Object 对象

目录

- 概述
- Object()
- Object 构造函数
- Object 的静态方法
  - Object.keys()，Object.getOwnPropertyNames()
  - 其他方法
- Object 的实例方法
  - Object.prototype.valueOf()
  - Object.prototype.toString()
  - toString() 的应用：判断数据类型
  - Object.prototype.toLocaleString()
  - Object.prototype.hasOwnProperty()
- 参考链接

概述

JavaScript 原生提供 `Object` 对象（注意起首的 `O` 是大写），本章介绍该对象原生的各种方法。

JavaScript 的所有其他对象都继承自 `Object` 对象，即那些对象都是 `Object` 的实例。

`Object` 对象的原生方法分成两类：`Object` 本身的方法与 `Object` 的实例方法。

（1）`Object` 对象本身的方法

所谓“本身的方法”就是直接定义在 `Object` 对象的方法。

```
Object.print = function (o) { console.log(o) };
```

上面代码中，`print` 方法就是直接定义在 `Object` 对象上。

（2）`Object`的实例方法

所谓实例方法就是定义在`Object`原型对象`Object.prototype`上的方法。它可以被`Object`实例直接使用。

```
Object.prototype.print = function () {
  console.log(this);
};

var obj = new Object();
obj.print() // Object
```

上面代码中，`Object.prototype`定义了一个`print`方法，然后生成一个`Object`的实例`obj`。`obj`直接继承了`Object.prototype`的属性和方法，可以直接使用`obj.print`调用`print`方法。也就是说，`obj`对象的`print`方法实质上就是调用`Object.prototype.print`方法。

关于原型对象`object.prototype`的详细解释，参见《面向对象编程》章节。这里只要知道，凡是定义在`Object.prototype`对象上面的属性和方法，将被所有实例对象共享就可以了。

以下先介绍`Object`作为函数的用法，然后再介绍`Object`对象的原生方法，分成对象自身的方法（又称为“静态方法”）和实例方法两部分。

Object()

`Object`

本身是一个函数，可以当作工具方法使用，将任意值转为对象。这个方法常用于保证某个值一定是对象。

如果参数为空（或者为`undefined`和`null`），`Object()`返回一个空对象。

```
var obj = Object();
// 等同于
var obj = Object(undefined);
var obj = Object(null);

obj instanceof Object // true
```

上面代码的含义，是将`undefined`和`null`转为对象，结果得到了一个空对象`obj`。

`instanceof`运算符用来验证，一个对象是否为指定的构造函数的实例。`obj instanceof Object`返回`true`，就表示`obj`对象是`Object`的实例。

如果参数是原始类型的值，`Object`方法将其转为对应的包装对象的实例（参见《包装对象》一章）。

```
var obj = Object(1);
obj instanceof Object // true
obj instanceof Number // true

var obj = Object('foo');
obj instanceof Object // true
obj instanceof String // true

var obj = Object(true);
obj instanceof Object // true
obj instanceof Boolean // true
```

上面代码中，`Object`函数的参数是各种原始类型的值，转换成对象就是原始类型值对应的包装对象。

如果`Object`方法的参数是一个对象，它总是返回该对象，即不用转换。

```
var arr = [];
var obj = Object(arr); // 返回原数组
obj === arr // true

var value = {};
var obj = Object(value) // 返回原对象
obj === value // true

var fn = function () {};
var obj = Object(fn); // 返回原函数
obj === fn // true
```

利用这一点，可以写一个判断变量是否为对象的函数。

```
function isObject(value) {
  return value === Object(value);
}

isObject([]) // true
isObject(true) // false
```

Object 构造函数

`Object`不仅可以当作工具函数使用，还可以当作构造函数使用，即前面可以使用`new`命令。

`Object`构造函数的首要用途，是直接通过它来生成新对象。

```
var obj = new Object();
```

> 注意，通过`var obj = new Object()`的写法生成新对象，与字面量的写法`var obj = {}`是等价的。或者说，后者只是前者的一种简便写法。

