，并作为这个**原型链**中的最后一个环节。

几乎所有 JavaScript 中的对象都是位于原型链顶端的 [Object](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object) 的实例。

尽管这种原型继承通常被认为是 JavaScript 的弱点之一，但是原型继承模型本身实际上比经典模型更强大。例如，在原型模型的基础上构建经典模型相当简单。

[**基于原型链的继承**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#%E5%9F%BA%E4%BA%8E%E5%8E%9F%E5%9E%8B%E9%93%BE%E7%9A%84%E7%BB%A7%E6%89%BF)

[继承属性](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#%E7%BB%A7%E6%89%BF%E5%B1%9E%E6%80%A7)

JavaScript 对象是动态的属性“包”（指其自己的属性）。JavaScript 对象有一个指向一个原型对象的链。当试图访问一个对象的属性时，它不仅仅在该对象上搜寻，还会搜寻该对象的原型，以及该对象的原型的原型，依次层层向上搜索，直到找到一个名字匹配的属性或到达原型链的末尾。

遵循ECMAScript标准，someObject.[[Prototype]] 符号是用于指向 someObject 的原型。从 ECMAScript 6 开始，[[Prototype]] 可以通过 [Object.getPrototypeOf()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/GetPrototypeOf) 和 [Object.setPrototypeOf()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/setPrototypeOf) 访问器来访问。这个等同于 JavaScript 的非标准但许多浏览器实现的属性 \_\_proto\_\_。

但它不应该与构造函数 func 的 prototype 属性相混淆。被构造函数创建的实例对象的 [[Prototype]] 指向 func 的 prototype 属性。**Object.prototype**属性表示 [Object](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object) 的原型对象。

这里演示当尝试访问属性时会发生什么：

// 让我们从一个函数里创建一个对象o，它自身拥有属性a和b的：

let f = function () {

this.a = 1;

this.b = 2;

}

/\* 这么写也一样

function f() {

this.a = 1;

this.b = 2;

}

\*/

let o = new f(); // {a: 1, b: 2}

// 在f函数的原型上定义属性

f.prototype.b = 3;

f.prototype.c = 4;

// 不要在 f 函数的原型上直接定义 f.prototype = {b:3,c:4};这样会直接打破原型链

// o.[[Prototype]] 有属性 b 和 c

// (其实就是 o.\_\_proto\_\_ 或者 o.constructor.prototype)

// o.[[Prototype]].[[Prototype]] 是 Object.prototype.

// 最后o.[[Prototype]].[[Prototype]].[[Prototype]]是null

// 这就是原型链的末尾，即 null，

// 根据定义，null 就是没有 [[Prototype]]。

// 综上，整个原型链如下:

// {a:1, b:2} ---> {b:3, c:4} ---> Object.prototype---> null

console.log(o.a); // 1

// a是o的自身属性吗？是的，该属性的值为 1

console.log(o.b); // 2

// b是o的自身属性吗？是的，该属性的值为 2

// 原型上也有一个'b'属性，但是它不会被访问到。

// 这种情况被称为"属性遮蔽 (property shadowing)"

console.log(o.c); // 4

// c是o的自身属性吗？不是，那看看它的原型上有没有

// c是o.[[Prototype]]的属性吗？是的，该属性的值为 4

console.log(o.d); // undefined

// d 是 o 的自身属性吗？不是，那看看它的原型上有没有

// d 是 o.[[Prototype]] 的属性吗？不是，那看看它的原型上有没有

// o.[[Prototype]].[[Prototype]] 为 null，停止搜索

// 找不到 d 属性，返回 undefined

Copy to Clipboard

代码来源链接：<https://repl.it/@khaled_hossain_code/prototype>

给对象设置属性会创建自有属性。获取和设置属性的唯一限制是内置 [getter 或 setter](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Guide/Working_with_Objects#defining_getters_and_setters) 的属性。

[继承方法](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#%E7%BB%A7%E6%89%BF%E6%96%B9%E6%B3%95)

JavaScript 并没有其他基于类的语言所定义的“方法”。在 JavaScript 里，任何函数都可以添加到对象上作为对象的属性。函数的继承与其他的属性继承没有差别，包括上面的“属性遮蔽”（这种情况相当于其他语言的方法重写）。

