# Proxy

1. **[概述](http://es6.ruanyifeng.com/" \l "docs/proxy#概述)**
2. **[Proxy 实例的方法](http://es6.ruanyifeng.com/" \l "docs/proxy#Proxy 实例的方法)**
3. **[Proxy.revocable()](http://es6.ruanyifeng.com/" \l "docs/proxy#Proxy.revocable())**
4. **[this 问题](http://es6.ruanyifeng.com/" \l "docs/proxy#this 问题)**
5. **[实例：Web 服务的客户端](http://es6.ruanyifeng.com/" \l "docs/proxy#实例：Web 服务的客户端)**

## 概述

Proxy 用于修改某些操作的默认行为，等同于在语言层面做出修改，所以属于一种“元编程”（meta programming），即对编程语言进行编程。

Proxy 可以理解成，在目标对象之前架设一层“拦截”，外界对该对象的访问，都必须先通过这层拦截，因此提供了一种机制，可以对外界的访问进行过滤和改写。Proxy 这个词的原意是代理，用在这里表示由它来“代理”某些操作，可以译为“代理器”。

var obj = new Proxy({}, {

get: function (target, key, receiver) {

console.log(`getting ${key}!`);

return Reflect.get(target, key, receiver);

},

set: function (target, key, value, receiver) {

console.log(`setting ${key}!`);

return Reflect.set(target, key, value, receiver);

}});

上面代码对一个空对象架设了一层拦截，重定义了属性的读取（get）和设置（set）行为。这里暂时先不解释具体的语法，只看运行结果。对设置了拦截行为的对象obj，去读写它的属性，就会得到下面的结果。

obj.count = 1

// setting count!++obj.count

// getting count!// setting count!// 2

上面代码说明，Proxy 实际上重载（overload）了点运算符，即用自己的定义覆盖了语言的原始定义。

ES6 原生提供 Proxy 构造函数，用来生成 Proxy 实例。

var proxy = new Proxy(target, handler);

Proxy 对象的所有用法，都是上面这种形式，不同的只是handler参数的写法。其中，new Proxy()表示生成一个Proxy实例，target参数表示所要拦截的目标对象，handler参数也是一个对象，用来定制拦截行为。

下面是另一个拦截读取属性行为的例子。

var proxy = new Proxy({}, {

get: function(target, property) {

return 35;

}});

proxy.time // 35proxy.name // 35proxy.title // 35

上面代码中，作为构造函数，Proxy接受两个参数。第一个参数是所要代理的目标对象（上例是一个空对象），即如果没有Proxy的介入，操作原来要访问的就是这个对象；第二个参数是一个配置对象，对于每一个被代理的操作，需要提供一个对应的处理函数，该函数将拦截对应的操作。比如，上面代码中，配置对象有一个get方法，用来拦截对目标对象属性的访问请求。get方法的两个参数分别是目标对象和所要访问的属性。可以看到，由于拦截函数总是返回35，所以访问任何属性都得到35。

注意，要使得Proxy起作用，必须针对Proxy实例（上例是proxy对象）进行操作，而不是针对目标对象（上例是空对象）进行操作。

如果handler没有设置任何拦截，那就等同于直接通向原对象。

var target = {};var handler = {};var proxy = new Proxy(target, handler);

proxy.a = 'b';

target.a // "b"

上面代码中，handler是一个空对象，没有任何拦截效果，访问proxy就等同于访问target。

一个技巧是将 Proxy 对象，设置到object.proxy属性，从而可以在object对象上调用。

var object = { proxy: new Proxy(target, handler) };

Proxy 实例也可以作为其他对象的原型对象。

var proxy = new Proxy({}, {

get: function(target, property) {

return 35;

}});

let obj = Object.create(proxy);

obj.time // 35

上面代码中，proxy对象是obj对象的原型，obj对象本身并没有time属性，所以根据原型链，会在proxy对象上读取该属性，导致被拦截。

同一个拦截器函数，可以设置拦截多个操作。

var handler = {

get: function(target, name) {

if (name === 'prototype') {

return Object.prototype;

