Object

在本文中

- 描述
- 构造函数
- 静态方法
- 实例属性
- 实例方法
- 示例
- 规范
- 浏览器兼容性
- 参见

Baseline Widely available

`Object` 是 JavaScript 的一种数据类型。它用于存储各种键值集合和更复杂的实体。可以通过 `Object()` 构造函数或者使用对象字面量的方式创建对象。

描述

在 JavaScript 中，几乎所有的对象都是 `Object` 的实例；一个典型的对象从 `Object.prototype` 继承属性（包括方法），尽管这些属性可能被覆盖（或者说重写）。唯一不从 `Object.prototype` 继承的对象是那些 `null` 原型对象，或者是从其他 `null` 原型对象继承而来的对象。

通过原型链，所有对象都能观察到 `Object.prototype` 对象的改变，除非这些改变所涉及的属性和方法沿着原型链被进一步重写。尽管有潜在的危险，但这为覆盖或扩展对象的行为提供了一个非常强大的机制。为了使其更加安全，`Object.prototype` 是核心 JavaScript 语言中唯一具有不可变原型的对象——`Object.prototype` 的原型始终为 `null` 且不可更改。

JavaScript 标准内置对象

对象原型属性

你应该避免调用任何 `Object.prototype` 方法，特别是那些不打算多态化的方法（即只有其初始行为是合理的，且无法被任何继承的对象以合理的方式重写）。所有从 `Object.prototype` 继承的对象都可以自定义一个具有相同名称但语义可能与你的预期不同的自有属性。此外，这些属性不会被 `null` 原型对象继承。现代 JavaScript 中用于操作对象的工具方法都是静态的。更具体地说：

- `valueOf()`、`toString()` 和 `toLocaleString()` 存在的目的是为了多态化，你应该期望对象会定义自己的实现并具有合理的行为，因此你可以将它们作为实例方法调用。但是，`valueOf()` 和 `toString()` 通常是通过强制类型转换隐式调用的，因此你不需要在代码中自己调用它们。
- `__defineGetter__()`、`__defineSetter__()`、`__lookupGetter__()` 和 `__lookupSetter__()` 已被弃用，不应该再使用。请使用静态方法 `Object.defineProperty()` 和 `Object.getOwnPropertyDescriptor()` 作为替代。
- `__proto__` 属性已被弃用，不应该再使用。请使用静态方法 `Object.getPrototypeOf()` 和 `Object.setPrototypeOf()` 作为替代。
- `propertyIsEnumerable()` 和 `hasOwnProperty()` 方法可以分别用静态方法 `Object.getOwnPropertyDescriptor()` 和 `Object.hasOwn()` 替换。
- 如果你正在检查一个构造函数的 `prototype` 属性，通常可以用 `instanceof` 代替 `isPrototypeOf()` 方法。

如果不存在语义上等价的静态方法，或者你真的想使用 `Object.prototype` 方法，你应该通过 `call()` 直接在目标对象上调用方法，以防止因目标对象上原有方法被重写而产生意外的结果。

JS

```
const obj = {
  foo: 1,
  // 如果可能的话，你不应该在自己的对象上定义这样的方法，
  // 但是如果你从外部输入接收对象，可能无法防止这种情况的发生
  propertyIsEnumerable() {
    return false;
  },
};
```

};
Object.hasOwn()

obj.propertyIsEnumerable("foo"); // false；预期外的结果
Object.prototype.propertyIsEnumerable.call(obj, "foo"); // true；预期的结果

Object.isExtensible()

从对象中删除属性

一个对象本身没有任何方法可以（像 Map.prototype.delete() 一样）删除自己的属性。要删除一个对象的属性，必须使用 delete 运算符。

null 原型对象

几乎所有的 JavaScript 对象最终都继承自 Object.prototype （参见继承与原型链）。然而，你可以使用 Object.create(null) 或定义了 __proto__: null 的对象字面量语法（注意：对象字面量中的 __proto__ 键不同于已弃用的 Object.prototype.__proto__ 属性）来创建 null 原型对象。你还可以通过调用 Object.setPrototypeOf(obj, null) 将现有对象的原型更改为 null 。

▼ 实例方法

JS

```
Object.prototype.__defineGetter__()
const obj = Object.create(null);
const obj2 = { __proto__: null };
Object.prototype.__defineSetter__()
```

null 原型对象可能会有一些预期外的行为表现，因为它不会从 Object.prototype 继承任何对象方法。这在调试时尤其需要注意，因为常见的对象属性转换/检测实用方法可能会产生错误或丢失信息（特别是在使用了忽略错误的静默错误捕获机制的情况下）。

例如， Object.prototype.toString() 方法的缺失通常会使得调试变得困难：

JS

```
const normalObj = {}; // 创建一个普通对象
const nullProtoObj = Object.create(null); // 创建一个 "null" 原型对象

console.log(`normalObj 是: ${normalObj}`); // 显示 "normalObj 是: [object Object]"
console.log(`nullProtoObj 是: ${nullProtoObj}`); // 抛出错误: Cannot convert object to primitive value
```

