ECMAScript 6 入门

作者：阮一峰

目录

其他

对象的新增方法

上一章 下一章

本章介绍 Object 对象的新增方法。

---

1. Object.is()

ES5 比较两个值是否相等，只有两个运算符：相等运算符（==）和严格相等运算符（===）。它们都有缺点，前者会自动转换数据类型，后者的 NaN 不等于自身，以及 +0 等于 -0 。JavaScript 缺乏一种运算，在所有环境中，只要两个值是一样的，它们就应该相等。

ES6 提出“Same-value equality”（同值相等）算法，用来解决这个问题。 Object.is 就是部署这个算法的新方法。它用来比较两个值是否严格相等，与严格比较运算符（===）的行为基本一致。

```
Object.is('foo', 'foo')
// true
Object.is({}, {})
// false
```

不同之处只有两个：一是 +0 不等于 -0 ，二是 NaN 等于自身。

```
+0 === -0 //true
NaN === NaN // false

Object.is(+0, -0) // false
Object.is(NaN, NaN) // true
```

ES5 可以通过下面的代码，部署 Object.is 。

```
Object.defineProperty(Object, 'is', {
  value: function(x, y) {
    if (x === y) {
      // 针对+0 不等于 -0的情况
      return x !== 0 || 1 / x === 1 / y;
    }
    // 针对NaN的情况
    return x !== x && y !== y;
  },
  configurable: true,
  enumerable: false,
  writable: true
});
```

---

2. Object.assign()

基本用法

Object.assign 方法用于对象的合并，将源对象（sour 可 复制到目标对象（target）。

```
const target = { a: 1 };

const source1 = { b: 2 };
const source2 = { c: 3 };

Object.assign(target, source1, source2);
target // {a:1, b:2, c:3}
```

`Object.assign` 方法的第一个参数是目标对象，后面的参数都是源对象。

注意，如果目标对象与源对象有同名属性，或多个源对象有同名属性，则后面的属性会覆盖前面的属性。

```
const target = { a: 1, b: 1 };

const source1 = { b: 2, c: 2 };
const source2 = { c: 3 };

Object.assign(target, source1, source2);
target // {a:1, b:2, c:3}
```

如果只有一个参数， `Object.assign` 会直接返回该参数。

```
const obj = {a: 1};
Object.assign(obj) === obj // true
```

如果该参数不是对象，则会先转成对象，然后返回。

```
typeof Object.assign(2) // "object"
```

由于 `undefined` 和 `null` 无法转成对象，所以如果它们作为参数，就会报错。

```
Object.assign(undefined) // 报错
Object.assign(null) // 报错
```

如果非对象参数出现在源对象的位置（即非首参数），那么处理规则有所不同。首先，这些参数都会转成对象，如果无法转成对象，就会跳过。这意味着，如果 `undefined` 和 `null` 不在首参数，就不会报错。

```
let obj = {a: 1};
Object.assign(obj, undefined) === obj // true
Object.assign(obj, null) === obj // true
```

其他类型的值（即数值、字符串和布尔值）不在首参数，也不会报错。但是，除了字符串会以数组形式，拷贝入目标对象，其他值都不会产生效果。

```
const v1 = 'abc';
const v2 = true;
const v3 = 10;

const obj = Object.assign({}, v1, v2, v3);
console.log(obj); // { "0": "a", "1": "b", "2": "c" }
```

上面代码中， `v1` 、 `v2` 、 `v3` 分别是字符串、布尔值和数值，结果只有字符串合入目标对象（以字符数组的形式），数值和布尔值都会被忽略。这是因为只有字符串的包装对象，会产生可枚举属性。

```
Object(true) // {[[PrimitiveValue]]: true}
Object(10)  //  {[[PrimitiveValue]]: 10}
```

```
Object('abc') // {0: "a", 1: "b", 2: "c", length: 3, [[PrimitiveValue]]: "abc"}
```

上面代码中，布尔值、数值、字符串分别转成对应的包装对象，可以看到它们的原始值都在包装对象的内部属性 `[[PrimitiveValue]]` 上面，这个属性是不会被 `Object.assign` 拷贝的。只有字符串的包装对象，会产生可枚举的实义属性，那些属性则会被拷贝。

