# 对象的扩展

1. **[属性的简洁表示法](http://es6.ruanyifeng.com/" \l "docs/object#属性的简洁表示法)**
2. **[属性名表达式](http://es6.ruanyifeng.com/" \l "docs/object#属性名表达式)**
3. **[方法的 name 属性](http://es6.ruanyifeng.com/" \l "docs/object#方法的 name 属性)**
4. **[Object.is()](http://es6.ruanyifeng.com/" \l "docs/object#Object.is())**
5. **[Object.assign()](http://es6.ruanyifeng.com/" \l "docs/object#Object.assign())**
6. **[属性的可枚举性和遍历](http://es6.ruanyifeng.com/" \l "docs/object#属性的可枚举性和遍历)**
7. **[Object.getOwnPropertyDescriptors()](http://es6.ruanyifeng.com/" \l "docs/object#Object.getOwnPropertyDescriptors())**
8. **[\_\_proto\_\_属性，Object.setPrototypeOf()，Object.getPrototypeOf()](http://es6.ruanyifeng.com/" \l "docs/object#__proto__属性，Object.setPrototypeOf()，Object.getPrototypeOf())**
9. **[super 关键字](http://es6.ruanyifeng.com/" \l "docs/object#super 关键字)**
10. **[Object.keys()，Object.values()，Object.entries()](http://es6.ruanyifeng.com/" \l "docs/object#Object.keys()，Object.values()，Object.entries())**
11. **[对象的扩展运算符](http://es6.ruanyifeng.com/" \l "docs/object#对象的扩展运算符)**
12. **[Null 传导运算符](http://es6.ruanyifeng.com/" \l "docs/object#Null 传导运算符)**

## 属性的简洁表示法

ES6 允许直接写入变量和函数，作为对象的属性和方法。这样的书写更加简洁。

const foo = 'bar';

const baz = {foo};

baz // {foo: "bar"}

// 等同于const baz = {foo: foo};

上面代码表明，ES6 允许在对象之中，直接写变量。这时，属性名为变量名, 属性值为变量的值。下面是另一个例子。

function f(x, y) {

return {x, y};}

// 等同于function f(x, y) {

return {x: x, y: y};}

f(1, 2) // Object {x: 1, y: 2}

除了属性简写，方法也可以简写。

const o = {

method() {

return "Hello!";

}};

// 等同于

const o = {

method: function() {

return "Hello!";

}};

下面是一个实际的例子。

let birth = '2000/01/01';

const Person = {

name: '张三',

//等同于birth: birth birth,

// 等同于hello: function ()... hello() { console.log('我的名字是', this.name); }

};

这种写法用于函数的返回值，将会非常方便。

function getPoint() {

const x = 1;

const y = 10;

return {x, y};}

getPoint()

// {x:1, y:10}

CommonJS 模块输出一组变量，就非常合适使用简洁写法。

let ms = {};

function getItem (key) {

return key in ms ? ms[key] : null;}

function setItem (key, value) {

ms[key] = value;}

function clear () {

ms = {};}

module.exports = { getItem, setItem, clear };

// 等同于module.exports = {

getItem: getItem,

setItem: setItem,

clear: clear};

属性的赋值器（setter）和取值器（getter），事实上也是采用这种写法。

const cart = {

\_wheels: 4,

get wheels () {

return this.\_wheels;

},

set wheels (value) {

if (value < this.\_wheels) {

throw new Error('数值太小了！');

}

this.\_wheels = value;

}}

注意，简洁写法的属性名总是字符串，这会导致一些看上去比较奇怪的结果。

const obj = {

class () {}};

// 等同于var obj = {

'class': function() {}};

上面代码中，class是字符串，所以不会因为它属于关键字，而导致语法解析报错。

如果某个方法的值是一个 Generator 函数，前面需要加上星号。

const obj = {

\* m() {

yield 'hello world';

}};

## 属性名表达式

JavaScript 定义对象的属性，有两种方法。

// 方法一obj.foo = true;

// 方法二obj['a' + 'bc'] = 123;