`Object`
构造函数的用法与工具方法很相似，几乎一模一样。使用时，可以接受一个参数，如果该参数是一个对象，则直接返回这个对象；如果是一个原始类型的值，则返回该值对应的包装对象（详见《包装对象》一章）。

```
var o1 = {a: 1};
var o2 = new Object(o1);
o1 === o2 // true

var obj = new Object(123);
obj instanceof Number // true
```

虽然用法相似，但是`Object(value)`与`new Object(value)`两者的语义是不同的，`Object(value)`表示将`value`转成一个对象，`new Object(value)`则表示新生成一个对象，它的值是`value`。

Object 的静态方法

所谓“静态方法”，是指部署在`Object`对象自身的方法。

Object.keys(), Object.getOwnPropertyNames()

`Object.keys`方法和`Object.getOwnPropertyNames`方法都用来遍历对象的属性。

`Object.keys`
方法的参数是一个对象，返回一个数组。该数组的成员都是该对象自身的（而不是继承的）所有属性名。

```
var obj = {
  p1: 123,
```

```
  p2: 456
};

Object.keys(obj) // ["p1", "p2"]
```

`Object.getOwnPropertyNames` `Object.keys`

```
var obj = {
  p1: 123,
  p2: 456
};

Object.getOwnPropertyNames(obj) // ["p1", "p2"]
```

`Object.keys()` `Object.getOwnPropertyNames()`
`Object.keys`
`Object.getOwnPropertyNames`

```
var a = ['Hello', 'World'];

Object.keys(a) // ["0", "1"]
Object.getOwnPropertyNames(a) // ["0", "1", "length"]
```

`length` `Object.getOwnPropertyNames`

JavaScript

```
var obj = {
  p1: 123,
  p2: 456
};

Object.keys(obj).length // 2
Object.getOwnPropertyNames(obj).length // 2
```

`Object.keys`

除了上面提到的两个方法，`Object`还有不少其他静态方法，将在后文逐一详细介绍。

（1）对象属性模型的相关方法

- `Object.getOwnPropertyDescriptor()`：获取某个属性的描述对象。
- `Object.defineProperty()`：通过描述对象，定义某个属性。
- `Object.defineProperties()`：通过描述对象，定义多个属性。

（2）控制对象状态的方法

- `Object.preventExtensions()`：防止对象扩展。
- `Object.isExtensible()`：判断对象是否可扩展。
- `Object.seal()`：禁止对象配置。
- `Object.isSealed()`：判断一个对象是否可配置。
- `Object.freeze()`：冻结一个对象。
- `Object.isFrozen()`：判断一个对象是否被冻结。

（3）原型链相关方法

- `Object.create()`：该方法可以指定原型对象和属性，返回一个新的对象。
- `Object.getPrototypeOf()`：获取对象的`Prototype`对象。

Object 的实例方法

除了静态方法，还有不少方法定义在`Object.prototype`对象。它们称为实例方法，所有`Object`的实例对象都继承了这些方法。

`Object`实例对象的方法，主要有以下六个。

- `Object.prototype.valueOf()`：返回当前对象对应的值。
- `Object.prototype.toString()`：返回当前对象对应的字符串形式。
- `Object.prototype.toLocaleString()`：返回当前对象对应的本地字符串形式。
- `Object.prototype.hasOwnProperty()`：判断某个属性是否为当前对象自身的属性，还是继承自原型对象的属性。
- `Object.prototype.isPrototypeOf()`：判断当前对象是否为另一个对象的原型。
- `Object.prototype.propertyIsEnumerable()`：判断某个属性是否可枚举。

本节介绍前四个方法，另外两个方法将在后文相关章节介绍。

Object.prototype.valueOf()

`valueOf`方法的作用是返回一个对象的“值”，默认情况下返回对象本身。

```
var obj = new Object();
obj.valueOf() === obj // true
```