`Object.assign` 拷贝的属性是有限制的，只拷贝源对象的自身属性（不拷贝继承属性），也不拷贝不可枚举的属性（`enumerable: false`）。

```
Object.assign({b: 'c'},
  Object.defineProperty({}, 'invisible', {
    enumerable: false,
    value: 'hello'
  })
)
// { b: 'c' }
```

上面代码中，`Object.assign` 要拷贝的对象只有一个不可枚举属性 `invisible`，这个属性并没有被拷贝进去。

属性名为 Symbol 值的属性，也会被 `Object.assign` 拷贝。

```
Object.assign({ a: 'b' }, { [Symbol('c')]: 'd' })
// { a: 'b', Symbol(c): 'd' }
```

注意点

（1）浅拷贝

`Object.assign` 方法实行的是浅拷贝，而不是深拷贝。也就是说，如果源对象某个属性的值是对象，那么目标对象拷贝得到的是这个对象的引用。

```
const obj1 = {a: {b: 1}};
const obj2 = Object.assign({}, obj1);

obj1.a.b = 2;
obj2.a.b // 2
```

上面代码中，源对象 `obj1` 的 `a` 属性的值是一个对象，`Object.assign` 拷贝得到的是这个对象的引用。这个对象的任何变化，都会反映到目标对象上面。

（2）同名属性的替换

对于这种嵌套的对象，一旦遇到同名属性，`Object.assign` 的处理方法是替换，而不是添加。

```
const target = { a: { b: 'c', d: 'e' } }
const source = { a: { b: 'hello' } }
Object.assign(target, source)
// { a: { b: 'hello' } }
```

上面代码中，`target` 对象的 `a` 属性被 `source` 对象的 `a` 属性整个替换掉了，而不会得到 `{ a: { b: 'hello', d: 'e' } }` 的结果。这通常不是开发者想要的，需要特别小心。

一些函数库提供 `Object.assign` 的定制版本（比如 Lodash 的 `_.defaultsDeep` 方法），可以得到深拷贝的合并。

（3）数组的处理

`Object.assign` 可以用来处理数组，但是会把数组视为对象。

```
Object.assign([1, 2, 3], [4, 5])
// [4, 5, 3]
```

上面代码中， `Object.assign` 把数组视为属性名为 0、1、2 的对象，因此源数组的 0 号属性 `4` 覆盖了目标数组的 0 号属性 `1` 。

（4）取值函数的处理

`Object.assign` 只能进行值的复制，如果要复制的值是一个取值函数，那么将求值后再复制。

```
const source = {
  get foo() { return 1 }
};
const target = {};

Object.assign(target, source)
// { foo: 1 }
```

上面代码中， `source` 对象的 `foo` 属性是一个取值函数， `Object.assign` 不会复制这个取值函数，只会拿到值以后，将这个值复制过去。

常见用途

`Object.assign` 方法有很多用处。

（1）为对象添加属性

```
class Point {
  constructor(x, y) {
    Object.assign(this, {x, y});
  }
}
```

上面方法通过 `Object.assign` 方法，将 `x` 属性和 `y` 属性添加到 `Point` 类的对象实例。

（2）为对象添加方法

```
Object.assign(SomeClass.prototype, {
  someMethod(arg1, arg2) {
    ···
  },
  anotherMethod() {
    ···
  }
});

// 等同于下面的写法
SomeClass.prototype.someMethod = function (arg1, arg2) {
  ···
};
SomeClass.prototype.anotherMethod = function () {
  ···
};
```

上面代码使用了对象属性的简洁表示法，直接将两个函数放在大括号中，再使用 `assign` 方法添加到 `SomeClass.prototype` 之中。

（3）克隆对象

```
function clone(origin) {
  return Object.assign({}, origin);
}
```

上面代码将原始对象拷贝到一个空对象，就得到了原始对象的克隆。

不过，采用这种方法克隆，只能克隆原始对象自身的值，不能克隆它继承的值。如果想要保持继承链，可以采用下面的代码。

```
function clone(origin) {
  let originProto = Object.getPrototypeOf(origin);
  return Object.assign(Object.create(originProto), origin);
}
```