上面代码的方法一是直接用标识符作为属性名，方法二是用表达式作为属性名，这时要将表达式放在方括号之内。

但是，如果使用字面量方式定义对象（使用大括号），在 ES5 中只能使用方法一（标识符）定义属性。

var obj = {

foo: true,

abc: 123};

ES6 允许字面量定义对象时，用方法二（表达式）作为对象的属性名，即把表达式放在方括号内。

let propKey = 'foo';

let obj = {

[propKey]: true,

['a' + 'bc']: 123};

下面是另一个例子。

let lastWord = 'last word';

const a = {

'first word': 'hello',

[lastWord]: 'world'};

a['first word'] // "hello"a[lastWord] // "world"a['last word'] // "world"

表达式还可以用于定义方法名。

let obj = {

['h' + 'ello']() {

return 'hi';

}};

obj.hello() // hi

注意，属性名表达式与简洁表示法，不能同时使用，会报错。

// 报错const foo = 'bar';

const bar = 'abc';

const baz = { [foo] };

// 正确const foo = 'bar';

const baz = { [foo]: 'abc'};

注意，属性名表达式如果是一个对象，默认情况下会自动将对象转为字符串[object Object]，这一点要特别小心。

const keyA = {a: 1};

const keyB = {b: 2};

const myObject = {

[keyA]: 'valueA',

[keyB]: 'valueB'};

myObject // Object {[object Object]: "valueB"}

上面代码中，[keyA]和[keyB]得到的都是[object Object]，所以[keyB]会把[keyA]覆盖掉，而myObject最后只有一个[object Object]属性。

## 方法的 name 属性

函数的name属性，返回函数名。对象方法也是函数，因此也有name属性。

const person = {

sayName() {

console.log('hello!');

},};

person.sayName.name // "sayName"

上面代码中，方法的name属性返回函数名（即方法名）。

如果对象的方法使用了取值函数（getter）和存值函数（setter），则name属性不是在该方法上面，而是该方法的属性的描述对象的get和set属性上面，返回值是方法名前加上get和set。

const obj = {

get foo() {},

set foo(x) {}};

obj.foo.name

// TypeError: Cannot read property 'name' of undefined

const descriptor = Object.getOwnPropertyDescriptor(obj, 'foo');

descriptor.get.name // "get foo"descriptor.set.name // "set foo"

有两种特殊情况：bind方法创造的函数，name属性返回bound加上原函数的名字；Function构造函数创造的函数，name属性返回anonymous。

(new Function()).name // "anonymous"var doSomething = function() {

// ...};

doSomething.bind().name // "bound doSomething"

如果对象的方法是一个 Symbol 值，那么name属性返回的是这个 Symbol 值的描述。

const key1 = Symbol('description');

const key2 = Symbol();let obj = {

[key1]() {},

[key2]() {},};

obj[key1].name // "[description]"obj[key2].name // ""

上面代码中，key1对应的 Symbol 值有描述，key2没有。

## 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')

// trueObject.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 //trueNaN === NaN // false

Object.is(+0, -0) // falseObject.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});

## Object.assign()

### 基本用法

Object.assign方法用于对象的合并，将源对象（source）的所有可枚举属性，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 // trueObject.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option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。

## 属性的可枚举性和遍历

### 可枚举性

对象的每个属性都有一个描述对象（Descriptor），用来控制该属性的行为。Object.getOwnPropertyDescriptor方法可以获取该属性的描述对象。

let obj = { foo: 123 };

Object.getOwnPropertyDescriptor(obj, 'foo')

// {// value: 123,// writable: true,// enumerable: true,// configurable: true// }

描述对象的enumerable属性，称为”可枚举性“，如果该属性为false，就表示某些操作会忽略当前属性。

目前，有四个操作会忽略enumerable为false的属性。

* for...in循环：只遍历对象自身的和继承的可枚举的属性。
* Object.keys()：返回对象自身的所有可枚举的属性的键名。
* JSON.stringify()：只串行化对象自身的可枚举的属性。
* Object.assign()： 忽略enumerable为false的属性，只拷贝对象自身的可枚举的属性。