}

return 'Hello, ' + name;

},

apply: function(target, thisBinding, args) {

return args[0];

},

construct: function(target, args) {

return {value: args[1]};

}};

var fproxy = new Proxy(function(x, y) {

return x + y;}, handler);

fproxy(1, 2) // 1new fproxy(1, 2) // {value: 2}fproxy.prototype === Object.prototype // truefproxy.foo === "Hello, foo" // true

对于可以设置、但没有设置拦截的操作，则直接落在目标对象上，按照原先的方式产生结果。

下面是 Proxy 支持的拦截操作一览，一共13种。

* ****get(target, propKey, receiver)****：拦截对象属性的读取，比如proxy.foo和proxy['foo']。
* ****set(target, propKey, value, receiver)****：拦截对象属性的设置，比如proxy.foo = v或proxy['foo'] = v，返回一个布尔值。
* ****has(target, propKey)****：拦截propKey in proxy的操作，返回一个布尔值。
* ****deleteProperty(target, propKey)****：拦截delete proxy[propKey]的操作，返回一个布尔值。
* ****ownKeys(target)****：拦截Object.getOwnPropertyNames(proxy)、Object.getOwnPropertySymbols(proxy)、Object.keys(proxy)，返回一个数组。该方法返回目标对象所有自身的属性的属性名，而Object.keys()的返回结果仅包括目标对象自身的可遍历属性。
* ****getOwnPropertyDescriptor(target, propKey)****：拦截Object.getOwnPropertyDescriptor(proxy, propKey)，返回属性的描述对象。
* ****defineProperty(target, propKey, propDesc)****：拦截Object.defineProperty(proxy, propKey, propDesc）、Object.defineProperties(proxy, propDescs)，返回一个布尔值。
* ****preventExtensions(target)****：拦截Object.preventExtensions(proxy)，返回一个布尔值。
* ****getPrototypeOf(target)****：拦截Object.getPrototypeOf(proxy)，返回一个对象。
* ****isExtensible(target)****：拦截Object.isExtensible(proxy)，返回一个布尔值。
* ****setPrototypeOf(target, proto)****：拦截Object.setPrototypeOf(proxy, proto)，返回一个布尔值。如果目标对象是函数，那么还有两种额外操作可以拦截。
* ****apply(target, object, args)****：拦截 Proxy 实例作为函数调用的操作，比如proxy(...args)、proxy.call(object, ...args)、proxy.apply(...)。
* ****construct(target, args)****：拦截 Proxy 实例作为构造函数调用的操作，比如new proxy(...args)。

## Proxy 实例的方法

下面是上面这些拦截方法的详细介绍。

### get()

get方法用于拦截某个属性的读取操作，可以接受三个参数，依次为目标对象、属性名和 proxy 实例本身（即this关键字指向的那个对象），其中最后一个参数可选。

get方法的用法，上文已经有一个例子，下面是另一个拦截读取操作的例子。

var person = {

name: "张三"};

var proxy = new Proxy(person, {

get: function(target, property) {

if (property in target) {

return target[property];

} else {

throw new ReferenceError("Property \"" + property + "\" does not exist.");

}

}});

proxy.name // "张三"proxy.age // 抛出一个错误

上面代码表示，如果访问目标对象不存在的属性，会抛出一个错误。如果没有这个拦截函数，访问不存在的属性，只会返回undefined。

get方法可以继承。

let proto = new Proxy({}, {

get(target, propertyKey, receiver) {

console.log('GET ' + propertyKey);

return target[propertyKey];

}});

let obj = Object.create(proto);

obj.foo // "GET foo"

上面代码中，拦截操作定义在Prototype对象上面，所以如果读取obj对象继承的属性时，拦截会生效。

下面的例子使用get拦截，实现数组读取负数的索引。

function createArray(...elements) {

let handler = {

get(target, propKey, receiver) {

let index = Number(propKey);

if (index < 0) {

propKey = String(target.length + index);

