对象原型

原型是 JavaScript 对象相互继承特性的一种机制。在本文中，我们将解释什么是原型、原型链如何工作以及如何设置对象的原型。

先决条件：	了解 JavaScript 函数，熟悉 JavaScript 基础知识（请参阅第一步和 构建块）和 OOJS 基础知识（请参阅 对象简介）。
客观的：	了解 JavaScript 对象原型、原型链如何工作以及如何设置对象的原型。

原型链

在浏览器的控制台中，尝试创建一个对象字面量：

```
const myObject = {
  city: "Madrid",
  greet() {
    console.log(`Greetings from ${this.city}`);
  },
};

myObject.greet(); // Greetings from Madrid
```

这是一个具有一个数据属性 `city` 和一个方法 的对象 `greet()`。如果您在控制台中键入对象名称*后跟一个句点* `myObject.`，例如，那么控制台将弹出该对象可用的所有属性的列表。您会看到它以及 `city` 和 `greet`，还有许多其他属性！

```
__defineGetter__
__defineSetter__
__lookupGetter__
```

```
__lookupSetter__
__proto__
city
constructor
greet
hasOwnProperty
isPrototypeOf
propertyIsEnumerable
toLocaleString
toString
valueOf
```

尝试访问其中之一：

```
myObject.toString(); // "[object Object]"
```

它有效（即使它的作用并不明显 `toString()`）。

这些额外的属性是什么，它们来自哪里?

JavaScript 中的每个对象都有一个内置属性，称为其**原型**。原型本身就是一个对象，所以原型会有自己的原型，这就是所谓的**原型链**。`null` 当我们到达一个有自己原型的原型时，链结束。

> **注意：** 指向其原型的对象的属性**不**被调用 `prototype` 。它的名称并不标准，但实际上所有浏览器都使用 `__proto__` . 访问对象原型的标准方法是 `Object.getPrototypeOf()` 方法。

当您尝试访问对象的属性时：如果在对象本身中找不到该属性，则会在原型中搜索该属性。如果仍然找不到属性，则搜索原型的原型，依此类推，直到找到属性或到达链的末尾，在这种情况下返回 `undefined` 。

所以当我们调用时 `myObject.toString()`，浏览器：

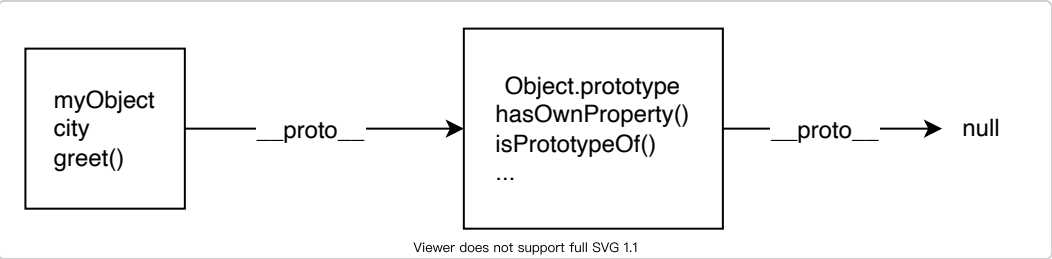
- `toString` 在寻找 `myObject`
- 在那里找不到它，所以在 `myObject` for的原型对象中查找 `toString`

- 在那里找到它，并调用它。

什么是原型 `myObject`？ 为了找出答案，我们可以使用函数 `Object.getPrototypeOf()`：

```
Object.getPrototypeOf(myObject); // Object { }
```

这是一个名为 的对象 `Object.prototype`， 它是最基本的原型，所有对象默认都有。的原型 `Object.prototype` 是 `null`，所以它在原型链的末尾：



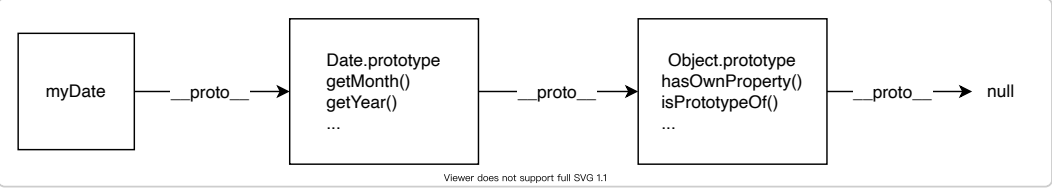
对象的原型并不总是 `Object.prototype`。尝试这个：

```
const myDate = new Date();
let object = myDate;

do {
  object = Object.getPrototypeOf(object);
  console.log(object);
} while (object);

// Date.prototype
// Object { }
// null
```

此代码创建一个 `Date` 对象，然后沿着原型链向上走，记录原型。它告诉我们，的原型 `myDate` 是一个对象， *that* `Date.prototype` 的原型是。`Object.prototype`