这四个操作之中，前三个是 ES5 就有的，最后一个Object.assign()是 ES6 新增的。其中，只有for...in会返回继承的属性，其他三个方法都会忽略继承的属性，只处理对象自身的属性。实际上，引入“可枚举”（enumerable）这个概念的最初目的，就是让某些属性可以规避掉for...in操作，不然所有内部属性和方法都会被遍历到。比如，对象原型的toString方法，以及数组的length属性，就通过“可枚举性”，从而避免被for...in遍历到。

Object.getOwnPropertyDescriptor(Object.prototype, 'toString').enumerable

// false

Object.getOwnPropertyDescriptor([], 'length').enumerable

// false

上面代码中，toString和length属性的enumerable都是false，因此for...in不会遍历到这两个继承自原型的属性。

另外，ES6 规定，所有 Class 的原型的方法都是不可枚举的。

Object.getOwnPropertyDescriptor(class {foo() {}}.prototype, 'foo').enumerable

// false

总的来说，操作中引入继承的属性会让问题复杂化，大多数时候，我们只关心对象自身的属性。所以，尽量不要用for...in循环，而用Object.keys()代替。

### 属性的遍历

ES6 一共有5种方法可以遍历对象的属性。

****（1）for...in****

for...in循环遍历对象自身的和继承的可枚举属性（不含 Symbol 属性）。

****（2）Object.keys(obj)****

Object.keys返回一个数组，包括对象自身的（不含继承的）所有可枚举属性（不含 Symbol 属性）的键名。

****（3）Object.getOwnPropertyNames(obj)****

Object.getOwnPropertyNames返回一个数组，包含对象自身的所有属性（不含 Symbol 属性，但是包括不可枚举属性）的键名。

****（4）Object.getOwnPropertySymbols(obj)****

Object.getOwnPropertySymbols返回一个数组，包含对象自身的所有 Symbol 属性的键名。

****（5）Reflect.ownKeys(obj)****

Reflect.ownKeys返回一个数组，包含对象自身的所有键名，不管键名是 Symbol 或字符串，也不管是否可枚举。

以上的5种方法遍历对象的键名，都遵守同样的属性遍历的次序规则。

* 首先遍历所有数值键，按照数值升序排列。
* 其次遍历所有字符串键，按照加入时间升序排列。
* 最后遍历所有 Symbol 键，按照加入时间升序排列。

Reflect.ownKeys({ [Symbol()]:0, b:0, 10:0, 2:0, a:0 })

// ['2', '10', 'b', 'a', Symbol()]

上面代码中，Reflect.ownKeys方法返回一个数组，包含了参数对象的所有属性。这个数组的属性次序是这样的，首先是数值属性2和10，其次是字符串属性b和a，最后是 Symbol 属性。

## Object.getOwnPropertyDescriptors()

前面说过，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: 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: 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 examplelet 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。

## \_\_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。

// es6 的写法const obj = {

method: function() { ... }};

obj.\_\_proto\_\_ = someOtherObj;

// es5 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 (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 // 10obj.y // 20obj.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 // trueObject.setPrototypeOf('foo', {}) === 'foo' // trueObject.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 // trueObject.getPrototypeOf('foo') === String.prototype // trueObject.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

## super 关键字

我们知道，this关键字总是指向函数所在的当前对象，ES6 又新增了另一个类似的关键字super，指向当前对象的原型对象。

const proto = {

foo: 'hello'};

const obj = {

find() {

return super.foo;

}};

Object.setPrototypeOf(obj, proto);

obj.find() // "hello"

上面代码中，对象obj的find方法之中，通过super.foo引用了原型对象proto的foo属性。

注意，super关键字表示原型对象时，只能用在对象的方法之中，用在其他地方都会报错。

// 报错const obj = {

foo: super.foo}

// 报错const obj = {

foo: () => super.foo}

// 报错const obj = {

foo: function () {

return super.foo

}}

上面三种super的用法都会报错，因为对于 JavaScript 引擎来说，这里的super都没有用在对象的方法之中。第一种写法是super用在属性里面，第二种和第三种写法是super用在一个函数里面，然后赋值给foo属性。目前，只有对象方法的简写法可以让 JavaScript 引擎确认，定义的是对象的方法。