}

return Reflect.get(target, propKey, receiver);

}

};

let target = [];

target.push(...elements);

return new Proxy(target, handler);}

let arr = createArray('a', 'b', 'c');

arr[-1] // c

上面代码中，数组的位置参数是-1，就会输出数组的倒数最后一个成员。

利用 Proxy，可以将读取属性的操作（get），转变为执行某个函数，从而实现属性的链式操作。

var pipe = (function () {

return function (value) {

var funcStack = [];

var oproxy = new Proxy({} , {

get : function (pipeObject, fnName) {

if (fnName === 'get') {

return funcStack.reduce(function (val, fn) {

return fn(val);

},value);

}

funcStack.push(window[fnName]);

return oproxy;

}

});

return oproxy;

}}());

var double = n => n \* 2;var pow = n => n \* n;var reverseInt = n => n.toString().split("").reverse().join("") | 0;

pipe(3).double.pow.reverseInt.get; // 63

上面代码设置 Proxy 以后，达到了将函数名链式使用的效果。

下面的例子则是利用get拦截，实现一个生成各种 DOM 节点的通用函数dom。

const dom = new Proxy({}, {

get(target, property) {

return function(attrs = {}, ...children) {

const el = document.createElement(property);

for (let prop of Object.keys(attrs)) {

el.setAttribute(prop, attrs[prop]);

}

for (let child of children) {

if (typeof child === 'string') {

child = document.createTextNode(child);

}

el.appendChild(child);

}

return el;

}

}});

const el = dom.div({},

'Hello, my name is ',

dom.a({href: '//example.com'}, 'Mark'), '. I like:',

dom.ul({},

dom.li({}, 'The web'),

dom.li({}, 'Food'),

dom.li({}, '…actually that\'s it')

));

document.body.appendChild(el);

下面是一个get方法的第三个参数的例子。

const proxy = new Proxy({}, {

get: function(target, property, receiver) {

return receiver;

}});

proxy.getReceiver === proxy // true

上面代码中，get方法的第三个参数receiver，总是为当前的 Proxy 实例，即get方法内部的this所指向的那个对象。

如果一个属性不可配置（configurable）和不可写（writable），则该属性不能被代理，通过 Proxy 对象访问该属性会报错。

const target = Object.defineProperties({}, {

foo: {

value: 123,

writable: false,

configurable: false

},});

const handler = {

get(target, propKey) {

return 'abc';

}};

const proxy = new Proxy(target, handler);

proxy.foo

// TypeError: Invariant check failed

### set()

set方法用来拦截某个属性的赋值操作，可以接受四个参数，依次为目标对象、属性名、属性值和 Proxy 实例本身，其中最后一个参数可选。

假定Person对象有一个age属性，该属性应该是一个不大于200的整数，那么可以使用Proxy保证age的属性值符合要求。

let validator = {

set: function(obj, prop, value) {

if (prop === 'age') {

if (!Number.isInteger(value)) {

throw new TypeError('The age is not an integer');

}

if (value > 200) {

throw new RangeError('The age seems invalid');

}

}

// 对于age以外的属性，直接保存 obj[prop] = value;

}};

let person = new Proxy({}, validator);

person.age = 100;

person.age // 100person.age = 'young' // 报错person.age = 300 // 报错

上面代码中，由于设置了存值函数set，任何不符合要求的age属性赋值，都会抛出一个错误，这是数据验证的一种实现方法。利用set方法，还可以数据绑定，即每当对象发生变化时，会自动更新 DOM。

有时，我们会在对象上面设置内部属性，属性名的第一个字符使用下划线开头，表示这些属性不应该被外部使用。结合get和set方法，就可以做到防止这些内部属性被外部读写。

const handler = {

get (target, key) {

invariant(key, 'get');

return target[key];