当继承的函数被调用时，[this](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/this) 指向的是当前继承的对象，而不是继承的函数所在的原型对象。

var o = {

a: 2,

m: function(){

return this.a + 1;

}

};

console.log(o.m()); // 3

// 当调用 o.m 时，'this' 指向了 o.

var p = Object.create(o);

// p是一个继承自 o 的对象

p.a = 4; // 创建 p 的自身属性 'a'

console.log(p.m()); // 5

// 调用 p.m 时，'this' 指向了 p

// 又因为 p 继承了 o 的 m 函数

// 所以，此时的 'this.a' 即 p.a，就是 p 的自身属性 'a'

Copy to Clipboard

[**在 JavaScript 中使用原型**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#%E5%9C%A8_javascript_%E4%B8%AD%E4%BD%BF%E7%94%A8%E5%8E%9F%E5%9E%8B)

接下去，来仔细分析一下这些应用场景下， JavaScript 在背后做了哪些事情。

正如之前提到的，在 JavaScript 中，函数（function）是允许拥有属性的。所有的函数会有一个特别的属性 —— prototype 。请注意，以下的代码是独立的（出于严谨，假定页面没有其他的JavaScript代码）。为了最佳的学习体验，我们强烈建议阁下打开浏览器的控制台（在Chrome和火狐浏览器中，按Ctrl+Shift+I即可），进入“console”选项卡，然后把如下的JavaScript代码复制粘贴到窗口中，最后通过按下回车键运行代码。

function doSomething(){}

console.log( doSomething.prototype );

// 和声明函数的方式无关，

// JavaScript 中的函数永远有一个默认原型属性。

var doSomething = function(){};

console.log( doSomething.prototype );

Copy to Clipboard

在控制台显示的JavaScript代码块中，我们可以看到doSomething函数的一个默认属性prototype。而这段代码运行之后，控制台应该显示类似如下的结果：

{

constructor: ƒ doSomething(),

\_\_proto\_\_: {

constructor: ƒ Object(),

hasOwnProperty: ƒ hasOwnProperty(),

isPrototypeOf: ƒ isPrototypeOf(),

propertyIsEnumerable: ƒ propertyIsEnumerable(),

toLocaleString: ƒ toLocaleString(),

toString: ƒ toString(),

valueOf: ƒ valueOf()

}

}

Copy to Clipboard

我们可以给doSomething函数的原型对象添加新属性，如下：

function doSomething(){}

doSomething.prototype.foo = "bar";

console.log( doSomething.prototype );

Copy to Clipboard

可以看到运行后的结果如下：

{

foo: "bar",

constructor: ƒ doSomething(),

\_\_proto\_\_: {

constructor: ƒ Object(),

hasOwnProperty: ƒ hasOwnProperty(),

isPrototypeOf: ƒ isPrototypeOf(),

propertyIsEnumerable: ƒ propertyIsEnumerable(),

toLocaleString: ƒ toLocaleString(),

toString: ƒ toString(),

valueOf: ƒ valueOf()

}

}

Copy to Clipboard

现在我们可以通过new操作符来创建基于这个原型对象的doSomething实例。使用new操作符，只需在调用doSomething函数语句之前添加new。这样，便可以获得这个函数的一个实例对象。一些属性就可以添加到该原型对象中。

请尝试运行以下代码：

function doSomething(){}

doSomething.prototype.foo = "bar"; // add a property onto the prototype

var doSomeInstancing = new doSomething();

doSomeInstancing.prop = "some value"; // add a property onto the object

console.log( doSomeInstancing );

Copy to Clipboard

运行的结果类似于以下的语句。

{

prop: "some value",

\_\_proto\_\_: {

foo: "bar",

constructor: ƒ doSomething(),

\_\_proto\_\_: {

constructor: ƒ Object(),

hasOwnProperty: ƒ hasOwnProperty(),

isPrototypeOf: ƒ isPrototypeOf(),

propertyIsEnumerable: ƒ propertyIsEnumerable(),

toLocaleString: ƒ toLocaleString(),

toString: ƒ toString(),

valueOf: ƒ valueOf()

}

}

}

Copy to Clipboard

如上所示, doSomeInstancing 中的\_\_proto\_\_是 doSomething.prototype. 但这是做什么的呢？当你访问doSomeInstancing 中的一个属性，浏览器首先会查看doSomeInstancing 中是否存在这个属性。