（4）合并多个对象

将多个对象合并到某个对象。

```
const merge =
  (target, ...sources) => Object.assign(target, ...sources);
```

如果希望合并后返回一个新对象，可以改写上面函数，对一个空对象合并。

```
const merge =
  (...sources) => Object.assign({}, ...sources);
```

（5）为属性指定默认值

```
const DEFAULTS = {
  logLevel: 0,
  outputFormat: 'html'
};

function processContent(options) {
  options = Object.assign({}, DEFAULTS, options);
  console.log(options);
  // ...
}
```

上面代码中，`DEFAULTS` 对象是默认值，`options` 对象是用户提供的参数。`Object.assign` 方法将 `DEFAULTS` 和 `options` 合并成一个新对象，如果两者有同名属性，则 `options` 的属性值会覆盖 `DEFAULTS` 的属性值。

注意，由于存在浅拷贝的问题，`DEFAULTS` 对象和 `options` 对象的所有属性的值，最好都是简单类型，不要指向另一个对象。否则，`DEFAULTS` 对象的该属性很可能不起作用。

```
const DEFAULTS = {
  url: {
    host: 'example.com',
    port: 7070
  },
};

processContent({ url: {port: 8000} })
// {
//   url: {port: 8000}
// }
```

上面代码的原意是将 `url.port` 改成 8000， `url.host` 不变。实际结果却是 `options.url` 覆盖掉 `DEFAULTS.url` ，所以 `url.host` 就不存在了。

---

3. Object.getOwnPropertyDescriptors()

ES5 的 `Object.getOwnPropertyDescriptor()` 方法会返回某个对象属性的描述对象（descriptor）。ES2017 引入了 `Object.getOwnPropertyDescriptors()` 方法，返回指定对象所有自身属性（非继承属性）的描述对象。

```
const obj = {
  foo: 123,
  get bar() { return 'abc' }
};

Object.getOwnPropertyDescriptors(obj)
// { foo:
//    { value: 123,
//      writable: true,
//      enumerable: true,
//      configurable: true },
//   bar:
//    { get: [Function: get bar],
//      set: undefined,
//      enumerable: true,
//      configurable: true } }
```

上面代码中， `Object.getOwnPropertyDescriptors()` 方法返回一个对象，所有原对象的属性名都是该对象的属性名，对应的属性值就是该属性的描述对象。

该方法的实现非常容易。

```
function getOwnPropertyDescriptors(obj) {
  const result = {};
  for (let key of Reflect.ownKeys(obj)) {
    result[key] = Object.getOwnPropertyDescriptor(obj, key);
  }
  return result;
}
```

该方法的引入目的，主要是为了解决 `Object.assign()` 无法正确拷贝 `get` 属性和 `set` 属性的问题。

```
const source = {
  set foo(value) {
    console.log(value);
  }
};

const target1 = {};
Object.assign(target1, source);

Object.getOwnPropertyDescriptor(target1, 'foo')
// { value: undefined,
//   writable: true,
//   enumerable: true,
//   configurable: true }
```

上面代码中， source 对象的 foo 属性的值是一个赋值函数， Object.assign 方法将这个属性拷贝给 target1 对象，结果该属性的值变成了 undefined 。这是因为 Object.assign 方法总是拷贝一个属性的值，而不会拷贝它背后的赋值方法或取值方法。

这时， Object.getOwnPropertyDescriptors() 方法配合 Object.defineProperties() 方法，就可以实现正确拷贝。

```
const source = {
  set foo(value) {
    console.log(value);
  }
};

const target2 = {};
Object.defineProperties(target2, Object.getOwnPropertyDescriptors(source));
Object.getOwnPropertyDescriptor(target2, 'foo')
// { get: undefined,
//   set: [Function: set foo],
//   enumerable: true,
//   configurable: true }
```

上面代码中，两个对象合并的逻辑可以写成一个函数。

```
const shallowMerge = (target, source) => Object.defineProperties(
  target,
  Object.getOwnPropertyDescriptors(source)
);
```