},

set (target, key, value) {

invariant(key, 'set');

target[key] = value;

return true;

}};function invariant (key, action) {

if (key[0] === '\_') {

throw new Error(`Invalid attempt to ${action} private "${key}" property`);

}}

const target = {};

const proxy = new Proxy(target, handler);

proxy.\_prop

// Error: Invalid attempt to get private "\_prop" propertyproxy.\_prop = 'c'

// Error: Invalid attempt to set private "\_prop" property

上面代码中，只要读写的属性名的第一个字符是下划线，一律抛错，从而达到禁止读写内部属性的目的。

下面是set方法第四个参数的例子。

const handler = {

set: function(obj, prop, value, receiver) {

obj[prop] = receiver;

}};

const proxy = new Proxy({}, handler);

proxy.foo = 'bar';

proxy.foo === proxy // true

上面代码中，set方法的第四个参数receiver，总是返回this关键字所指向的那个对象，即proxy实例本身。

注意，如果目标对象自身的某个属性，不可写也不可配置，那么set不得改变这个属性的值，只能返回同样的值，否则报错。

### apply()

apply方法拦截函数的调用、call和apply操作。

apply方法可以接受三个参数，分别是目标对象、目标对象的上下文对象（this）和目标对象的参数数组。

var handler = {

apply (target, ctx, args) {

return Reflect.apply(...arguments);

}};

下面是一个例子。

var target = function () { return 'I am the target'; };var handler = {

apply: function () {

return 'I am the proxy';

}};

var p = new Proxy(target, handler);

p()

// "I am the proxy"

上面代码中，变量p是 Proxy 的实例，当它作为函数调用时（p()），就会被apply方法拦截，返回一个字符串。

下面是另外一个例子。

var twice = {

apply (target, ctx, args) {

return Reflect.apply(...arguments) \* 2;

}};function sum (left, right) {

return left + right;};var proxy = new Proxy(sum, twice);proxy(1, 2) // 6proxy.call(null, 5, 6) // 22proxy.apply(null, [7, 8]) // 30

上面代码中，每当执行proxy函数（直接调用或call和apply调用），就会被apply方法拦截。

另外，直接调用Reflect.apply方法，也会被拦截。

Reflect.apply(proxy, null, [9, 10]) // 38

### has()

has方法用来拦截HasProperty操作，即判断对象是否具有某个属性时，这个方法会生效。典型的操作就是in运算符。

下面的例子使用has方法隐藏某些属性，不被in运算符发现。

var handler = {

has (target, key) {

if (key[0] === '\_') {

return false;

}

return key in target;

}};var target = { \_prop: 'foo', prop: 'foo' };var proxy = new Proxy(target, handler);'\_prop' in proxy // false

上面代码中，如果原对象的属性名的第一个字符是下划线，proxy.has就会返回false，从而不会被in运算符发现。

如果原对象不可配置或者禁止扩展，这时has拦截会报错。

var obj = { a: 10 };

Object.preventExtensions(obj);

var p = new Proxy(obj, {

has: function(target, prop) {

return false;

}});

'a' in p // TypeError is thrown

上面代码中，obj对象禁止扩展，结果使用has拦截就会报错。也就是说，如果某个属性不可配置（或者目标对象不可扩展），则has方法就不得“隐藏”（即返回false）目标对象的该属性。

值得注意的是，has方法拦截的是HasProperty操作，而不是HasOwnProperty操作，即has方法不判断一个属性是对象自身的属性，还是继承的属性。

另外，虽然for...in循环也用到了in运算符，但是has拦截对for...in循环不生效。

let stu1 = {name: '张三', score: 59};let stu2 = {name: '李四', score: 99};

let handler = {

has(target, prop) {

if (prop === 'score' && target[prop] < 60) {

console.log(`${target.name} 不及格`);

return false;

}

return prop in target;

}}

let oproxy1 = new Proxy(stu1, handler);let oproxy2 = new Proxy(stu2, handler);