如果 doSomeInstancing 不包含属性信息, 那么浏览器会在 doSomeInstancing 的 \_\_proto\_\_ 中进行查找(同 doSomething.prototype). 如属性在 doSomeInstancing 的 \_\_proto\_\_ 中查找到，则使用 doSomeInstancing 中 \_\_proto\_\_ 的属性。

否则，如果 doSomeInstancing 中 \_\_proto\_\_ 不具有该属性，则检查doSomeInstancing 的 \_\_proto\_\_ 的  \_\_proto\_\_ 是否具有该属性。默认情况下，任何函数的原型属性 \_\_proto\_\_ 都是 window.Object.prototype. 因此, 通过doSomeInstancing 的 \_\_proto\_\_ 的  \_\_proto\_\_  ( 同 doSomething.prototype 的 \_\_proto\_\_ (同  Object.prototype)) 来查找要搜索的属性。

如果属性不存在 doSomeInstancing 的 \_\_proto\_\_ 的  \_\_proto\_\_ 中， 那么就会在doSomeInstancing 的 \_\_proto\_\_ 的  \_\_proto\_\_ 的  \_\_proto\_\_ 中查找。然而, 这里存在个问题：doSomeInstancing 的 \_\_proto\_\_ 的  \_\_proto\_\_ 的  \_\_proto\_\_ 其实不存在。因此，只有这样，在 \_\_proto\_\_ 的整个原型链被查看之后，这里没有更多的 \_\_proto\_\_ ， 浏览器断言该属性不存在，并给出属性值为 undefined 的结论。

让我们在控制台窗口中输入更多的代码，如下：

function doSomething(){}

doSomething.prototype.foo = "bar";

var doSomeInstancing = new doSomething();

doSomeInstancing.prop = "some value";

console.log("doSomeInstancing.prop: " + doSomeInstancing.prop);

console.log("doSomeInstancing.foo: " + doSomeInstancing.foo);

console.log("doSomething.prop: " + doSomething.prop);

console.log("doSomething.foo: " + doSomething.foo);

console.log("doSomething.prototype.prop: " + doSomething.prototype.prop);

console.log("doSomething.prototype.foo: " + doSomething.prototype.foo);

Copy to Clipboard

结果如下：

doSomeInstancing.prop: some value

doSomeInstancing.foo: bar

doSomething.prop: undefined

doSomething.foo: undefined

doSomething.prototype.prop: undefined

doSomething.prototype.foo: bar

Copy to Clipboard

[**使用不同的方法来创建对象和生成原型链**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#%E4%BD%BF%E7%94%A8%E4%B8%8D%E5%90%8C%E7%9A%84%E6%96%B9%E6%B3%95%E6%9D%A5%E5%88%9B%E5%BB%BA%E5%AF%B9%E8%B1%A1%E5%92%8C%E7%94%9F%E6%88%90%E5%8E%9F%E5%9E%8B%E9%93%BE)

[使用语法结构创建的对象](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#%E4%BD%BF%E7%94%A8%E8%AF%AD%E6%B3%95%E7%BB%93%E6%9E%84%E5%88%9B%E5%BB%BA%E7%9A%84%E5%AF%B9%E8%B1%A1)

var o = {a: 1};

// o 这个对象继承了 Object.prototype 上面的所有属性

// o 自身没有名为 hasOwnProperty 的属性

// hasOwnProperty 是 Object.prototype 的属性

// 因此 o 继承了 Object.prototype 的 hasOwnProperty

// Object.prototype 的原型为 null

// 原型链如下:

// o ---> Object.prototype ---> null

var a = ["yo", "whadup", "?"];

// 数组都继承于 Array.prototype

// (Array.prototype 中包含 indexOf, forEach 等方法)

// 原型链如下:

// a ---> Array.prototype ---> Object.prototype ---> null

function f(){

return 2;

}

// 函数都继承于 Function.prototype

// (Function.prototype 中包含 call, bind等方法)

// 原型链如下:

// f ---> Function.prototype ---> Object.prototype ---> null

Copy to Clipboard

[使用构造器创建的对象](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#%E4%BD%BF%E7%94%A8%E6%9E%84%E9%80%A0%E5%99%A8%E5%88%9B%E5%BB%BA%E7%9A%84%E5%AF%B9%E8%B1%A1)

