

此页面由社区从英文翻译而来。了解更多并加入 MDN Web Docs 社区。

## 继承与原型链

对于使用过基于类的语言（如 Java 或 C++）的开发者来说，JavaScript 实在是有些令人困惑——JavaScript 是动态的且没有静态类型。

当谈到继承时，JavaScript 只有一种结构：对象。每个对象（object）都有一个私有属性指向另一个名为**原型**（prototype）的对象。原型对象也有一个自己的原型，层层向上直到一个对象的原型为 `null`。根据定义，`null` 没有原型，并作为这个**原型链**（prototype chain）中的最后一个环节。可以改变原型链中的任何成员，甚至可以在运行时换出原型，因此 JavaScript 中不存在**静态分派**的概念。

尽管这种混杂通常被认为是 JavaScript 的弱点之一，但是原型继承模型本身实际上比类式模型更强大。例如，在原型模型的基础上构建类式模型（即类的实现方式）相当简单。

尽管类现在被广泛采用并成为 JavaScript 中新的范式，但类并没有带来新的继承模式。虽然类为大部分原型的机制提供了抽象，但了解原型在底层是如何工作的仍然十分有用。

## 基于原型链的继承

### 继承属性

JavaScript 对象是动态的属性（指其自有属性）“包”。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。

**备注：** 遵循 ECMAScript 标准，符号 `someObject.[[Prototype]]` 用于标识 `someObject` 的原型。内部插槽 `[[Prototype]]` 可以通过 `Object.getPrototypeOf()` 和

`Object.setPrototypeOf()` 函数来访问。这个等同于 JavaScript 的非标准但被许多 JavaScript 引擎实现的属性 `__proto__` 访问器。为在保持简洁的同时避免混淆，在我们的符号中会避免使用 `obj.__proto__`，而是使用 `obj.[[Prototype]]` 作为代替。其对应于 `Object.getPrototypeOf(obj)`。

它不应与函数的 `func.prototype` 属性混淆，后者指定在给定函数被用作构造函数时分配给所有对象实例的 `[ [ Prototype ] ]`。我们将在后面的小节中讨论构造函数的原型属性。

有几种可以指定对象的 `[ [ Prototype ] ]` 的方法，这些方法将在后面的小节中列出。现在，我们将使用 `__proto__` 语法进行说明。值得注意的是，`{ __proto__: ... }` 语法与 `obj.__proto__` 访问器不同：前者是标准且未被弃用的。

在像 `{ a: 1, b: 2, __proto__: c }` 这样的对象字面量中，`c` 值（必须为 `null` 或另一个对象）将变成字面量所表示的对象的 `[ [ Prototype ] ]`，而其他键（如 `a` 和 `b`）将变成对象的自有属性。这种语法读起来非常自然，因为 `[ [ Prototype ] ]` 只是对象的“内部属性”。

下面演示当尝试访问属性时会发生什么：

JS

```
const o = {
  a: 1,
  b: 2,
  // __proto__ 设置了 [ [ Prototype ] ]。它在这里被指定为另一个对象字面量。
  __proto__: {
    b: 3,
    c: 4,
  },
};

// o.[ [ Prototype ] ] 具有属性 b 和 c。
// o.[ [ Prototype ] ].[ [ Prototype ] ] 是 Object.prototype（我们会在下文解释其含义）。
// 最后，o.[ [ Prototype ] ].[ [ Prototype ] ].[ [ Prototype ] ] 是 null。
// 这是原型链的末尾，值为 null。
// 根据定义，其没有 [ [ Prototype ] ]。
// 因此，完整的原型链看起来像这样：
// { a: 1, b: 2 } ---> { b: 3, c: 4 } ---> Object.prototype ---> null

console.log(o.a); // 1
// o 上有自有属性“a”吗？有，且其值为 1。

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

```
// o 上有自有属性“b”吗？有，且其值为 2。  
// 原型也有“b”属性，但其没有被访问。  
// 这被称为属性遮蔽（Property Shadowing）
```