JavaScript 引擎内部，super.foo等同于Object.getPrototypeOf(this).foo（属性）或Object.getPrototypeOf(this).foo.call(this)（方法）。

const proto = {

x: 'hello',

foo() {

console.log(this.x);

},};

const obj = {

x: 'world',

foo() {

super.foo();

}}

Object.setPrototypeOf(obj, proto);

obj.foo() // "world"

上面代码中，super.foo指向原型对象proto的foo方法，但是绑定的this却还是当前对象obj，因此输出的就是world。

## 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](https://github.com/tc39/proposal-object-values-entries)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;}

## 对象的扩展运算符

《数组的扩展》一章中，已经介绍过扩展运算符（...）。

const [a, ...b] = [1, 2, 3];

a // 1b // [2, 3]

ES2017 将这个运算符[引入](https://github.com/sebmarkbage/ecmascript-rest-spread)了对象。

****（1）解构赋值****

对象的解构赋值用于从一个对象取值，相当于将所有可遍历的、但尚未被读取的属性，分配到指定的对象上面。所有的键和它们的值，都会拷贝到新对象上面。

let { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 };

x // 1y // 2z // { a: 3, b: 4 }

上面代码中，变量z是解构赋值所在的对象。它获取等号右边的所有尚未读取的键（a和b），将它们连同值一起拷贝过来。

由于解构赋值要求等号右边是一个对象，所以如果等号右边是undefined或null，就会报错，因为它们无法转为对象。

let { x, y, ...z } = null; // 运行时错误let { x, y, ...z } = undefined; // 运行时错误

解构赋值必须是最后一个参数，否则会报错。

let { ...x, y, z } = obj; // 句法错误let { x, ...y, ...z } = obj; // 句法错误

上面代码中，解构赋值不是最后一个参数，所以会报错。

注意，解构赋值的拷贝是浅拷贝，即如果一个键的值是复合类型的值（数组、对象、函数）、那么解构赋值拷贝的是这个值的引用，而不是这个值的副本。

let obj = { a: { b: 1 } };let { ...x } = obj;

obj.a.b = 2;

x.a.b // 2

上面代码中，x是解构赋值所在的对象，拷贝了对象obj的a属性。a属性引用了一个对象，修改这个对象的值，会影响到解构赋值对它的引用。

另外，扩展运算符的解构赋值，不能复制继承自原型对象的属性。

let o1 = { a: 1 };let o2 = { b: 2 };

o2.\_\_proto\_\_ = o1;let { ...o3 } = o2;

o3 // { b: 2 }o3.a // undefined

上面代码中，对象o3复制了o2，但是只复制了o2自身的属性，没有复制它的原型对象o1的属性。

下面是另一个例子。

const o = Object.create({ x: 1, y: 2 });

o.z = 3;

let { x, ...{ y, z } } = o;

x // 1y // undefinedz // 3

上面代码中，变量x是单纯的解构赋值，所以可以读取对象o继承的属性；变量y和z是扩展运算符的解构赋值，只能读取对象o自身的属性，所以变量z可以赋值成功，变量y取不到值。

解构赋值的一个用处，是扩展某个函数的参数，引入其他操作。

function baseFunction({ a, b }) {

// ...}function wrapperFunction({ x, y, ...restConfig }) {

// 使用x和y参数进行操作 // 其余参数传给原始函数 return baseFunction(restConfig);}

上面代码中，原始函数baseFunction接受a和b作为参数，函数wrapperFunction在baseFunction的基础上进行了扩展，能够接受多余的参数，并且保留原始函数的行为。

****（2）扩展运算符****

扩展运算符（...）用于取出参数对象的所有可遍历属性，拷贝到当前对象之中。

let z = { a: 3, b: 4 };let n = { ...z };

n // { a: 3, b: 4 }

这等同于使用Object.assign方法。

let aClone = { ...a };

// 等同于let aClone = Object.assign({}, a);