'score' in oproxy1

// 张三 不及格// false'score' in oproxy2

// truefor (let a in oproxy1) {

console.log(oproxy1[a]);}

// 张三// 59for (let b in oproxy2) {

console.log(oproxy2[b]);}

// 李四// 99

上面代码中，has拦截只对in循环生效，对for...in循环不生效，导致不符合要求的属性没有被排除在for...in循环之外。

### construct()

construct方法用于拦截new命令，下面是拦截对象的写法。

var handler = {

construct (target, args, newTarget) {

return new target(...args);

}};

construct方法可以接受两个参数。

* target: 目标对象
* args：构建函数的参数对象

下面是一个例子。

var p = new Proxy(function () {}, {

construct: function(target, args) {

console.log('called: ' + args.join(', '));

return { value: args[0] \* 10 };

}});

(new p(1)).value

// "called: 1"// 10

construct方法返回的必须是一个对象，否则会报错。

var p = new Proxy(function() {}, {

construct: function(target, argumentsList) {

return 1;

}});

new p() // 报错

### deleteProperty()

deleteProperty方法用于拦截delete操作，如果这个方法抛出错误或者返回false，当前属性就无法被delete命令删除。

var handler = {

deleteProperty (target, key) {

invariant(key, 'delete');

return true;

}};function invariant (key, action) {

if (key[0] === '\_') {

throw new Error(`Invalid attempt to ${action} private "${key}" property`);

}}

var target = { \_prop: 'foo' };var proxy = new Proxy(target, handler);

delete proxy.\_prop

// Error: Invalid attempt to delete private "\_prop" property

上面代码中，deleteProperty方法拦截了delete操作符，删除第一个字符为下划线的属性会报错。

注意，目标对象自身的不可配置（configurable）的属性，不能被deleteProperty方法删除，否则报错。

### defineProperty()

defineProperty方法拦截了Object.defineProperty操作。

var handler = {

defineProperty (target, key, descriptor) {

return false;

}};var target = {};var proxy = new Proxy(target, handler);

proxy.foo = 'bar'

// TypeError: proxy defineProperty handler returned false for property '"foo"'

上面代码中，defineProperty方法返回false，导致添加新属性会抛出错误。

注意，如果目标对象不可扩展（extensible），则defineProperty不能增加目标对象上不存在的属性，否则会报错。另外，如果目标对象的某个属性不可写（writable）或不可配置（configurable），则defineProperty方法不得改变这两个设置。

### getOwnPropertyDescriptor()

getOwnPropertyDescriptor方法拦截Object.getOwnPropertyDescriptor()，返回一个属性描述对象或者undefined。

var handler = {

getOwnPropertyDescriptor (target, key) {

if (key[0] === '\_') {

return;

}

return Object.getOwnPropertyDescriptor(target, key);

}};var target = { \_foo: 'bar', baz: 'tar' };var proxy = new Proxy(target, handler);

Object.getOwnPropertyDescriptor(proxy, 'wat')

// undefinedObject.getOwnPropertyDescriptor(proxy, '\_foo')

// undefinedObject.getOwnPropertyDescriptor(proxy, 'baz')

// { value: 'tar', writable: true, enumerable: true, configurable: true }

上面代码中，handler.getOwnPropertyDescriptor方法对于第一个字符为下划线的属性名会返回undefined。

### getPrototypeOf()

getPrototypeOf方法主要用来拦截获取对象原型。具体来说，拦截下面这些操作。

* Object.prototype.\_\_proto\_\_
* Object.prototype.isPrototypeOf()
* Object.getPrototypeOf()
* Reflect.getPrototypeOf()
* instanceof

下面是一个例子。

var proto = {};var p = new Proxy({}, {

getPrototypeOf(target) {

return proto;

}});