在 JavaScript 中，构造器其实就是一个普通的函数。当使用 [new 操作符](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/new) 来作用这个函数时，它就可以被称为构造方法（构造函数）。

function Graph() {

this.vertices = [];

this.edges = [];

}

Graph.prototype = {

addVertex: function(v){

this.vertices.push(v);

}

};

var g = new Graph();

// g 是生成的对象，他的自身属性有 'vertices' 和 'edges'。

// 在 g 被实例化时，g.[[Prototype]] 指向了 Graph.prototype。

Copy to Clipboard

[使用 Object.create 创建的对象](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#%E4%BD%BF%E7%94%A8_object.create_%E5%88%9B%E5%BB%BA%E7%9A%84%E5%AF%B9%E8%B1%A1)

ECMAScript 5 中引入了一个新方法：[Object.create()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/create)。可以调用这个方法来创建一个新对象。新对象的原型就是调用 create 方法时传入的第一个参数：

var a = {a: 1};

// a ---> Object.prototype ---> null

var b = Object.create(a);

// b ---> a ---> Object.prototype ---> null

console.log(b.a); // 1 (继承而来)

var c = Object.create(b);

// c ---> b ---> a ---> Object.prototype ---> null

var d = Object.create(null);

// d ---> null

console.log(d.hasOwnProperty); // undefined, 因为d没有继承Object.prototype

Copy to Clipboard

[使用 class 关键字创建的对象](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#%E4%BD%BF%E7%94%A8_class_%E5%85%B3%E9%94%AE%E5%AD%97%E5%88%9B%E5%BB%BA%E7%9A%84%E5%AF%B9%E8%B1%A1)

ECMAScript6 引入了一套新的关键字用来实现 [class](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Classes)。使用基于类语言的开发人员会对这些结构感到熟悉，但它们是不同的。JavaScript 仍然基于原型。这些新的关键字包括 [class](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Statements/class), [constructor](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Classes/constructor)，[static](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Classes/static)，[extends](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Classes/extends) 和 [super](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/super)。

"use strict";

class Polygon {

constructor(height, width) {

this.height = height;

this.width = width;

}

}

class Square extends Polygon {

constructor(sideLength) {

super(sideLength, sideLength);

}

get area() {

return this.height \* this.width;

}

set sideLength(newLength) {

this.height = newLength;

this.width = newLength;

}

}

var square = new Square(2);

Copy to Clipboard

[性能](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#%E6%80%A7%E8%83%BD)

在原型链上查找属性比较耗时，对性能有副作用，这在性能要求苛刻的情况下很重要。另外，试图访问不存在的属性时会遍历整个原型链。

遍历对象的属性时，原型链上的**每个**可枚举属性都会被枚举出来。要检查对象是否具有自己定义的属性，而不是其原型链上的某个属性，则必须使用所有对象从 Object.prototype 继承的 [hasOwnProperty (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty) 方法。下面给出一个具体的例子来说明它：

console.log(g.hasOwnProperty('vertices'));

// true

console.log(g.hasOwnProperty('nope'));

// false

console.log(g.hasOwnProperty('addVertex'));

// false

console.log(g.\_\_proto\_\_.hasOwnProperty('addVertex'));

// true

Copy to Clipboard

[hasOwnProperty (en-US)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/hasOwnProperty) 是 JavaScript 中唯一一个处理属性并且**不会**遍历原型链的方法。（译者注：原文如此。另一种这样的方法：[Object.keys()](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Object/keys)）

注意：检查属性是否为 [undefined](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/undefined) 是**不能够**检查其是否存在的。该属性可能已存在，但其值恰好被设置成了 undefined。

[错误实践：扩展原生对象的原型](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#%E9%94%99%E8%AF%AF%E5%AE%9E%E8%B7%B5%EF%BC%9A%E6%89%A9%E5%B1%95%E5%8E%9F%E7%94%9F%E5%AF%B9%E8%B1%A1%E7%9A%84%E5%8E%9F%E5%9E%8B)