```
console.log(o.c); // 4  
// o 上有自有属性“c”吗？没有，检查其原型。  
// o.[[Prototype]] 上有自有属性“c”吗？有，其值为 4。  
  
console.log(o.d); // undefined  
// o 上有自有属性“d”吗？没有，检查其原型。  
// o.[[Prototype]] 上有自有属性“d”吗？没有，检查其原型。  
// o.[[Prototype]].[[Prototype]] 是 Object.prototype 且  
// 其默认没有“d”属性，检查其原型。  
// o.[[Prototype]].[[Prototype]].[[Prototype]] 为 null，停止搜索，  
// 未找到该属性，返回 undefined。
```

给对象设置属性会创建自有属性。获取和设置行为规则的唯一例外是当它被 [getter 或 setter](#) 拦截时。

同理，你可以创建更长的原型链，并在原型链上查找一个属性。

JS

```
const o = {  
  a: 1,  
  b: 2,  
  // __proto__ 设置了 [[Prototype]]。它在这里被指定为另一个对象字面量。  
  __proto__: {  
    b: 3,  
    c: 4,  
    __proto__: {  
      d: 5,  
    },  
  },  
};  
  
// { a: 1, b: 2 } ---> { b: 3, c: 4 } ---> { d: 5 } ---> Object.prototype ---> null
```

 mdn web docs

继承“方法”

JavaScript 并没有其他基于类的语言所定义的“[方法](#)”。在 JavaScript 中，任何函数都可以添加到对象上作为其属性。函数的继承与其他属性的继承没有差别，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` 值指向的是当前继承的对象，而不是拥有该函数属性的原型对象。

JS

```
const parent = {
  value: 2,
  method() {
    return this.value + 1;
  },
};

console.log(parent.method()); // 3
// 当调用 parent.method 时，“this”指向了 parent

// child 是一个继承了 parent 的对象
const child = {
  __proto__: parent,
};

console.log(child.method()); // 3
// 调用 child.method 时，“this”指向了 child。
// 又因为 child 继承的是 parent 的方法，
// 首先在 child 上寻找“value”属性。但由于 child 本身
// 没有名为“value”的自有属性，该属性会在
// [[Prototype]] 上被找到，即 parent.value。

child.value = 4; // 在 child，将“value”属性赋值为 4。
// 这会遮蔽 parent 上的“value”属性。
// child 对象现在看起来是这样的：
// { value: 4, __proto__: { value: 2, method: [Function] } }
console.log(child.method()); // 5
// 因为 child 现在拥有“value”属性，“this.value”现在表示
// child.value
```

## 构造函数

原型的强大之处在于，如果一组属性应该出现在每一个实例上，那我们就可以重用它们——尤其是对于方法。假设我们要创建多个盒子，其中每一个盒子都是一个对象，包含一个可以通过`getValue` 函数访问的值。一个简单的实现可能是：

JS

```
const boxes = [
  { value: 1, getValue() { return this.value; } },
  { value: 2, getValue() { return this.value; } },
  { value: 3, getValue() { return this.value; } },
];
```

这是不够好的，因为每一个实例都有自己的，做相同事情的函数属性，这是冗余且不必要的。相反，我们可以将 `getValue` 移动到所有盒子的 `[[Prototype]]` 上：

JS

```
const boxPrototype = {
  getValue() {
    return this.value;
  },
};

const boxes = [
  { value: 1, __proto__: boxPrototype },
  { value: 2, __proto__: boxPrototype },
  { value: 3, __proto__: boxPrototype },
];
```

这样，所有盒子的 `getValue` 方法都会引用相同的函数，降低了内存使用率。但是，手动绑定每个对象创建的 `__proto__` 仍旧非常不方便。这时，我们就可以使用构造函数，它会自动为每个构造的对象设置 `[[Prototype]]`。构造函数是使用 [new](#) 调用的函数。

JS

