

# 属性描述对象

---

## 1. 概述

JavaScript 提供一个内部数据结构，用来描述对象的属性，控制它的行为，比如该属性是否可以遍历、可写等。这个内部数据结构成为“属性描述对象”。每个属性都有自己对应的属性描述对象，保存改属性的一些元信息。

```
{
  value: 123,
  writable: false,
  enumerable: true,
  configurable: false,
  get: undefined,
  set: undefined
}
```

属性描述对象提供 6 个原属性。

(1) `value`

属性值，默认 `undefined`。

(2) `writable`

是否可改变（是否可写），默认是 `true`。

(3) `enumerable`

是否可遍历，默认是 `true`。如果设为 `false`，会使得某些操作（比如 `for...in` 循环、`Object.keys()`）跳过该属性。

(4) `configurable`

是否可配置，默认是 `true`，如果设为 `false`，将阻止某些操作改写该属性，比如无法删除该属性，也不得改变该属性的属性描述对象（`value` 属性除外）。也就是说，`configurable` 属性控制了属性描述对象的可写性。

(5) `get`

`get` 是一个函数，表示该属性的取值函数（getter），默认为 `undefined`。

(6) `set`

`set` 是一个函数，表示该属性的存值函数（setter），默认为 `undefined`。

## 2. `Object.getOwnPropertyDescriptor()`

`Object.getOwnPropertyDescriptor()` 方法可以获取属性描述对象。它的第一个参数是目标对象，第二个对象是一个属性，对应目标对象的某个属性名。

```
let o = { b: 2 };
Object.getOwnPropertyDescriptor(o, "b") // { configurable: true, enumerable: true,
value: 2, writable: true, }
```

`Object.getOwnPropertyDescriptor()` 方法只能用于对象自身的属性，不能用于继承的属性。

```
let obj = { p: "a" };
Object.getOwnPropertyDescriptor(obj, "toString"); // undefined
```

### 3. `Object.defineProperty()` 和 `Object.defineProperties()`

`Object.defineProperty()` 方法允许通过属性描述对象，定义或修改一个属性，然后返回修改后的对象，它的用法如下。

```
Object.defineProperty(object, propertyName, attributesObject);
```

`Object.defineProperty` 方法接受三个参数，依次如下。

- `object`：属性所在的对象
- `propertyName`：字符串，表示属性名
- `attributesObject`：属性描述对象

定义 `obj.p` 可以写成下面这样。

```
let o = Object.defineProperty({}, "p", { writable: false, value: 4 });

o.p; // 4
o.p = 3;
o.p; // 4
```

`Object.defineProperty()` 方法定义了 `obj.p` 属性。由于属性描述对象的 `writable` 属性为 `false`，所以 `obj.p` 属性不可写。

如果属性已经存在，`Object.defineProperty()` 方法相当于更新该属性的属性描述对象。

如果一次性定义或修改多个属性，可以使用 `Object.defineProperties()` 方法。

```
let o = Object.defineProperties(
  {},
  {
    p1: { value: 1, enumerable: true },
    p2: { value: 2, enumerable: false },
    p3: {
```

```
        get: function () {
            return this.p1 + this.p2;
        },
    },
    }
);

obj.p1; // 1
obj.p2; // 2
obj.p3; // 3

for (let i in o3) {
    console.log(i);
}
// p1
Object.keys(o3); // ["p1"]
Object.getOwnPropertyNames(o3); // (3) ["p1", "p2", "p3"]
```

一旦定义了取值函数 `get`（或存值函数 `set`），就不能将 `writable` 属性设为 `true`，或者同时定义 `value` 属性，否则会报错。

```
let o4 = Object.defineProperty({ a: 1 }, "x", {
    value: 3,
    get: function () {
        return 4;
    },
});
// Uncaught TypeError: Invalid property descriptor. Cannot both specify accessors
// and a value or writable attribute
// 未捕获的类型错误，无效的属性描述符，不能同时指定访问器和值或写属性
```

`Object.defineProperty()` 和 `Object.defineProperties()` 参数里面的属性描述对象，`writable`、`configurable`、`enumerable` 这三个属性的默认值都为 `false`。

```
let o = Object.defineProperty({}, "x", {});
Object.getOwnPropertyDescriptor(o, "x"); // {value: undefined, writable: false,
enumerable: false, configurable: false}
```

## 4. `Object.prototype.propertyIsEnumerable()`

实例对象的 `propertyIsEnumerable()` 方法返回一个布尔值，用来判断某个属性是否可遍历。注意，这个方法只能用于判断对象自身的属性，对于继承的属性一律返回 `false`。

```
let o9 = { x: "a" };
o9.propertyIsEnumerable("x"); // true
o9.propertyIsEnumerable("toString"); // false
```

## 5. 元属性

属性描述对象的各个属性称为“元属性”，因为它们可以看作是控制属性的属性。

### 5.1. value

`value` 属性是目标属性的值。

```
let o = { p: 33 };
Object.getOwnPropertyDescriptor(o, "p").value; // 33
let o2 = Object.defineProperty({}, "x", { value: 22 });
console.log(o2); // {x: 22}
```

### 5.2. writable

`writable` 属性是一个布尔值，决定目标属性的值是否可以更改。

```
let o24 = Object.defineProperty({}, "x", { value: 2, writable: false });
o24.x; // 2
o24.x = 5;
o24.x; // 2
```