上面代码比较 `obj.valueOf()` 与 `obj` 本身，两者是一样的。

`valueOf` 方法的主要用途是，JavaScript 自动类型转换时会默认调用这个方法（详见《数据类型的转换》一章）。

```
var obj = new Object();
1 + obj // "1[object Object]"
```

上面代码将对象 `obj` 与数字 `1` 相加，这时 JavaScript 就会默认调用 `valueOf()` 方法，求出 `obj` 的值再与 `1` 相加。所以，如果自定义 `valueOf` 方法，就可以得到想要的结果。

```
var obj = new Object();
obj.valueOf = function () {
  return 2;
};

1 + obj // 3
```

上面代码自定义了 `obj` 对象的 `valueOf` 方法，于是 `1 + obj` 就得到了 `3`。这种方法就相当于用自定义的 `obj.valueOf`，覆盖 `Object.prototype.valueOf`。

Object.prototype.toString()

`toString` 方法的作用是返回一个对象的字符串形式，默认情况下返回类型字符串。

```
var o1 = new Object();
o1.toString() // "[object Object]"

var o2 = {a:1};
o2.toString() // "[object Object]"
```

上面代码表示，对于一个对象调用 `toString` 方法，会返回字符串 `[object Object]`，该字符串说明对象的类型。

字符串 `[object Object]` 本身没有太大的用处，但是通过自定义 `toString` 方法，可以让对象在自动类型转换时，得到想要的字符串形式。

```
var obj = new Object();

obj.toString = function () {
  return 'hello';
};

obj + ' ' + 'world' // "hello world"
```

上面代码表示，当对象用于字符串加法时，会自动调用 `toString` 方法。由于自定义了 `toString` 方法，所以返回字符串 `hello world` 。

数组、字符串、函数、 Date 对象都分别部署了自定义的 `toString` 方法，覆盖了 `Object.prototype.toString` 方法。

```
[1, 2, 3].toString() // "1,2,3"

'123'.toString() // "123"

(function () {
  return 123;
}).toString()
// "function () {
//   return 123;
// }"

(new Date()).toString()
// "Tue May 10 2016 09:11:31 GMT+0800 (CST)"
```

上面代码中，数组、字符串、函数、 Date 对象调用 `toString` 方法，并不会返回 `[object Object]` ，因为它们都自定义了 `toString` 方法，覆盖原始方法。

toString() 的应用：判断数据类型

`Object.prototype.toString` 方法返回对象的类型字符串，因此可以用来判断一个值的类型。

```
var obj = {};
obj.toString() // "[object Object]"
```

上面代码调用空对象的 `toString` 方法，结果返回一个字符串 `[object Object]` ，其中第二个 `Object` 表示该值的构造函数。这是一个十分有用的判断数据类型的方法。

□□□□□□□□□□□□ `toString` □□□□□□ `Object.prototype.toString`
□□□□□□□□□□□□□□□□□□□□□□ `Object.prototype.toString` □□□□□□□□ `call`
□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
Object.prototype.toString.call(value)
```

□□□□□□□ `value` □□□□□ `Object.prototype.toString` □□□

□□□□□□□ `Object.prototype.toString` □□□□□□□□

- □□□□□ `[object Number]`□
- □□□□□□ `[object String]`□
- □□□□□□ `[object Boolean]`□
- undefined□□□ `[object Undefined]`□
- null□□□ `[object Null]`□
- □□□□□ `[object Array]`□
- arguments □□□□□ `[object Arguments]`□
- □□□□□ `[object Function]`□
- Error □□□□□ `[object Error]`□
- Date □□□□□ `[object Date]`□
- RegExp □□□□□ `[object RegExp]`□
- □□□□□□□ `[object Object]`□

□□□□□ `Object.prototype.toString` □□□□□□□□□□□□□□□□□