```
// 一个构造函数
function Box(value) {
  this.value = value;
}

// 使用 Box() 构造函数创建的所有盒子都将具有的属性
Box.prototype.getValue = function () {
  return this.value;
};

const boxes = [new Box(1), new Box(2), new Box(3)];
```

我们说 `new Box(1)` 是通过 `Box` 构造函数创建的一个实例。`Box.prototype` 与我们之前创建的 `boxPrototype` 并无太大区别——它只是一个普通的对象。通过构造函数创建的每一个实例都会自动将构造函数的 `prototype` 属性作为其 `[[Prototype]]`。即，`Object.getPrototypeOf(new Box()) === Box.prototype`。`Constructor.prototype` 默认具有一个自有属性：`constructor`，它引用了构造函数本身。即，`Box.prototype.constructor === Box`。这允许我们在任何实例中访问原始构造函数。

**备注：**如果构造函数返回非原始值，则该值将成为 `new` 表达式的结果。在这种情况下，`[[Prototype]]` 可能无法正确绑定——但在实践中应该很少发生。

上面的构造函数可以重写为类：

JS

```
class Box {
  constructor(value) {
    this.value = value;
  }

  // 在 Box.prototype 上创建方法
  getValue() {
    return this.value;
  }
}
```

类是构造函数的语法糖，这意味着你仍然可以修改 `Box.prototype` 来改变所有实例的行为。然而，由于类被设计为对底层原型机制的抽象，我们将在本教程中使用更轻量级的构造函数语法，以充分展示原型的工作原理。

因为 `Box.prototype` 引用了（作为所有实例的 `[[Prototype]]` 的）相同的对象，所以我们可以通  
过改变 `Box.prototype` 来改变所有实例的行为。

JS

```
function Box(value) {
  this.value = value;
}

Box.prototype.getValue = function () {
  return this.value;
};
```

```
const box = new Box(1);

// 在创建实例后修改 Box.prototype
Box.prototype.getValue = function () {
    return this.value + 1;
};

box.getValue(); // 2
```

有个推论是：重新赋值 `Constructor.prototype` (`Constructor.prototype = ...`) 是一个不好的主意，原因有两点：

- 在重新赋值之前创建的实例的 `[[Prototype]]` 现在引用的是与重新赋值之后创建的实例的 `[[Prototype]]` 不同的对象——改变一个的 `[[Prototype]]` 不再改变另一个的 `[[Prototype]]`。
- 除非你手动重新设置 `constructor` 属性，否则无法再通过 `instance.constructor` 追踪到构造函数，这可能会破坏用户期望的行为。一些内置操作也会读取 `constructor` 属性，如果没有设置，它们可能无法按预期工作。

`Constructor.prototype` 仅在构造实例时有用。它与 `Constructor.[[Prototype]]` 无关，后者是构造函数的自有原型，即 `Function.prototype`。也就是说，`Object.getPrototypeOf(Constructor) === Function.prototype`。

## 字面量的隐式构造函数

JavaScript 中的一些字面量语法会创建隐式设置 `[[Prototype]]` 的实例。例如：

JS

```
// 对象字面量（没有 `__proto__` 键）自动将
// `Object.prototype` 作为它们的 `[[Prototype]]`
const object = { a: 1 };
Object.getPrototypeOf(object) === Object.prototype; // true

// 数组字面量自动将 `Array.prototype` 作为它们的 `[[Prototype]]`
const array = [1, 2, 3];
Object.getPrototypeOf(array) === Array.prototype; // true

// 正则表达式字面量自动将 `RegExp.prototype` 作为它们的 `[[Prototype]]`
const regexp = /abc/;
Object.getPrototypeOf(regexp) === RegExp.prototype; // true
```

我们可以将它们“解糖（de-sugar）”为构造函数形式。

JS

