Object 对象

1. Object 对象的原生方法

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

1.1. Object 本身的方法

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

原生的方法就是上面那种，直接定义在 Object 对象上。

1.2. Object 的实例的方法

定义在 Object 原型对象 Object.prototype 上的方法。可以被 Object 实例直接使用。

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

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

2. Object()

Object 本身是一个函数，可以当作工具函数使用，将任意值转换为对象。

如果参数为空（或者是 undefined 或 null），Object() 返回一个空对象。

```
let obj = Object(undefined);
// 等价于
let obj = Object(null);
// 等价于
let obj = Object();

obj instanceof Object; // true
```

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。

同样的，可以验证其他类型的对象是否是它对应的构造函数的实例。

```
Object(0) instanceof Number; // true
Object("") instanceof String; // true
Object(true) instanceof Boolean; // true
Object(false) instanceof Boolean; // true
Object(false) instanceof String; // false
```

其他类型的对象都是 `Object` 对象的实例。

```
Number instanceof Object; // true
String instanceof Object; // true
Boolean instanceof Object; // true
Array instanceof Object; // true
Function instanceof Object; // true
```

3. 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。

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

> 通过 `let obj = new Object()` 的写法生成新对象，与字面量的写法 `let 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。

虽然用法相似，但是 `Object(value)`与 `new Object(value)` 两者的语义不同，**`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`。**

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

4. Object 的静态方法

静态方法就是部署在 `Object` 对象自身的方法。

4.1. Object.keys()

`Object.keys()` 方法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。

```
let o = { a: 1, b: 2 };
let a = Object.keys(o);
console.table(a);
```

(index)	Value
0	"a"
1	"b"

4.2. Object.getOwnPropertyNames()

`Object.getOwnPropertyNames()` 方法与 `Object.keys()` 类似，也是接受一个对象作为参数，返回一个数组，包含了这个对象自身的所有属性名。

```
let o = { a: 2, b: 3 };
let a = Object.getOwnPropertyNames(o);
console.table(a);
```

(index)	Value
0	"a"
1	"b"

对于一般的对象来说，`Object.keys()` 和 `Object.getOwnPropertyNames()` 返回的结果是一样的。只有涉及不可枚举属性时，才会有不一样的结果。

`Object.getOwnPropertyNames()` 方法还返回不可枚举的属性名。几乎总是使用 `Object.keys()` 方法，遍历对象的属性。

```
let a89 = ["x", "y"];
Object.keys(a89); // (2) ["0", "1"]
Object.getOwnPropertyNames(a89); // (3) ["0", "1", "length"]
```

5. Object 的实例方法

除了静态方法，`Object` 还有实例方法，所有 `Object` 实例对象都继承了这些方法。

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

5.1. Object.prototype.valueOf()

`valueOf()` 方法是返回一个对象的值，默认情况下返回对象本身。

```
let o = new Object();
o.valueOf() === o; // true
```

5.2. Object.prototype.toString()

`toString()` 方法的作用是返回一个对象的字符串形式，默认返回类型字符串。

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

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

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。

```
let obj = new Object();
obj.toString = function () {
  return "hello";
};
obj + " " + "world"; // "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(); // "Mon Apr 05 2021 21:50:49 GMT+0800 (中国标准时间)"
```

5.3. `toString()` 的应用

`Object.prototype.toString()` 方法返回对象的类型字符串，用来判断数据类型。

由于实例对象可能会自定义 `toString()` 方法，覆盖掉 `Object.prototype.toString()` 方法，所以为了得到类型字符串，最好直接使用 `Object.prototype.toString()` 方法。通过函数的 `call()` 方法，可以在任意值上调用这个方法，帮助我们判断这个值的类型。

不同数据类型的 `Object.prototype.toString` 方法返回值如下。

```
Object.prototype.toString.call(1) // 数值, '[object Number]'
Object.prototype.toString.call(Infinity) // 数值, '[object Number]'
Object.prototype.toString.call(NaN) // 数值, '[object Number]'

Object.prototype.toString.call('1') // 字符串, '[object String]'

Object.prototype.toString.call(true) // 布尔值, '[object Boolean]'
Object.prototype.toString.call(false) // 布尔值, '[object Boolean]'

Object.prototype.toString.call(undefined)// undefined, '[object Undefined]'

Object.prototype.toString.call(null) // null, '[object Null]'

Object.prototype.toString.call([]) // 数组, '[object Array]'

(function fn() { return Object.prototype.toString.call(arguments); })() //
arguments 对象, '[object Arguments]'

Object.prototype.toString.call(function(){}) // 函数, '[object Function]'

Object.prototype.toString.call(new Error('error')) // Error 对象, '[object
Error]'

Object.prototype.toString.call(new Date()) // Date 对象, '[object Date]'

Object.prototype.toString.call(/^[a-zA-Z1-9]{4,6}$/) // RegExp 对象, '[object
RegExp]'

Object.prototype.toString.call({}) // 其它对象, '[object Object]'
```

5.4. Object.prototype.toLocaleString()

Object.prototype.toLocaleString() 方法与 toString() 的返回结果相同，也是返回一个值的字符串形式。

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。

```
let person = {
  toString: function () {
    return "Donald Trump";
  },
  toLocaleString: function () {
    return "唐纳德·特朗普";
  },
};
person.toString(); // Donald Trump
person.toLocaleString(); // 唐纳德·特朗普
```

主要有三个对象自定义了 `toLocaleString()` 方法。`Array.prototype.toLocaleString()`、`Number.prototype.toLocaleString()`、`Date.prototype.toLocaleString()`。

```
let d = new Date();
d.toString(); // "Mon Apr 05 2021 22:13:23 GMT+0800 (中国标准时间)"
d.toLocaleString(); // "2021/4/5下午10:13:23"
```

5.5. `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。

```
let obj = {
  p: 123,
};
obj.hasOwnProperty("p"); // true
obj.hasOwnProperty("toString"); // false
```

对象 `obj` 自身具有 `p` 属性，所以返回 `true`。`toString` 属性是继承的，所以返回 `false`。

`hasOwnProperty()` 和 `in` 的区别是，前者不包含的继承的属性，后者包含。

```
let o = { a: 1 };
"a" in o; // true
"toString" in o; // true

o.hasOwnProperty("a"); // true
o.hasOwnProperty("toString"); // false
```