Object.getOwnPropertyDescriptors() 方法的另一个用处，是配合 Object.create() 方法，将对象属性克隆到一个新对象。这属于浅拷贝。

```
const clone = Object.create(Object.getPrototypeOf(obj),
  Object.getOwnPropertyDescriptors(obj));

// 或者

const shallowClone = (obj) => Object.create(
  Object.getPrototypeOf(obj),
  Object.getOwnPropertyDescriptors(obj)
);
```

上面代码会克隆对象 obj 。

另外， Object.getOwnPropertyDescriptors() 方法可以实现一个对象继承另一个对象。以前，继承另一个对象，常常写成下面这样。

```
const obj = {
  __proto__: prot,
  foo: 123,
};
```

ES6 规定 __proto__ 只有浏览器要部署，其他环境不用部署。如果去除 __proto__ ，上面代码就要改成下面这样。

```
const obj = Object.create(prot);
obj.foo = 123;

// 或者

const obj = Object.assign(
  Object.create(prot),
  {
    foo: 123,
```

```
  }
);
```

有了 `Object.getOwnPropertyDescriptors()`，我们就有了另一种写法。

```
const obj = Object.create(
  prot,
  Object.getOwnPropertyDescriptors({
    foo: 123,
  })
);
```

`Object.getOwnPropertyDescriptors()` 也可以用来实现 Mixin（混入）模式。

```
let mix = (object) => ({
  with: (...mixins) => mixins.reduce(
    (c, mixin) => Object.create(
      c, Object.getOwnPropertyDescriptors(mixin)
    ), object)
});

// multiple mixins example
let a = {a: 'a'};
let b = {b: 'b'};
let c = {c: 'c'};
let d = mix(c).with(a, b);

d.c // "c"
d.b // "b"
d.a // "a"
```

上面代码返回一个新的对象 `d`，代表了对象 `a` 和 `b` 被混入了对象 `c` 的操作。

出于完整性的考虑，`Object.getOwnPropertyDescriptors()` 进入标准以后，以后还会新增 `Reflect.getOwnPropertyDescriptors()` 方法。

---

4. `__proto__` 属性，Object.setPrototypeOf()，Object.getPrototypeOf()

JavaScript 语言的对象继承是通过原型链实现的。ES6 提供了更多原型对象的操作方法。

`__proto__` 属性

`__proto__` 属性（前后各两个下划线），用来读取或设置当前对象的 `prototype` 对象。目前，所有浏览器（包括 IE11）都部署了这个属性。

```
// es5 的写法
const obj = {
  method: function() { ... }
};
obj.__proto__ = someOtherObj;

// es6 的写法
```

```
var obj = Object.create(someOtherObj);
obj.method = function() { ... };
```

该属性没有写入 ES6 的正文，而是写入了附录，原因是 `__proto__` 前后的双下划线，说明它本质上是一个内部属性，而不是一个正式的对外的 API，只是由于浏览器广泛支持，才被加入了 ES6。标准明确规定，只有浏览器必须部署这个属性，其他运行环境不一定需要部署，而且新的代码最好认为这个属性是不存在的。因此，无论从语义的角度，还是从兼容性的角度，都不要使用这个属性，而是使用下面的 `Object.setPrototypeOf()`（写操作）、`Object.getPrototypeOf()`（读操作）、`Object.create()`（生成操作）代替。

实现上，`__proto__` 调用的是 `Object.prototype.__proto__`，具体实现如下。

```
Object.defineProperty(Object.prototype, '__proto__', {
  get() {
    let _thisObj = Object(this);
    return Object.getPrototypeOf(_thisObj);
  },
  set(proto) {
    if (this === undefined || this === null) {
      throw new TypeError();
    }
    if (!isObject(this)) {
      return undefined;
    }
    if (!isObject(proto)) {
      return undefined;
    }
    let status = Reflect.setPrototypeOf(this, proto);
    if (!status) {
      throw new TypeError();
    }
  },
});

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

如果一个对象本身部署了 `__proto__` 属性，该属性的值就是对象的原型。