```
const array = new Array(1, 2, 3);
const regexp = new RegExp("abc");
```

例如，像 `map()` 这样的“数组方法”只是在 `Array.prototype` 上定义的方法，而它们又自动在所有数组实例上可用，就是因为这个原因。

**警告：**有一个常见的错误实践（misfeature）：扩展 `Object.prototype` 或其它内置原型。这种不良特性例子是，定义 `Array.prototype.myMethod = function () {...}`，然后在所有数组实例上使用 `myMethod`。

这种错误实践被称为猴子修补（monkey patching）。使用猴子修补存在向前兼容的风险，因为如果语言在未来添加了此方法但具有不同的签名，你的代码将会出错。它已经导致了类似于 `SmooshGate` 这样的事件，并且由于 JavaScript 致力于“不破坏 web”，因此这可能会对语言的发展造成极大的麻烦。

扩展内置原型的唯一理由是向后移植新的 JavaScript 引擎的特性，比如 `Array.prototype.forEach`。

有趣的是，由于历史原因，一些内置构造函数的 `prototype` 属性本身就是其自身的实例。例如，`Number.prototype` 是数字 0，`Array.prototype` 是一个空数组，`RegExp.prototype` 是 `/(?:)/`。

JS

```
Number.prototype + 1; // 1
Array.prototype.map((x) => x + 1); // []
String.prototype + "a"; // "a"
RegExp.prototype.source; // "(?:)"
Function.prototype(); // Function.prototype 本身就是一个无操作函数
```

然而，对于用户定义的构造函数，以及 `Map` 等现代的构造函数，则并非如此。

JS

```
Map.prototype.get(1);
// Uncaught TypeError: get method called on incompatible Map.prototype
```

构建更长的继承链

`Constructor.prototype` 属性将成为构造函数实例的 `[[Prototype]]`，包括 `Constructor.prototype` 自身的 `[[Prototype]]`。默认情况下，`Constructor.prototype` 是一个普通对象——即 `Object.getPrototypeOf(Constructor.prototype) === Object.prototype`。唯一的例外是 `Object.prototype` 本身，其 `[[Prototype]]` 是 `null`——即 `Object.getPrototypeOf(Object.prototype) === null`。因此，一个典型的构造函数将构建以下原型链：

JS

```
function Constructor() {}

const obj = new Constructor();
// obj ---> Constructor.prototype ---> Object.prototype ---> null
```

要构建更长的原型链，我们可用通过 [`Object.setPrototypeOf\(\)`](#) 函数设置 `Constructor.prototype` 的 `[[Prototype]]`。

JS

```
function Base() {}
function Derived() {}

// 将 `Derived.prototype` 的 `[[Prototype]]`
// 设置为 `Base.prototype`
Object.setPrototypeOf(Derived.prototype, Base.prototype);

const obj = new Derived();
// obj ---> Derived.prototype ---> Base.prototype ---> Object.prototype ---> null
```

在类的术语中，这等同于使用 [`extends`](#) 语法。

JS

```
class Base {}
class Derived extends Base {}

const obj = new Derived();
// obj ---> Derived.prototype ---> Base.prototype ---> Object.prototype ---> null
```

你可能还会看到一些使用 [`Object.create\(\)`](#) 来构建继承链的旧代码。然而，因为这会重新为 `prototype` 属性赋值并删除 [`constructor`](#) 属性，所以更容易出错，而且如果构造函数还没有创建任何实例，性能提升可能并不明显。

---

```
function Base() {}
function Derived() {}

// 将 `Derived.prototype` 重新赋值为 `Base.prototype` ,
// 以作为其 `[[Prototype]]` 的新对象
// 请不要这样做—使用 Object.setPrototypeOf 来修改它
Derived.prototype = Object.create(Base.prototype);
```

## 检查原型：更深入的研究

让我们来仔细看看幕后发生了什么。

如上所述，在 JavaScript 中，函数可以拥有属性。所有函数都有一个名为 `prototype` 的特殊属性。请注意，下面的代码是独立的（出于严谨，假设页面没有其他的 JavaScript 代码）。为获得最佳的学习体验，强烈建议你打开控制台，进入“console”标签页，复制并粘贴以下 JavaScript 代码，然后按回车键运行。（大多数 web 浏览器的开发者工具中都包含控制台。请参阅 [Firefox 开发者工具](#)、[Chrome 开发者工具](#) 和 [Edge 开发者工具](#)，以了解详情。）

---