经常使用的一个错误实践是扩展 Object.prototype 或其他内置原型。

这种技术被称为猴子补丁并且会破坏封装。尽管一些流行的框架（如 Prototype.js）在使用该技术，但仍然没有足够好的理由使用附加的非标准方法来混入内置原型。

扩展内置原型的**唯一**理由是支持 JavaScript 引擎的新特性，如 Array.forEach。

[总结：4 个用于拓展原型链的方法](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#%E6%80%BB%E7%BB%93%EF%BC%9A4_%E4%B8%AA%E7%94%A8%E4%BA%8E%E6%8B%93%E5%B1%95%E5%8E%9F%E5%9E%8B%E9%93%BE%E7%9A%84%E6%96%B9%E6%B3%95)

下面列举四种用于拓展原型链的方法，以及他们的优势和缺陷。下列四个例子都创建了完全相同的 inst 对象（所以在控制台上的输出也是一致的），为了举例，唯一的区别是他们的创建方法不同。

|  |  |  |  |
| --- | --- | --- | --- |
| 名称 | 例子 | 优势 | 缺陷 |
| New-initialization | function foo(){}  foo.prototype = {  foo\_prop: "foo val"  };  function bar(){}  var proto = new foo;  proto.bar\_prop = "bar val";  bar.prototype = proto;  var inst = new bar;  console.log(inst.foo\_prop);  console.log(inst.bar\_prop);  Copy to Clipboard | 支持目前以及所有可想象到的浏览器(IE5.5都可以使用)。 这种方法非常快，非常符合标准，并且充分利用JIT优化。 | 为使用此方法，必须对相关函数初始化。 在初始化过程中，构造函数可以存储每个对象必须生成的唯一信息。但是，这种唯一信息只生成一次，可能会带来潜在的问题。此外，构造函数的初始化，可能会将不需要的方法放在对象上。然而，如果你只在自己的代码中使用，你也清楚（或有通过注释等写明）各段代码在做什么，这些在大体上都不是问题（事实上，通常是有益处的）。 |
| Object.create | function foo(){}  foo.prototype = {  foo\_prop: "foo val"  };  function bar(){}  var proto = Object.create(  foo.prototype  );  proto.bar\_prop = "bar val";  bar.prototype = proto;  var inst = new bar;  console.log(inst.foo\_prop);  console.log(inst.bar\_prop);  Copy to Clipboard  function foo(){}  foo.prototype = {  foo\_prop: "foo val"  };  function bar(){}  var proto = Object.create(  foo.prototype,  {  bar\_prop: {  value: "bar val"  }  }  );  bar.prototype = proto;  var inst = new bar;  console.log(inst.foo\_prop);  console.log(inst.bar\_prop)  Copy to Clipboard | 支持当前所有非微软版本或者 IE9 以上版本的浏览器。允许一次性地直接设置 \_\_proto\_\_ 属性，以便浏览器能更好地优化对象。同时允许通过 Object.create(null) 来创建一个没有原型的对象。 | 不支持 IE8 以下的版本。然而，随着微软不再对系统中运行的旧版本浏览器提供支持，这将不是在大多数应用中的主要问题。 另外，这个慢对象初始化在使用第二个参数的时候有可能成为一个性能黑洞，因为每个对象的描述符属性都有自己的描述对象。当以对象的格式处理成百上千的对象描述的时候，可能会造成严重的性能问题。 |
| Object.setPrototypeOf | function foo(){}  foo.prototype = {  foo\_prop: "foo val"  };  function bar(){}  var proto = {  bar\_prop: "bar val"  };  Object.setPrototypeOf(  proto, foo.prototype  );  bar.prototype = proto;  var inst = new bar;  console.log(inst.foo\_prop);  console.log(inst.bar\_prop);  Copy to Clipboard  function foo(){}  foo.prototype = {  foo\_prop: "foo val"  };  function bar(){}  var proto;  proto=Object.setPrototypeOf(  { bar\_prop: "bar val" },  foo.prototype  );  bar.prototype = proto;  var inst = new bar;  console.log(inst.foo\_prop);  console.log(inst.bar\_prop)  Copy to Clipboard | 支持所有现代浏览器和微软IE9+浏览器。允许动态操作对象的原型，甚至能强制给通过 Object.create(null) 创建出来的没有原型的对象添加一个原型。 | 这个方式表现并不好，应该被弃用。如果你在生产环境中使用这个方法，那么快速运行 Javascript 就是不可能的，因为许多浏览器优化了原型，尝试在调用实例之前猜测方法在内存中的位置，但是动态设置原型干扰了所有的优化，甚至可能使浏览器为了运行成功，使用完全未经优化的代码进行重编译。 不支持 IE8 及以下的浏览器版本。 |
| \_\_proto\_\_ | function foo(){}  foo.prototype = {  foo\_prop: "foo val"  };  function bar(){}  var proto = {  bar\_prop: "bar val",  \_\_proto\_\_: foo.prototype  };  bar.prototype = proto;  var inst = new bar;  console.log(inst.foo\_prop);  console.log(inst.bar\_prop);  Copy to Clipboard  var inst = {  \_\_proto\_\_: {  bar\_prop: "bar val",  \_\_proto\_\_: {  foo\_prop: "foo val",  \_\_proto\_\_: Object.prototype  }  }  };  console.log(inst.foo\_prop);  console.log(inst.bar\_prop)  Copy to Clipboard | 支持所有现代非微软版本以及 IE11 以上版本的浏览器。将 \_\_proto\_\_ 设置为非对象的值会静默失败，并不会抛出错误。 | 应该完全将其抛弃因为这个行为完全不具备性能可言。 如果你在生产环境中使用这个方法，那么快速运行 Javascript 就是不可能的，因为许多浏览器优化了原型，尝试在调用实例之前猜测方法在内存中的位置，但是动态设置原型干扰了所有的优化，甚至可能使浏览器为了运行成功，使用完全未经优化的代码进行重编译。不支持 IE10 及以下的浏览器版本。 |