```
Object.prototype.toString.call(2) // "[object Number]"
Object.prototype.toString.call('') // "[object String]"
Object.prototype.toString.call(true) // "[object Boolean]"
Object.prototype.toString.call(undefined) // "[object Undefined]"
Object.prototype.toString.call(null) // "[object Null]"
Object.prototype.toString.call(Math) // "[object Math]"
Object.prototype.toString.call({}) // "[object Object]"
Object.prototype.toString.call([]) // "[object Array]"
```

□□□□□□□□□□□□□□□ `typeof` □□□□□□□□□□□□□□□□

```
var type = function (o){
  var s = Object.prototype.toString.call(o);
  return s.match(/\[object (.*?)\]/)[1].toLowerCase();
};

type({}); // "object"
type([]); // "array"
```

```
type(5); // "number"
type(null); // "null"
type(); // "undefined"
type(/abcd/); // "regex"
type(new Date()); // "date"
```

□□□□□ `type` □□□□□□□□□□□□□□□□□□□□□□□□□□□□□

```
var type = function (o){
  var s = Object.prototype.toString.call(o);
  return s.match(/\[object (.*?)\]/)[1].toLowerCase();
};

['Null',
 'Undefined',
 'Object',
 'Array',
 'String',
 'Number',
 'Boolean',
 'Function',
 'RegExp'
].forEach(function (t) {
  type['is' + t] = function (o) {
    return type(o) === t.toLowerCase();
  };
});

type.isObject({}) // true
type.isNumber(NaN) // true
type.isRegExp(/abc/) // true
```

Object.prototype.toLocaleString()

`Object.prototype.toLocaleString` □□□ `toString` □□□□□□□□□□□□□□□□□□□□□□□□

```
var obj = {};
obj.toString(obj) // "[object Object]"
obj.toLocaleString(obj) // "[object Object]"
```

这个方法的主要作用是留出一个接口，让各种不同的对象实现自己版本的 `toLocaleString` ，用来返回针对某些地域的特定的值。

```
var person = {
  toString: function () {
    return 'Henry Norman Bethune';
  },
  toLocaleString: function () {
    return '白求恩';
  }
};

person.toString() // Henry Norman Bethune
person.toLocaleString() // 白求恩
```

上面代码中， `toString()` 方法返回对象的一般字符串形式， `toLocaleString()` 方法返回本地的字符串形式。

目前，主要有三个对象自定义了 `toLocaleString` 方法。

- Array.prototype.toLocaleString()
- Number.prototype.toLocaleString()
- Date.prototype.toLocaleString()

举例来说，日期的实例对象的 `toString` 和 `toLocaleString` 返回值就不一样，而且 `toLocaleString` 的返回值跟用户设定的所在地域相关。

```
var date = new Date();
date.toString() // "Tue Jan 01 2018 12:01:33 GMT+0800 (CST)"
date.toLocaleString() // "1/01/2018, 12:01:33 PM"
```

Object.prototype.hasOwnProperty()

`Object.prototype.hasOwnProperty` 方法接受一个字符串作为参数，返回一个布尔值，表示该实例对象自身是否具有该属性。

```
var obj = {
  p: 123
};

obj.hasOwnProperty('p') // true
obj.hasOwnProperty('toString') // false
```

□□□□□□□□ `obj` □□□□ `p` □□□□□□□□ `true` □ `toString` □□□□□□□□□□□□ `false` □

□□□□

- Axel Rauschmayer, Protecting objects in JavaScript
- kangax, Understanding delete
- Jon Bretman, Type Checking in JavaScript
- Cody Lindley, Thinking About ECMAScript 5 Parts
- Bjorn Tipling, Advanced objects in JavaScript
- Javier Márquez, JavaScript properties are enumerable, writable and configurable
- Sella Rafaeli, Native JavaScript Data-Binding: □□□□□□□□ model□ view□□□□□

- Lea Verou, Copying object properties, the robust way

□□□ #2
□ □□□ □□□ 2018-01-16 12:10:44 UTC
□ □□□ □□□ 2025-09-30 12:52:05 UTC