```
function doSomething() {}
console.log(doSomething.prototype);
// 你如何声明函数并不重要;
// JavaScript 中的函数总有一个默认的
// 原型属性—有一个例外:
// 箭头函数没有默认的原型属性:
const doSomethingFromArrowFunction = () => {};
console.log(doSomethingFromArrowFunction.prototype);
```

如上所示，`doSomething()` 有一个默认的 `prototype` 属性（正如控制台所示）。运行这段代码后，控制台应该显示一个类似于下面的对象。

```
{
  constructor: f doSomething(),
  [[Prototype]]: {
    constructor: f Object(),
    hasOwnProperty: f hasOwnProperty(),
    isPrototypeOf: f isPrototypeOf(),
    propertyIsEnumerable: f propertyIsEnumerable(),
    toLocaleString: f toLocaleString(),
```

```
        toString: f toString(),
        valueOf: f valueOf()
    }
}
```

**备注：** Chrome 控制台使用 `[[Prototype]]` 来表示对象的原型，遵循规范的术语；Firefox 使用 `<prototype>`。为了保持一致性，我们将使用 `[[Prototype]]`。

我们可以像下面这样，向 `doSomething()` 的原型添加属性。

JS

```
function doSomething() {}
doSomething.prototype.foo = "bar";
console.log(doSomething.prototype);
```

其结果为：

```
{
  foo: "bar",
  constructor: f doSomething(),
  [[Prototype]]: {
    constructor: f Object(),
    hasOwnProperty: f hasOwnProperty(),
    isPrototypeOf: f isPrototypeOf(),
    propertyIsEnumerable: f propertyIsEnumerable(),
    toLocaleString: f toLocaleString(),
    toString: f toString(),
    valueOf: f valueOf()
  }
}
```

我们现在可以使用 `new` 运算符来创建基于该原型的 `doSomething()` 的实例。要使用 `new` 运算符，只需像往常一样调用函数，只是要在前面加上 `new`。使用 `new` 运算符调用函数会返回一个函数的实例对象。然后可以在该对象上添加属性。

尝试以下代码：

JS

```
function doSomething() {}  
doSomething.prototype.foo = "bar"; // 向原型上添加一个属性  
const doSomeInstancing = new doSomething();  
doSomeInstancing.prop = "some value"; // 向该对象添加一个属性  
console.log(doSomeInstancing);
```

这会产生类似于下面的输出：

```
{  
  prop: "some value",  
  [[Prototype]]: {  
    foo: "bar",  
    constructor: f doSomething(),  
    [[Prototype]]: {  
      constructor: f Object(),  
      hasOwnProperty: f hasOwnProperty(),  
      isPrototypeOf: f isPrototypeOf(),  
      propertyIsEnumerable: f propertyIsEnumerable(),  
      toLocaleString: f toLocaleString(),  
      toString: f toString(),  
      valueOf: f valueOf()  
    }  
  }  
}
```

如上所示，`doSomeInstancing` 的 `[[Prototype]]` 是 `doSomething.prototype`。但是，这是做什么的呢？当你访问 `doSomeInstancing` 的属性时，运行时首先会查找 `doSomeInstancing` 是否有该属性。

如果 `doSomeInstancing` 没有该属性，那么运行时会在 `doSomeInstancing.[[Prototype]]`（也就是 `doSomething.prototype`）中查找该属性。如果 `doSomeInstancing.[[Prototype]]` 有该属性，那么就会使用 `doSomeInstancing.[[Prototype]]` 上的该属性。

否则，如果 `doSomeInstancing.[[Prototype]]` 没有该属性，那么就会在 `doSomeInstancing.[[Prototype]].[[Prototype]]` 中查找该属性。默认情况下，任何函数的 `prototype` 属性的 `[[Prototype]]` 都是 `Object.prototype`。因此会在 `doSomeInstancing.[[Prototype]].[[Prototype]]`（也就是 `doSomething.prototype.[[Prototype]]`（也就是 `Object.prototype`））上查找该属性。

如果在 `doSomeInstancing.[[Prototype]].[[Prototype]]` 中没有找到该属性，那么就会在 `doSomeInstancing.[[Prototype]].[[Prototype]].[[Prototype]]` 中查找该属性。但是，这里有一个

问题：`doSomeInstancing.[[Prototype]].[[Prototype]].[[Prototype]]` 不存在，因为  
`Object.prototype.[[Prototype]]` 是 `null`。然后，只有在查找完整个 `[ [ Prototype ] ]` 链之后，运行时才会断言该属性不存在，并得出该属性的值为 `undefined`。

让我们在控制台中输入更多的代码：

JS