```
Object.getPrototypeOf({ __proto__: null })
// null
```

Object.setPrototypeOf()

`Object.setPrototypeOf` 方法的作用与 `__proto__` 相同，用来设置一个对象的 `prototype` 对象，返回参数对象本身。它是 ES6 正式推荐的设置原型对象的方法。

```
// 格式
Object.setPrototypeOf(object, prototype)

// 用法
const o = Object.setPrototypeOf({}, null);
```

该方法等同于下面的函数。

```
function setPrototypeOf(obj, proto) {
  obj.__proto__ = proto;
```

```
  return obj;
}
```

下面是一个例子。

```
let proto = {};
let obj = { x: 10 };
Object.setPrototypeOf(obj, proto);

proto.y = 20;
proto.z = 40;

obj.x // 10
obj.y // 20
obj.z // 40
```

上面代码将 `proto` 对象设为 `obj` 对象的原型，所以从 `obj` 对象可以读取 `proto` 对象的属性。

如果第一个参数不是对象，会自动转为对象。但是由于返回的还是第一个参数，所以这个操作不会产生任何效果。

```
Object.setPrototypeOf(1, {}) === 1 // true
Object.setPrototypeOf('foo', {}) === 'foo' // true
Object.setPrototypeOf(true, {}) === true // true
```

由于 `undefined` 和 `null` 无法转为对象，所以如果第一个参数是 `undefined` 或 `null` ，就会报错。

```
Object.setPrototypeOf(undefined, {})
// TypeError: Object.setPrototypeOf called on null or undefined

Object.setPrototypeOf(null, {})
// TypeError: Object.setPrototypeOf called on null or undefined
```

Object.getPrototypeOf()

该方法与 `Object.setPrototypeOf` 方法配套，用于读取一个对象的原型对象。

```
Object.getPrototypeOf(obj);
```

下面是一个例子。

```
function Rectangle() {
  // ...
}

const rec = new Rectangle();

Object.getPrototypeOf(rec) === Rectangle.prototype
// true

Object.setPrototypeOf(rec, Object.prototype);
Object.getPrototypeOf(rec) === Rectangle.prototype
// false
```

如果参数不是对象，会被自动转为对象。

```
// 等同于 Object.getPrototypeOf(Number(1))
Object.getPrototypeOf(1)
// Number {[[PrimitiveValue]]: 0}

// 等同于 Object.getPrototypeOf(String('foo'))
Object.getPrototypeOf('foo')
// String {length: 0, [[PrimitiveValue]]: ""}

// 等同于 Object.getPrototypeOf(Boolean(true))
Object.getPrototypeOf(true)
// Boolean {[[PrimitiveValue]]: false}

Object.getPrototypeOf(1) === Number.prototype // true
Object.getPrototypeOf('foo') === String.prototype // true
Object.getPrototypeOf(true) === Boolean.prototype // true
```

如果参数是 `undefined` 或 `null`，它们无法转为对象，所以会报错。

```
Object.getPrototypeOf(null)
// TypeError: Cannot convert undefined or null to object

Object.getPrototypeOf(undefined)
// TypeError: Cannot convert undefined or null to object
```

---

5. Object.keys()，Object.values()，Object.entries()

Object.keys()

ES5 引入了 `Object.keys` 方法，返回一个数组，成员是参数对象自身的（不含继承的）所有可遍历（enumerable）属性的键名。

```
var obj = { foo: 'bar', baz: 42 };
Object.keys(obj)
// ["foo", "baz"]
```

ES2017 引入了跟 `Object.keys` 配套的 `Object.values` 和 `Object.entries`，作为遍历一个对象的补充手段，供 `for...of` 循环使用。

```
let {keys, values, entries} = Object;
let obj = { a: 1, b: 2, c: 3 };

for (let key of keys(obj)) {
  console.log(key); // 'a', 'b', 'c'
}

for (let value of values(obj)) {
  console.log(value); // 1, 2, 3
}

for (let [key, value] of entries(obj)) {
  console.log([key, value]); // ['a', 1], ['b', 2], ['c', 3]
}
```

Object.values()

`Object.values` 方法返回一个数组，成员是参数对象自身的（不含继承的）所有可遍历（enumerable）属性的键值。

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

返回数组的成员顺序，与本章的《属性的遍历》部分介绍的排列规则一致。