事实上，当您调用熟悉的方法（如 ）时 `myDate2.getMonth()`，您是在调用定义在 上的方法 `Date.prototype` 。

阴影属性

如果在对象的原型中定义了具有相同名称的属性时，如果在对象中定义属性会发生什么情况？让我们来看看：

```
const myDate = new Date(1995, 11, 17);

console.log(myDate.getYear()); // 95

myDate.getYear = function () {
  console.log("something else!");
};

myDate.getYear(); // 'something else!'
```

鉴于原型链的描述，这应该是可以预见的。当我们调用 `getYear()` 浏览器时，首先查找具有该名称的属性，如果没有定义它， `myDate` 则只检查原型。 `myDate` 所以当我们添加 `getYear()` 到的时候 `myDate` ，那么in的版本 `myDate` 就会被调用。

这称为“隐藏”属性。

设置原型

在 JavaScript 中有多种设置对象原型的方法，这里我们将介绍两种： `Object.create()` 和构造函数。

使用 Object.create

该 `Object.create()` 方法创建一个新对象并允许您指定将用作新对象原型的对象。

这是一个例子：

```
const personPrototype = {
  greet() {
    console.log("hello!");
  },
};

const carl = Object.create(personPrototype);
carl.greet(); // hello!
```

这里我们创建了一个对象 `personPrototype`， 它有一个 `greet()` 方法。然后我们使用 `Object.create()` 创建一个新对象作为 `personPrototype` 它的原型。现在我们可以调用 `greet()` 新对象，原型提供了它的实现。

使用构造函数

在 JavaScript 中，所有函数都有一个名为 `prototype` . 当您将函数作为构造函数调用时，此属性将设置为新构造对象的原型（按照惯例，在名为 的属性中 `__proto__` ）。

因此，如果我们设置 `prototype` 构造函数的 ， 我们可以确保使用该构造函数创建的所有对象都被赋予该原型：

```
const personPrototype = {
  greet() {
    console.log(`hello, my name is ${this.name}!`);
  },
};

function Person(name) {
  this.name = name;
}

Object.assign(Person.prototype, personPrototype);
// or
// Person.prototype.greet = personPrototype.greet;
```

在这里我们创建：

- 一个对象 `personPrototype`， 它有一个 `greet()` 方法

- 一个 `Person()` 构造函数，它初始化要创建的人的名字。

然后，我们使用[Object.assign](#)将定义的方法放入函数的属性 `personPrototype` 中。 `Person prototype`

在这段代码之后，使用创建的对象 `Person()` 将 `Person.prototype` 作为它们的原型，它自动包含该 `greet` 方法。

```
const reuben = new Person("Reuben");
reuben.greet(); // hello, my name is Reuben!
```

这也解释了为什么我们前面说的原型 `myDate` 被调用 `Date.prototype`： 它是构造函数 `prototype` 的属性 `Date` 。

自有物业

我们使用上面的构造函数创建的对象 `Person` 有两个属性：

- 一个 `name` 在构造函数中设置的属性，因此它直接出现在 `Person` 对象上
- 一个 `greet()` 方法，在原型中设置。

这种模式很常见，方法在原型上定义，但数据属性在构造函数中定义。这是因为我们创建的每个对象的方法通常都是相同的，而我们通常希望每个对象的数据属性都有自己的值（就像这里每个人都有不同的名字一样）。

像这里这样直接在对象中定义的属性 `name` 称为**自有属性**，您可以使用静态方法检查属性是否为自有属性 [`Object.hasOwn()`](#)：

```
const irma = new Person("Irma");

console.log(Object.hasOwn(irma, "name")); // true
console.log(Object.hasOwn(irma, "greet")); // false
```

> 法，但我们建议你 `Object.hasOwn()` 尽可能使用。

原型和继承

原型是 JavaScript 的一个强大且非常灵活的特性，使得重用代码和组合对象成为可能。

特别是它们支持继承的版本。继承是面向对象编程语言的一个特性，它让程序员表达这样的想法，即系统中的某些对象是其他对象的更特殊版本。

例如，如果我们正在为一所学校建模，我们可能有*教授*和*学生*：他们都是人，所以有一些共同的特征（例如，他们都有名字），但每个人都可能添加额外的特征（例如，教授有他们教授的主题），或者可能以不同的方式实现相同的功能。在 OOP 系统中，我们可以说教授和学生都**继承**自人。

你可以看到在 JavaScript 中，如果 `Professor` 和 `Student` 对象可以有 `Person` 原型，那么它们可以继承共同的属性，同时添加和重新定义那些需要不同的属性。

在下一篇文章中，我们将讨论继承以及面向对象编程语言的其他主要特性，并了解 JavaScript 如何支持它们。

概括

本文涵盖了 JavaScript 对象原型，包括原型对象链如何允许对象相互继承特性、原型属性以及如何使用它向构造函数添加方法，以及其他相关主题。

在下一篇文章中，我们将了解面向对象编程的基本概念。

此页面最后修改于 2023 年 2 月 24 日由MDN 贡献者提供。