```
function doSomething() {}  
doSomething.prototype.foo = "bar";  
const 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);
```

其结果如下：

```
doSomeInstancing.prop:      some value  
doSomeInstancing.foo:       bar  
doSomething.prop:        undefined  
doSomething.foo:         undefined  
doSomething.prototype.prop: undefined  
doSomething.prototype.foo:  bar
```

## 使用不同的方法来创建对象和改变原型链

我们碰到过很多创建对象和改变其原型链的方法。我们将系统地总结不同的方法，并比较每种方法的优缺点。

### 使用语法结构创建对象

JS

```
const o = { a: 1 };  
// 新创建的对象 o 以 Object.prototype 作为它的 [[Prototype]]  
// Object.prototype 的原型为 null。  
// o ---> Object.prototype ---> null
```

```

const b = ["yo", "whadup", "?"];
// 数组继承了 Array.prototype (具有 indexOf、forEach 等方法)
// 其原型链如下所示:
// b ---> Array.prototype ---> Object.prototype ---> null

function f() {
  return 2;
}

// 函数继承了 Function.prototype (具有 call、bind 等方法)
// f ---> Function.prototype ---> Object.prototype ---> null

const p = { b: 2, __proto__: o };
// 可以通过 __proto__ 字面量属性将新创建对象的
// [[Prototype]] 指向另一个对象。
// (不要与 Object.prototype.__proto__ 访问器混淆)
// p ---> o ---> Object.prototype ---> null

```

### 在对象初始化器中使用 \_\_proto\_\_ 键的优缺点

<b>优点</b>	被所有的现代引擎所支持。将 __proto__ 属性指向非对象的值只会被忽略，而非抛出异常。与 <code>Object.prototype.__proto__</code> setter 相反，对象字面量初始化器中的 __proto__ 是标准化，被优化的。甚至可以比 <code>Object.create</code> 更高效。在创建对象时声明额外的自有属性比 <code>Object.create</code> 更符合习惯。
<b>缺点</b>	不支持 IE10 及以下的版本。对于不了解其与 <code>Object.prototype.__proto__</code> 访问器差异的人可能会将两者混淆。

## 使用构造函数

JS

```

function Graph() {
  this.vertices = [];
  this.edges = [];
}

Graph.prototype.addVertex = function (v) {
  this.vertices.push(v);
};

const g = new Graph();
// g 是一个带有自有属性“vertices”和“edges”的对象。
// 在执行 new Graph() 时, g.[[Prototype]] 是 Graph.prototype 的值。

```

## 使用构造函数的优缺点

优点	所有引擎都支持——直到 IE 5.5。此外，其速度很快、非常标准，且极易被 JIT 优化。
缺点	<ul style="list-style-type: none"><li>要使用这个方法，必须初始化该函数。在初始化过程中，构造函数可能会存储每一个对象都必须生成的唯一信息。这些唯一信息只会生成一次，可能会导致问题。</li><li>构造函数的初始化过程可能会将不需要的方法放到对象上。</li></ul> <p>这两者在实践中通常都不是问题。</p>

## 使用 Object.create()

调用 [Object.create\(\)](#) 来创建一个新对象。该对象的 `[[Prototype]]` 是该函数的第一个参数：

JS

