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

原生的方法就是直接定义在 Object 对象上。

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

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
```

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. 原型和原型链相关方法

> - `Object.create()` 以一个现有对象作为原型，创建一个新对象。

```
const person = {
  isHuman: false,
  printIntroduction: function () {
    `My name is ${this.name}. Am I human? ${this.isHuman}`;
  },
};
```

```
const me = Object.create(person);

me.name = 'Matthew';    // "name" is a property set on "me", but not on "person"
me.isHuman = true;      // Inherited properties can be overwritten
me.printIntroduction(); // "My name is Matthew. Am I human? true"
```

- `Object.setPrototypeOf()` 将一个指定对象的原型设置为另一个对象或者 null。

```
const obj = {};
const parent = { foo: 'bar' };

obj.foo; // undefined
Object.setPrototypeOf(obj, parent);
obj.foo; // "bar"
```

- `Object.getPrototypeOf()` 返回指定对象的原型。

```
const prototype1 = {};
const object1 = Object.create(prototype1);

Object.getPrototypeOf(object1) === prototype1; // true
```

- `Object.prototype.isPrototypeOf()` 检查一个对象是否存在于另一个对象的原型链中。

```
function Foo() {}
function Bar() {}

Bar.prototype = Object.create(Foo.prototype);

const bar = new Bar();
Foo.prototype.isPrototypeOf(bar); // true
Bar.prototype.isPrototypeOf(bar); // true
```

5. 属性相关方法

- `Object.defineProperty()` 直接在一个对象上定义一个新属性，或修改其现有属性，并返回此对象。

```
const object1 = {};

Object.defineProperty(object1, 'property1', {
  value: 42,
  writable: false,
});
```

```
object1.property1 = 77; // Throws an error in strict mode
object1.property1;      // 42
```

- `Object.defineProperties()` 在一个对象上定义新的属性或修改现有属性，并返回该对象。

```
const object1 = {};
Object.defineProperties(object1, {
  property1: {
    value: 42,
    writable: true,
  },
  property2: {},
});
object1.property1; // 42
```

- `Object.getOwnPropertyDescriptor()` 返回一个对象，该对象描述给定对象上特定属性（即直接存在于对象上而不在对象的原型链中的属性）的配置。返回的对象是可变的，但对其进行更改不会影响原始属性的配置。

```
const object1 = {
  property1: 42,
};

const descriptor1 = Object.getOwnPropertyDescriptor(object1, 'property1');
descriptor1.configurable; // true
descriptor1.value;        // 42
```

- `Object.getOwnPropertyDescriptors()` 返回给定对象的所有自有属性描述符。

```
const object1 = {
  property1: 42,
};

const descriptors1 = Object.getOwnPropertyDescriptors(object1);
descriptors1.property1.writable; // true
descriptors1.property1.value;    // 42
```

- `Object.getOwnPropertyNames()` 返回一个数组，其包含给定对象中所有自有属性（包括不可枚举属性，但不包括使用 symbol 值作为名称的属性）。

```
const object1 = {
  a: 1,
  b: 2,
  c: 3,
```

```
};
Object.getOwnPropertyNames(object1); // Array ["a", "b", "c"]
```

- `Object.getOwnPropertySymbols()` 返回一个包含给定对象所有自有 Symbol 属性的数组。

```
const object1 = {};
const a = Symbol('a');
const b = Symbol.for('b');

object1[a] = 'localSymbol';
object1[b] = 'globalSymbol';

const objectSymbols = Object.getOwnPropertySymbols(object1);
objectSymbols.length; // 2
```

- `Object.prototype.propertyIsEnumerable()` 返回一个布尔值，表示指定的属性是否是对象的可枚举自有属性。

```
const object1 = {};
const array1 = [];
object1.property1 = 42;
array1[0] = 42;

object1.propertyIsEnumerable('property1'); // true
array1.propertyIsEnumerable(0);            // true
array1.propertyIsEnumerable('length');     // false
```

6. 遍历相关方法

- `Object.entries()` 返回一个数组，包含给定对象自有的可枚举字符串键属性的键值对。

```
const object1 = {
  a: 'somestring',
  b: 42,
};
for (const [key, value] of Object.entries(object1)) {
  console.log(`${key}: ${value}`);
}
// "a: somestring"
// "b: 42"
```

- `Object.keys()` 返回一个由给定对象自身的可枚举的字符串键属性名组成的数组。