其他方法也会失败。

JS

```
normalObj.valueOf(); // 显示 {}
nullProtoObj.valueOf(); // 抛出错误: nullProtoObj.valueOf is not a function

normalObj.hasOwnProperty("p"); // 显示 "true"
nullProtoObj.hasOwnProperty("p"); // 抛出错误: nullProtoObj.hasOwnProperty is not a function
```

继承

Object/Function

▶ 静态方法

▶ 静态属性

▶ 实例方法

▶ 实例属性

我们可以通过为 null 原型对象分配属性的方式将 toString 方法添加回去：

JS

```
nullProtoObj.toString = Object.prototype.toString; // 由于新对象缺少 `toString` 方法，因此需要将原始的通用 `toString` 方法添加回来。

console.log(nullProtoObj.toString()); // 显示 "[object Object]"
console.log(`nullProtoObj 是: ${nullProtoObj}`); // 显示 "nullProtoObj 是: [object Object]"
```

普通对象的 toString() 方法是在对象的原型上的，而与普通对象不同的是，这里的 toString() 方法是 nullProtoObj 的

自有属性。这是因为 `nullProtoObj` 没有原型（即为 `null`）。

在实践中，`null` 原型对象通常被用作 map 的简单替代品。由于存在 `Object.prototype` 属性，会导致一些错误：

JS

```
const ages = { alice: 18, bob: 27 };

function hasPerson(name) {
  return name in ages;
}

function getAge(name) {
  return ages[name];
}

hasPerson("hasOwnProperty"); // true
getAge("toString"); // [Function: toString]
```

使用一个 `null` 原型对象可以消除这种风险，同时不会令 `hasPerson` 和 `getAge` 函数变得复杂：

JS

```
const ages = Object.create(null, {
  alice: { value: 18, enumerable: true },
  bob: { value: 27, enumerable: true },
});

hasPerson("hasOwnProperty"); // false
getAge("toString"); // undefined
```

在这种情况下，添加任何方法都应该慎重，因为它们可能会与存储为数据的其他键值对混淆。

让你的对象不继承自 `Object.prototype` 还可以防止原型污染攻击。如果恶意脚本向 `Object.prototype` 添加一个属性，程序中的每个对象上都可访问它，除了那些原型为 `null` 的对象。

JS

```
const user = {};

// 恶意脚本：
Object.prototype.authenticated = true;

// 意外允许未经身份验证的用户通过
if (user.authenticated) {
  // 访问机密数据
}
```

JavaScript 还具有内置的 API，用于生成 `null` 原型对象，特别是那些将对象用作临时键值对集合的 API。例如：

- `Object.groupBy()` 方法的返回值
- `RegExp.prototype.exec()` 方法返回结果中的 `groups` 和 `indices.groups` 属性
- `Array.prototype[Symbol.unscopables]` 属性（所有 `[Symbol.unscopables]` 对象原型都应该为 `null`）
- `import.meta` 对象
- 通过 `import * as ns from "module"` 或 `import()` (en-US) 获取的模块命名空间对象

“`null` 原型对象”这个术语通常也包括其原型链中没有 `Object.prototype` 的任何对象。当使用类时，可以通过 `extends null` 来创建这样的对象。

对象强制转换

许多内置操作首先将它们的参数强制转换为对象。该过程 可以概括如下：

- 对象则按原样返回。
- undefined 和 null 则抛出 TypeError。
- Number、String、Boolean、Symbol、BigInt 等基本类型被封装成其对应的基本类型对象。

在 JavaScript 中实现相同效果的最佳方式是使用 Object() 构造函数。Object(x) 可以将 x 转换为对象，对于 undefined 或 null，它会返回一个普通对象而不是抛出 TypeError 异常。

使用对象强制转换的地方包括：

- for...in 循环的 object 参数。
- Array 方法的 this 值。
- Object 方法的参数，如 Object.keys()。
- 当访问基本类型的属性时进行自动转换，因为基本类型没有属性。
- 在调用非严格函数时的 this 值。基本类型值被封装为对象，而 null 和 undefined 被替换为全局对象。