[**prototype 和 Object.getPrototypeOf**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#prototype_%E5%92%8C_object.getprototypeof)

对于从 Java 或 C++ 转过来的开发人员来说，JavaScript 会有点让人困惑，因为它完全是动态的，都是运行时，而且不存在类（class）。所有的都是实例（对象）。即使我们模拟出的 “类”，也只是一个函数对象。

你可能已经注意到我们的 function A 有一个叫做 prototype 的特殊属性。该特殊属性可与 JavaScript 的 new 操作符一起使用。对原型对象的引用被复制到新实例的内部 [[Prototype]] 属性。例如，当执行 var a1 = new A(); 时，JavaScript（在内存中创建对象之后，和在运行函数 A() 把 this 指向对象之前）设置 a1.[[Prototype]] = A.prototype;。然后当您访问实例的属性时，JavaScript 首先会检查它们是否直接存在于该对象上，如果不存在，则会 [[Prototype]] 中查找。这意味着你在 prototype 中定义的所有内容都可以由所有实例有效地共享，你甚至可以稍后更改部分 prototype，并在所有现有实例中显示更改（如果有必要的话）。

像上面的例子中，如果你执行 var a1 = new A(); var a2 = new A(); 那么 a1.doSomething 事实上会指向 Object.getPrototypeOf(a1).doSomething，它就是你在 A.prototype.doSomething 中定义的内容。也就是说：Object.getPrototypeOf(a1).doSomething == Object.getPrototypeOf(a2).doSomething == A.prototype.doSomething（补充：实际上，执行 a1.doSomething() 相当于执行 Object.getPrototypeOf(a1).doSomething.call(a1)==A.prototype.doSomething.call(a1)）

简而言之， prototype 是用于类的，而 Object.getPrototypeOf() 是用于实例的（instances），两者功能一致。

[[Prototype]] 看起来就像**递归**引用， 如 a1.doSomething、Object.getPrototypeOf(a1).doSomething、Object.getPrototypeOf(Object.getPrototypeOf(a1)).doSomething 等等等， 直到它被找到或 Object.getPrototypeOf 返回 null。

因此，当你执行：

var o = new Foo();

Copy to Clipboard

JavaScript 实际上执行的是：

var o = new Object();

o.\_\_proto\_\_ = Foo.prototype;

Foo.call(o);

Copy to Clipboard

（或者类似上面这样的），然后，当你执行：

o.someProp;

Copy to Clipboard

它检查 o 是否具有 someProp 属性。如果没有，它会查找 Object.getPrototypeOf(o).someProp，如果仍旧没有，它会继续查找 Object.getPrototypeOf(Object.getPrototypeOf(o)).someProp。

[**结论**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#%E7%BB%93%E8%AE%BA)