```
const object1 = {
  a: 'somestring',
```

```
  b: 42,
  c: false,
};
Object.keys(object1); // Array ["a", "b", "c"]
```

- `Object.values()` 返回一个给定对象的自有可枚举字符串键属性值组成的数组。

```
const object1 = {
  a: 'somestring',
  b: 42,
  c: false,
};
Object.values(object1); // Array ["somestring", 42, false]
```

7. 阻止对象扩展相关方法

- `Object.seal()` 密封一个对象。阻止其扩展并且使得现有属性不可配置。

```
const object1 = {
  property1: 42,
};

Object.seal(object1);
object1.property1 = 33;
object1.property1;        // 33

delete object1.property1; // Cannot delete when sealed
object1.property1;        // 33
```

- `Object.isSealed()` 判断一个对象是否被密封。

```
const object1 = {
  property1: 42,
};

Object.isSealed(object1); // false
Object.seal(object1);
Object.isSealed(object1); // true
```

- `Object.freeze()` 可以使一个对象被冻结。冻结对象可以防止扩展，并使现有的属性不可写入和不可配置。冻结一个对象是 JavaScript 提供的最高完整性级别保护措施。

```
const obj = {
  prop: 42,
};
```

```
Object.freeze(obj);
obj.prop = 33; // Throws an error in strict mode
obj.prop;      // 42
```

- `Object.isFrozen()` 判断一个对象是否被冻结。

```
const object1 = {
  property1: 42,
};
Object.isFrozen(object1); // false
Object.freeze(object1);
Object.isFrozen(object1); // true
```

- `Object.preventExtensions()` 防止新属性被添加到对象中（即防止该对象被扩展）。它还可以防止对象的原型被重新指定。只能防止添加自有属性。但其对象类型的原型依然可以添加新的属性。

```
const object1 = {};
Object.preventExtensions(object1);
try {
  Object.defineProperty(object1, 'property1', {
    value: 42,
  });
} catch (e) {
  console.log(e); // TypeError: Cannot define property property1, object is not
extensible
}
```

- `Object.isExtensible()` 判断一个对象是否是可扩展的（是否可以在它上面添加新的属性）。

```
const object1 = {};
Object.isExtensible(object1); // true
Object.preventExtensions(object1);
Object.isExtensible(object1); // false
```

8. 其它方法

- `Object.assign()` 将一个或者多个源对象中所有可枚举的自有属性复制到目标对象，并返回修改后的目标对象。

```
const target = { a: 1, b: 2 };
const source = { b: 4, c: 5 };

const returnedTarget = Object.assign(target, source);
```

```
target;                    // Object { a: 1, b: 4, c: 5 }
returnedTarget === target; // true
```

- `Object.fromEntries()` 将键值对列表转化为一个对象。

```
const entries = new Map([
  ['foo', 'bar'],
  ['baz', 42],
]);
const obj = Object.fromEntries(entries);
obj; // Object { foo: "bar", baz: 42 }
```

- `Object.hasOwn()` 如果指定的对象自身有指定的属性，则静态方法 Object.hasOwn() 返回 true。如果属性是继承的或者不存在，该方法返回 false。

```
const object1 = {
  prop: 'exists',
};
Object.hasOwn(object1, 'prop');                    // true
Object.hasOwn(object1, 'toString');                // false
Object.hasOwn(object1, 'undeclaredPropertyValue'); // false
```

- `Object.is()` 确定两个值是否为相同值。

```
Object.is('1', 1);   // false
Object.is(NaN, NaN); // true
Object.is(-0, 0);    // false
const obj = {};
Object.is(obj, {});  // false
```

- `Object.prototype.toLocaleString()` 回一个表示对象的字符串。该方法旨在由派生对象重写，以达到其特定于语言环境的目的。

```
const number = 123456.789;
number.toLocaleString('de-DE'); // "123.456,789"
```

- `Object.prototype.toString()` 返回一个表示该对象的字符串。该方法旨在重写（自定义）派生类对象的类型转换的逻辑。

```
function Dog(name) {
  this.name = name;
```

```
}
const dog1 = new Dog('Gabby');
Dog.prototype.toString = function dogToString() {
  return `${this.name}`;
};
dog1.toString(); // "Gabby"
```

> - `Object.prototype.valueOf()` 将 this 值转换成对象。该方法旨在被派生对象重写，以实现自定义类型转换逻辑。

```
function MyNumberType(n) {
  this.number = n;
}
MyNumberType.prototype.valueOf = function () {
  return this.number;
};
const object1 = new MyNumberType(4);
object1 + 3; // 7
```