与转换为基本类型不同，对象强制转换过程本身无法以任何方式被观察到，因为它不会调用像 toString 或 valueOf 方法这样的自定义代码。

构造函数

Object()
将输入转换为一个对象。

静态方法

Object.assign()
将一个或多个源对象的所有可枚举自有属性的值复制到目标对象中。

Object.create()
使用指定的原型对象和属性创建一个新对象。

Object.defineProperties()
向对象添加多个由给定描述符描述的命名属性。

Object.defineProperty()
向对象添加一个由给定描述符描述的命名属性。

Object.entries()
返回包含给定对象自有可枚举字符串属性的所有 [key, value] 数组。

Object.freeze()
冻结一个对象。其他代码不能删除或更改其任何属性。

Object.fromEntries()
从一个包含 [key, value] 对的可迭代对象中返回一个新的对象（Object.entries 的反操作）。

Object.getOwnPropertyDescriptor()
返回一个对象的已命名属性的属性描述符。

Object.getOwnPropertyDescriptors()
返回一个包含对象所有自有属性的属性描述符的对象。

`Object.getOwnPropertyNames()`

返回一个包含给定对象的所有自有可枚举和不可枚举属性名称的数组。

`Object.getOwnPropertySymbols()`

返回一个数组，它包含了指定对象所有自有 symbol 属性。

`Object.getPrototypeOf()`

返回指定对象的原型（内部的 `[[Prototype]]` 属性）。

`Object.hasOwn()`

如果指定属性是指定对象的自有属性，则返回 `true`，否则返回 `false`。如果该属性是继承的或不存在，则返回 `false`。

`Object.is()`

比较两个值是否相同。所有 `NaN` 值都相等（这与 `==` 使用的 `IsLooselyEqual` 和 `===` 使用的 `IsStrictlyEqual` 不同）。

`Object.isExtensible()`

判断对象是否可扩展。

`Object.isFrozen()`

判断对象是否已经冻结。

`Object.isSealed()`

判断对象是否已经封闭。

`Object.keys()`

返回一个包含所有给定对象自有可枚举字符串属性名称的数组。

`Object.preventExtensions()`

防止对象的任何扩展。

`Object.seal()`

防止其他代码删除对象的属性。

`Object.setPrototypeOf()`

设置对象的原型（即内部 `[[Prototype]]` 属性）。

`Object.values()`

返回包含给定对象所有自有可枚举字符串属性的值的数组。

实例属性

这些属性在 `Object.prototype` 上定义，被所有 `Object` 实例所共享。

`Object.prototype.__proto__`

指向实例对象在实例化时使用的原型对象。

`Object.prototype.constructor`

创建该实例对象的构造函数。对于普通的 `Object` 实例，初始值为 `Object` 构造函数。其它构造函数的实例都会从它们各自的 `Constructor.prototype` 对象中继承 `constructor` 属性。

实例方法

Object.prototype.__defineGetter__()

将一个属性与一个函数相关联，当该属性被访问时，执行该函数，并且返回函数的返回值。

Object.prototype.__defineSetter__()

将一个属性与一个函数相关联，当该属性被设置时，执行该函数，执行该函数去修改某个属性。

Object.prototype.__lookupGetter__()

返回绑定在指定属性上的 getter 函数。

Object.prototype.__lookupSetter__()

返回绑定在指定属性上的 setter 函数。

Object.prototype.hasOwnProperty()

返回一个布尔值，用于表示一个对象自身是否包含指定的属性，该方法并不会查找原型链上继承来的属性。

Object.prototype.isPrototypeOf()

返回一个布尔值，用于表示该方法所调用的对象是否在指定对象的原型链中。

Object.prototype.propertyIsEnumerable()

返回一个布尔值，指示指定属性是否是对象的可枚举自有属性。

Object.prototype.toLocaleString()

调用 toString() 方法。

Object.prototype.toString()

返回一个代表该对象的字符串。

Object.prototype.valueOf()

返回指定对象的基本类型值。

示例

构造空对象

以下示例使用带有不同参数的 new 关键字创建空对象：

JS

```
const o1 = new Object();
const o2 = new Object(undefined);
const o3 = new Object(null);
```

使用 Object 生成布尔对象

下面的例子将 Boolean 对象存到 o 中：

JS

```
// 等价于 const o = new Boolean(true);
const o = new Object(true);
```

JS

```
// 等价于 const o = new Boolean(false);
const o = new Object(Boolean());
```

Object prototype

当我们要修改现有的 `Object.prototype` 方法时，请你考虑一下在现有逻辑之前或者之后通过包装扩展代码的方式来注入代码。例如，以下（未经测试的）代码将会在执行内部逻辑或者是其他扩展之前，有条件地执行一段自定义的逻辑。

在使用钩子修改原型时，通过在函数上调用 `apply()` 方法并传递 `this` 和参数（即调用状态），将其传递给当前行为。这种模式可以用于任何原型，例如 `Node.prototype` 、 `Function.prototype` 等。

JS

```
const current = Object.prototype.valueOf;

// 由于我的属性"-prop-value"是横跨多个原型链的，并且不总是在同一个原型链上，
// 因此我想修改 Object.prototype：
Object.prototype.valueOf = function (...args) {
  if (Object.hasOwn(this, "-prop-value")) {
    return this["-prop-value"];
  } else {
    // 这似乎不是我的对象，因此让我们尽可能实现默认行为。
    // 在某些其他语言中，apply 的行为类似于 "super"。
    // 即使 valueOf() 不需要参数，但其他的方法可能需要参数。
    return current.apply(this, args);
  }
};
```

> 警告： 修改任何内置构造函数的 `prototype` 属性被认为是一种不好的做法，可能会影响向前兼容性。

你可以阅读更多关于原型的内容，参见继承与原型链。

规范

Specification

ECMAScript Language Specification
sec-object-objects

浏览器兼容性

BCD tables only load in the browser

参见

- 初始化对象