在使用原型继承编写复杂代码之前，理解原型继承模型是**至关重要**的。此外，请注意代码中原型链的长度，并在必要时将其分解，以避免可能的性能问题。此外，原生原型**不应该**被扩展，除非它是为了与新的 JavaScript 特性兼容。

**译者注：**在英文原版中，以下内容已被移除。保留仅作参考。

[**示例**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain#%E7%A4%BA%E4%BE%8B)

B 继承自 A：

function A(a){

this.varA = a;

}

// 以上函数 A 的定义中，既然 A.prototype.varA 总是会被 this.varA 遮蔽，

// 那么将 varA 加入到原型（prototype）中的目的是什么？

A.prototype = {

varA : null,

/\*

既然它没有任何作用，干嘛不将 varA 从原型（prototype）去掉 ?

也许作为一种在隐藏类中优化分配空间的考虑 ?

https://developers.google.com/speed/articles/optimizing-javascript

如果varA并不是在每个实例中都被初始化，那这样做将是有效果的。

\*/

doSomething : function(){

// ...

}

}

function B(a, b){

A.call(this, a);

this.varB = b;

}

B.prototype = Object.create(A.prototype, {

varB : {

value: null,

enumerable: true,

configurable: true,

writable: true

},

doSomething : {

value: function(){ // override

A.prototype.doSomething.apply(this, arguments);

// call super

// ...

},

enumerable: true,

configurable: true,

writable: true

}

});

B.prototype.constructor = B;

var b = new B();

b.doSomething();

Copy to Clipboard

最重要的部分是：

* 类型被定义在 .prototype 中
* 用 Object.create() 来继承

Found a problem with this page?

* [Source on **GitHub**](https://github.com/mdn/translated-content/blob/main/files/zh-cn/web/javascript/inheritance_and_the_prototype_chain/index.html)
* [Report a problem with this content on **GitHub**](https://github.com/mdn/content/issues/new?body=MDN+URL%3A+https%3A%2F%2Fdeveloper.mozilla.org%2Fzh-CN%2Fdocs%2FWeb%2FJavaScript%2FInheritance_and_the_prototype_chain%0A%0A%23%23%23%23+What+information+was+incorrect%2C+unhelpful%2C+or+incomplete%3F%0A%0A%0A%23%23%23%23+Specific+section+or+headline%3F%0A%0A%0A%23%23%23%23+What+did+you+expect+to+see%3F%0A%0A%0A%23%23%23%23+Did+you+test+this%3F+If+so%2C+how%3F%0A%0A%0A%3C%21--+Do+not+make+changes+below+this+line+--%3E%0A%3Cdetails%3E%0A%3Csummary%3EMDN+Content+page+report+details%3C%2Fsummary%3E%0A%0A*+Folder%3A+%60zh-cn%2Fweb%2Fjavascript%2Finheritance_and_the_prototype_chain%60%0A*+MDN+URL%3A+https%3A%2F%2Fdeveloper.mozilla.org%2Fzh-CN%2Fdocs%2FWeb%2FJavaScript%2FInheritance_and_the_prototype_chain%0A*+GitHub+URL%3A+https%3A%2F%2Fgithub.com%2Fmdn%2Ftranslated-content%2Fblob%2Fmain%2Ffiles%2Fzh-cn%2Fweb%2Fjavascript%2Finheritance_and_the_prototype_chain%2Findex.html%0A*+Last+commit%3A+https%3A%2F%2Fgithub.com%2Fmdn%2Ftranslated-content%2Fcommit%2F52ccdf1f0a1cee3f0659ac78e81c79dae152399b%0A*+Document+last+modified%3A+2021-06-23T00%3A34%3A06.000Z%0A%0A%3C%2Fdetails%3E&title=Issue+with+%22%E7%BB%A7%E6%89%BF%E4%B8%8E%E5%8E%9F%E5%9E%8B%E9%93%BE%22%3A+%28short+summary+here+please%29&labels=Content%3AJavaScript%2Cneeds-triage)
* Want to fix the problem yourself? See [our Contribution guide](https://github.com/mdn/content/blob/main/README.md).

1. 1. [继承和原型链](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Inheritance_and_the_prototype_chain)
   2. [严格模式](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Strict_mode)
   3. [JavaScript 类型化数组](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Typed_arrays)
   4. [内存管理](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Memory_Management)
   5. [Concurrency model and Event Loop](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/EventLoop)