```
const obj = { 100: 'a', 2: 'b', 7: 'c' };
Object.values(obj)
// ["b", "c", "a"]
```

上面代码中，属性名为数值的属性，是按照数值大小，从小到大遍历的，因此返回的顺序是 `b` 、 `c` 、 `a` 。

`Object.values` 只返回对象自身的可遍历属性。

```
const obj = Object.create({}, {p: {value: 42}});
Object.values(obj) // []
```

上面代码中， `Object.create` 方法的第二个参数添加的对象属性（属性 `p` ），如果不显式声明，默认是不可遍历的，因为 `p` 的属性描述对象的 `enumerable` 默认是 `false` ， `Object.values` 不会返回这个属性。只要把 `enumerable` 改成 `true` ， `Object.values` 就会返回属性 `p` 的值。

```
const obj = Object.create({}, {p:
  {
    value: 42,
    enumerable: true
  }
});
Object.values(obj) // [42]
```

`Object.values` 会过滤属性名为 Symbol 值的属性。

```
Object.values({ [Symbol()]: 123, foo: 'abc' });
// ['abc']
```

如果 `Object.values` 方法的参数是一个字符串，会返回各个字符组成的一个数组。

```
Object.values('foo')
// ['f', 'o', 'o']
```

上面代码中，字符串会先转成一个类似数组的对象。字符串的每个字符，就是该对象的一个属性。因此， `Object.values` 返回每个属性的键值，就是各个字符组成的一个数组。

如果参数不是对象， `Object.values` 会先将其转为对象。由于数值和布尔值的包装对象，都不会为实例添加非继承的属性。所以， `Object.values` 会返回空数组。

```
Object.values(42) // []
Object.values(true) // []
```

Object.entries()

`Object.entries()` 方法返回一个数组，成员是参数对象自身的（不含继承的）所有可遍历（enumerable）属性的键值对数组。

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

除了返回值不一样，该方法的行为与 `Object.values` 基本一致。

如果原对象的属性名是一个 Symbol 值，该属性会被忽略。

```
Object.entries({ [Symbol()]: 123, foo: 'abc' });
// [ [ 'foo', 'abc' ] ]
```

上面代码中，原对象有两个属性， `Object.entries` 只输出属性名非 Symbol 值的属性。将来可能会有 `Reflect.ownEntries()` 方法，返回对象自身的所有属性。

`Object.entries` 的基本用途是遍历对象的属性。

```
let obj = { one: 1, two: 2 };
for (let [k, v] of Object.entries(obj)) {
  console.log(
    `${JSON.stringify(k)}: ${JSON.stringify(v)}`
  );
}
// "one": 1
// "two": 2
```

`Object.entries` 方法的另一个用处是，将对象转为真正的 `Map` 结构。

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

自己实现 `Object.entries` 方法，非常简单。

```
// Generator函数的版本
function* entries(obj) {
  for (let key of Object.keys(obj)) {
    yield [key, obj[key]];
  }
}

// 非Generator函数的版本
function entries(obj) {
  let arr = [];
  for (let key of Object.keys(obj)) {
    arr.push([key, obj[key]]);
  }
  return arr;
}
```


6. Object.fromEntries()

`Object.fromEntries()` 方法是 `Object.entries()` 的逆操作，用于将一个键值对数组转为对象。

```
Object.fromEntries([
  ['foo', 'bar'],
  ['baz', 42]
])
// { foo: "bar", baz: 42 }
```

该方法的主要目的，是将键值对的数据结构还原为对象，因此特别适合将 Map 结构转为对象。

```
// 例一
const entries = new Map([
  ['foo', 'bar'],
  ['baz', 42]
]);

Object.fromEntries(entries)
// { foo: "bar", baz: 42 }

// 例二
const map = new Map().set('foo', true).set('bar', false);
Object.fromEntries(map)
// { foo: true, bar: false }
```

该方法的一个用处是配合 `URLSearchParams` 对象，将查询字符串转为对象。

```
Object.fromEntries(new URLSearchParams('foo=bar&baz=qux'))
// { foo: "bar", baz: "qux" }
```

留言