上面的例子只是拷贝了对象实例的属性，如果想完整克隆一个对象，还拷贝对象原型的属性，可以采用下面的写法。

// 写法一const clone1 = {

\_\_proto\_\_: Object.getPrototypeOf(obj),

...obj};

// 写法二const clone2 = Object.assign(

Object.create(Object.getPrototypeOf(obj)),

obj);

// 写法三const clone3 = Object.create(

Object.getPrototypeOf(obj),

Object.getOwnPropertyDescriptors(obj))

上面代码中，写法一的\_\_proto\_\_属性在非浏览器的环境不一定部署，因此推荐使用写法二和写法三。

扩展运算符可以用于合并两个对象。

let ab = { ...a, ...b };

// 等同于let ab = Object.assign({}, a, b);

如果用户自定义的属性，放在扩展运算符后面，则扩展运算符内部的同名属性会被覆盖掉。

let aWithOverrides = { ...a, x: 1, y: 2 };

// 等同于let aWithOverrides = { ...a, ...{ x: 1, y: 2 } };

// 等同于let x = 1, y = 2, aWithOverrides = { ...a, x, y };

// 等同于let aWithOverrides = Object.assign({}, a, { x: 1, y: 2 });

上面代码中，a对象的x属性和y属性，拷贝到新对象后会被覆盖掉。

这用来修改现有对象部分的属性就很方便了。

let newVersion = {

...previousVersion,

name: 'New Name' // Override the name property};

上面代码中，newVersion对象自定义了name属性，其他属性全部复制自previousVersion对象。

如果把自定义属性放在扩展运算符前面，就变成了设置新对象的默认属性值。

let aWithDefaults = { x: 1, y: 2, ...a };

// 等同于let aWithDefaults = Object.assign({}, { x: 1, y: 2 }, a);

// 等同于let aWithDefaults = Object.assign({ x: 1, y: 2 }, a);

与数组的扩展运算符一样，对象的扩展运算符后面可以跟表达式。

const obj = {

...(x > 1 ? {a: 1} : {}),

b: 2,};

如果扩展运算符后面是一个空对象，则没有任何效果。

{...{}, a: 1}

// { a: 1 }

如果扩展运算符的参数是null或undefined，这两个值会被忽略，不会报错。

let emptyObject = { ...null, ...undefined }; // 不报错

扩展运算符的参数对象之中，如果有取值函数get，这个函数是会执行的。

// 并不会抛出错误，因为 x 属性只是被定义，但没执行let aWithXGetter = {

...a,

get x() {

throw new Error('not throw yet');

}};

// 会抛出错误，因为 x 属性被执行了let runtimeError = {

...a,

...{

get x() {

throw new Error('throw now');

}

}};

## Null 传导运算符

编程实务中，如果读取对象内部的某个属性，往往需要判断一下该对象是否存在。比如，要读取message.body.user.firstName，安全的写法是写成下面这样。

const firstName = (message

&& message.body

&& message.body.user

&& message.body.user.firstName) || 'default';

这样的层层判断非常麻烦，因此现在有一个[提案](https://github.com/claudepache/es-optional-chaining)，引入了“Null 传导运算符”（null propagation operator）?.，简化上面的写法。

const firstName = message?.body?.user?.firstName || 'default';

上面代码有三个?.运算符，只要其中一个返回null或undefined，就不再往下运算，而是返回undefined。

“Null 传导运算符”有四种用法。

* obj?.prop // 读取对象属性
* obj?.[expr] // 同上
* func?.(...args) // 函数或对象方法的调用
* new C?.(...args) // 构造函数的调用

传导运算符之所以写成obj?.prop，而不是obj?prop，是为了方便编译器能够区分三元运算符?:（比如obj?prop:123）。

下面是更多的例子。

// 如果 a 是 null 或 undefined, 返回 undefined// 否则返回 a.b.c().da?.b.c().d

// 如果 a 是 null 或 undefined，下面的语句不产生任何效果// 否则执行 a.b = 42a?.b = 42

// 如果 a 是 null 或 undefined，下面的语句不产生任何效果delete a?.b