严格模式下，对属性 `writable` 为 `false` 的属性，修改其值，会报错。

```
"use strict";
let o = {};

Object.defineProperty(o, "a", {
  value: 37,
  writable: false,
});

o.a = 37; // Uncaught TypeError: Cannot assign to read only property 'a' of object
'#<Object>'
```

### 5.3. enumerable

`enumerable` 返回一个布尔值，表示目标属性是否可以遍历。

如果一个属性的 `enumerable` 为 `false`，下面三个操作不会取到该属性。

- `for...in` 循环
- `Object.keys` 方法
- `JSON.stringify` 方法

`enumerable` 可以用来设置“秘密”属性。

```
let obj = {};  
  
Object.defineProperty(obj, "x", {  
  value: 123,  
  enumerable: false,  
});  
  
obj.x; // 123  
  
for (let key in obj) {  
  console.log(key);  
}  
// undefined  
  
Object.keys(obj); // []  
JSON.stringify(obj); // "{}"
```

## 5.4. configurable

`configurable`（可配置性）返回一个布尔值，决定了是否可以修改属性描述对象。也就是说，`configurable` 为 `false` 时，`value`、`writable`、`enumerable` 和 `configurable` 都不能被修改了。

```
let o28 = Object.defineProperty({}, "x", {  
  value: 23,  
  writable: false,  
  enumerable: false,  
  configurable: false,  
});  
// 下面四个操作全部报错: Uncaught TypeError: Cannot redefine property: x  
Object.defineProperty(o28, "x", { value: 2 });  
Object.defineProperty(o28, "x", { writable: true });  
Object.defineProperty(o28, "x", { enumerable: true });  
Object.defineProperty(o28, "x", { configurable: true });
```

上面代码中，`obj.p` 的 `configurable` 为 `false`。然后，改动 `value`、`writable`、`enumerable`、`configurable`，结果都报错。

`writable` 只有在 `false` 改为 `true` 会报错，`true` 改为 `false` 是允许的。至于 `value`，只要 `writable` 和 `configurable` 有一个为 `true`，就允许改动。

```
let o1 = Object.defineProperty({}, "p", {  
  value: 1,  
  writable: true,  
  configurable: false,  
});
```

```
Object.defineProperty(o1, "p", { value: 2 });
// {p: 2}, 修改成功

let o2 = Object.defineProperty({}, "p", {
  value: 1,
  writable: false,
  configurable: true,
});

Object.defineProperty(o2, "p", { value: 2 });
// {p: 2}, 修改成功
```

可配置性决定了目标属性是否可以被删除（delete）。

```
let o = Object.defineProperties(
  {},
  { a: { value: 1, configurable: false }, b: { value: 2, configurable: true } }
);
delete o.a; // false
delete o.b; // true
o; // {a: 1}
o.a; // 1
o.b; // 2
```

## 6. 存取器（访问器）

除了直接定义以外，属性还可以用存取器（**accessor**）定义。其中，存值函数称为 **setter**，使用属性描述对象的 **set** 属性；取值函数称为 **getter**，使用属性描述对象的 **get** 属性。

```
let o = Object.defineProperty({}, "a", {
  get: function () {
    return "123";
  },
  set: function (p) {
    return "123" + p;
  },
});
o.a; // "123"
o.a = "999"; // "999"
o.a; // "123999"
```

**o.p** 定义了 **get** 和 **set o.p** 取值时，就会调用 **get**；赋值时，就会调用 **set**。

```
// 写法二
let o = {
  get a() {
    return "getter";
  }
}
```

```
    },  
    set a(value) {  
        console.log("setter: " + value);  
    },  
};
```

虽然属性 `a` 的读取和赋值行为是一样的，但是有一些细微的区别。第一种写法，属性 `a` 的 `configurable` 和 `enumerable` 都为 `false`，从而导致属性 `a` 是不可遍历的；第二种写法，属性 `a` 的 `configurable` 和 `enumerable` 都为 `true`，因此属性 `a` 是可遍历的。实际开发中，写法二更常用。

**取值函数 `get` 不能接受参数，存值函数 `set` 只能接受一个参数（即属性的值）。**

## 7. 对象控制状态

有时需要冻结对象的读写状态，防止对象被改变。JavaScript 提供了三种冻结方法，最弱的一种是 `Object.preventExtensions`，其次是 `Object.seal`，最强的是 `Object.freeze`。

### 7.1. Object.freeze()

`Object.freeze` 方法可以使得一个对象无法添加新属性、无法删除旧属性、也无法修改属性的值，使得这个对象变成了常量。

```
let o = { a: 3 };  
Object.freeze(o);  
  
o.a = 4;  
o.a; // 3, 无法修改原属性  
  
o.b = 5;  
o.b; // undefined  
o; // {a: 3}, 无法添加新属性  
  
delete o.a;  
o.a; // 3, 无法删除原属性
```

### 7.2 Object.isFrozen()

`Object.isFrozen()` 检查一个对象是否被冻结（是否被使用了 `Object.freeze()` 方法）。

```
let o1 = { a: 1 };  
let o2 = { a: 2 };  
  
Object.freeze(o1);  
  
Object.isFrozen(o1); // true  
Object.isFrozen(o2); // false
```