```
const a = { a: 1 };
// a ---> Object.prototype ---> null

const b = Object.create(a);
// b ---> a ---> Object.prototype ---> null
console.log(b.a); // 1 (inherited)

const c = Object.create(b);
// c ---> b ---> a ---> Object.prototype ---> null

const d = Object.create(null);
// d ---> null (d 是一个直接以 null 为原型的对象)
console.log(d.hasOwnProperty());
// undefined, 因为 d 没有继承 Object.prototype
```

## [Object.create](#) 的优缺点

优点	被所有现代引擎所支持。允许在创建时直接设置对象的 <code>[[Prototype]]</code> ，这允许运行时进一步优化对象。还允许使用 <code>Object.create(null)</code> 创建没有原型的对象。
----	--

## 缺点

不支持 IE8 及以下版本。但是，由于微软已经停止了对运行 IE8 及以下版本的系统的扩展支持，这对大多数应用程序而言应该不是问题。此外，如果使用了第二个参数，慢对象的初始化可能会成为性能瓶颈，因为每个对象描述符属性都有自己单独的描述符对象。当处理上万个对象描述符时，这种延时可能会成为一个严重的问题。

## 使用类

JS

```
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;
  }
}

const square = new Square(2);
// square ---> Square.prototype ---> Polygon.prototype ---> Object.prototype ---> null
```

## 类的优缺点

## 优点

被所有现代引擎所支持。非常高的可读性和可维护性。[私有属性](#)是原型继承中没有简单替代方案的特性。

## 缺点

类，尤其是带有私有属性的类，比传统的类的性能要差（尽管引擎实现者正在努力改进这一点）。不支持旧环境，通常需要转译器才能在生产中使用类。

## 使用 Object.setPrototypeOf()

虽然上面的所有方法都会在对象创建时设置原型链，但是 [Object.setPrototypeOf\(\)](#) 允许修改现有对象的 `[[Prototype]]` 内部属性。

JS

```
const obj = { a: 1 };
const anotherObj = { b: 2 };
Object.setPrototypeOf(obj, anotherObj);
// obj ---> anotherObj ---> Object.prototype ---> null
```

### [Object.setPrototypeOf](#) 的优缺点

优点	被所有现代引擎所支持。允许动态地修改对象的原型，甚至可以强制为使用 <code>Object.create(null)</code> 创建的无原型对象设置原型。
缺点	性能不佳。如果可以在创建对象时设置原型，则应避免此方法。许多引擎会优化原型，并在调用实例时会尝试提前猜测方法在内存中的位置；但是动态设置原型会破坏这些优化。它可能会导致某些引擎重新编译你的代码以进行反优化，以使其按照规范工作。不支持 IE8 及以下版本。

## 使用 `__proto__` 访问器

所有对象都继承了 [Object.prototype.\\_\\_proto\\_\\_](#) 访问器，它可以用来设置现有对象的 `[[Prototype]]`（如果对象没有覆盖 `__proto__` 属性）。

**警告：** `Object.prototype.__proto__` 访问器是**非标准的**，且已被弃用。你几乎总是应该使用 `Object.setPrototypeOf` 来代替。

JS

```
const obj = {};
// 请不要使用该方法：仅作为示例。
obj.__proto__ = { barProp: "bar val" };
obj.__proto__.__proto__ = { fooProp: "foo val" };
console.log(obj.fooProp);
console.log(obj.barProp);
```

## 设置 `__proto__` 属性的优缺点

优点	被所有现代引擎所支持。将 <code>__proto__</code> 设置为非对象的值只会被忽略，而非抛出异常。
缺点	性能不佳且已被弃用。许多引擎会优化原型，并在调用实例时会尝试提前猜测方法在内存中的位置；但是动态设置原型会破坏这些优化，甚至可能会导致某些引擎重新编译你的代码以进行反优化，以使其按照规范工作。不支持 IE10 及以下版本。 <code>__proto__</code> 访问器是规范中可选的特性，因此可能无法在所有平台上使用。你几乎总是应该使用 <code>Object.setPrototypeOf</code> 代替。

## 性能

原型链上较深层的属性的查找时间可能会对性能产生负面影响，这在性能至关重要的代码中可能会非常明显。此外，尝试访问不存在的属性始终会遍历整个原型链。

此外，在遍历对象的属性时，原型链中的每个可枚举属性都将被枚举。要检查对象是否具有在其自身上定义的属性，而不是在其原型链上的某个地方，则有必要使用 `hasOwnProperty` 或 `Object.hasOwnProperty` 方法。除 `[[Prototype]]` 为 `null` 的对象外，所有对象都从 `Object.prototype` 继承 `hasOwnProperty`——除非它已经在原型链的更深处被覆盖。我们将使用上面的图示例代码来说明它，具体如下：

JS

```
function Graph() {
  this.vertices = [];
  this.edges = [];
}

Graph.prototype.addVertex = function (v) {
  this.vertices.push(v);
};

const g = new Graph();
// g ---> Graph.prototype ---> Object.prototype ---> null

g.hasOwnProperty("vertices"); // true
Object.hasOwnProperty(g, "vertices"); // true

g.hasOwnProperty("nope"); // false
Object.hasOwnProperty(g, "nope"); // false
```

```
g.hasOwnProperty("addVertex"); // false  
Object.hasOwnProperty(g, "addVertex"); // false  
  
Object.getPrototypeOf(g).hasOwnProperty("addVertex"); // true
```

注意：仅检查属性是否为 `undefined` 是**不够的**。该属性很可能存在，但其值恰好设置为 `undefined`。

## 结论

对于 Java 或 C++ 的开发者来说，JavaScript 可能有点令人困惑，因为它是完全动态、完全是在执行期间确定的，而且根本没有静态类型。一切都是对象（实例）或函数（构造函数），甚至函数本身也是 `Function` 构造函数的实例。即使是语法结构中的“类”也只是运行时的构造函数。

JavaScript 中的所有构造函数都有一个被称为 `prototype` 的特殊属性，它与 `new` 运算符一起使用。对原型对象的引用被复制到新实例的内部属性 `[[Prototype]]` 中。例如，当你执行 `const a1 = new A()` 时，JavaScript（在内存中创建对象之后，为其定义 `this` 并执行 `A()` 之前）设置 `a1.[[Prototype]] = A.prototype`。然后，当你访问实例的属性时，JavaScript 首先检查它们是否直接存在于该对象上，如果不存在，则在 `[[Prototype]]` 中查找。会递归查询 `[[Prototype]]`，即 `a1.doSomething`、`Object.getPrototypeOf(a1).doSomething`、`Object.getPrototypeOf(Object.getPrototypeOf(a1)).doSomething`，以此类推，直至找到或 `Object.getPrototypeOf` 返回 `null`。这意味着在 `prototype` 上定义的所有属性实际上都由所有实例共享，并且甚至可以更改 `prototype` 的部分内容，使得更改被应用到所有现有的实例中。

在上面的示例中，如果你执行 `const a1 = new A(); const a2 = new A();`，那么 `a1.doSomething` 实际上会引用 `Object.getPrototypeOf(a1).doSomething`——这与你定义的 `A.prototype.doSomething` 相同，即 `Object.getPrototypeOf(a1).doSomething === Object.getPrototypeOf(a2).doSomething === A.prototype.doSomething`。

了解原型继承模型是使用它编写复杂代码的重要基础。此外，要注意代码中原型链的长度，在必要时可以将其分解，以避免潜在的性能问题。此外，除非是为了与新的 JavaScript 特性兼容，否则**永远不应扩展原生原型**。

Help improve MDN

Was this page helpful to you?



Yes

No

[Learn how to contribute.](#)



This page was last modified on 2023年11月9日 by [MDN contributors](#).