Object.getPrototypeOf(p) === proto // true

上面代码中，getPrototypeOf方法拦截Object.getPrototypeOf()，返回proto对象。

注意，getPrototypeOf方法的返回值必须是对象或者null，否则报错。另外，如果目标对象不可扩展（extensible）， getPrototypeOf方法必须返回目标对象的原型对象。

### isExtensible()

isExtensible方法拦截Object.isExtensible操作。

var p = new Proxy({}, {

isExtensible: function(target) {

console.log("called");

return true;

}});

Object.isExtensible(p)

// "called"// true

上面代码设置了isExtensible方法，在调用Object.isExtensible时会输出called。

注意，该方法只能返回布尔值，否则返回值会被自动转为布尔值。

这个方法有一个强限制，它的返回值必须与目标对象的isExtensible属性保持一致，否则就会抛出错误。

Object.isExtensible(proxy) === Object.isExtensible(target)

下面是一个例子。

var p = new Proxy({}, {

isExtensible: function(target) {

return false;

}});

Object.isExtensible(p) // 报错

### ownKeys()

ownKeys方法用来拦截对象自身属性的读取操作。具体来说，拦截以下操作。

* Object.getOwnPropertyNames()
* Object.getOwnPropertySymbols()
* Object.keys()

下面是拦截Object.keys()的例子。

let target = {

a: 1,

b: 2,

c: 3};

let handler = {

ownKeys(target) {

return ['a'];

}};

let proxy = new Proxy(target, handler);

Object.keys(proxy)

// [ 'a' ]

上面代码拦截了对于target对象的Object.keys()操作，只返回a、b、c三个属性之中的a属性。

下面的例子是拦截第一个字符为下划线的属性名。

let target = {

\_bar: 'foo',

\_prop: 'bar',

prop: 'baz'};

let handler = {

ownKeys (target) {

return Reflect.ownKeys(target).filter(key => key[0] !== '\_');

}};

let proxy = new Proxy(target, handler);for (let key of Object.keys(proxy)) {

console.log(target[key]);}

// "baz"

注意，使用Object.keys方法时，有三类属性会被ownKeys方法自动过滤，不会返回。

* 目标对象上不存在的属性
* 属性名为 Symbol 值
* 不可遍历（enumerable）的属性

let target = {

a: 1,

b: 2,

c: 3,

[Symbol.for('secret')]: '4',};

Object.defineProperty(target, 'key', {

enumerable: false,

configurable: true,

writable: true,

value: 'static'});

let handler = {

ownKeys(target) {

return ['a', 'd', Symbol.for('secret'), 'key'];

}};

let proxy = new Proxy(target, handler);

Object.keys(proxy)

// ['a']

上面代码中，ownKeys方法之中，显式返回不存在的属性（d）、Symbol 值（Symbol.for('secret')）、不可遍历的属性（key），结果都被自动过滤掉。

ownKeys方法还可以拦截Object.getOwnPropertyNames()。

var p = new Proxy({}, {

ownKeys: function(target) {

return ['a', 'b', 'c'];

}});

Object.getOwnPropertyNames(p)

// [ 'a', 'b', 'c' ]

ownKeys方法返回的数组成员，只能是字符串或 Symbol 值。如果有其他类型的值，或者返回的根本不是数组，就会报错。

var obj = {};

var p = new Proxy(obj, {

ownKeys: function(target) {

return [123, true, undefined, null, {}, []];

}});

Object.getOwnPropertyNames(p)

// Uncaught TypeError: 123 is not a valid property name

上面代码中，ownKeys方法虽然返回一个数组，但是每一个数组成员都不是字符串或 Symbol 值，因此就报错了。

如果目标对象自身包含不可配置的属性，则该属性必须被ownKeys方法返回，否则报错。

var obj = {};

Object.defineProperty(obj, 'a', {

configurable: false,

enumerable: true,

value: 10 });

var p = new Proxy(obj, {

ownKeys: function(target) {

return ['b'];

}});

Object.getOwnPropertyNames(p)

// Uncaught TypeError: 'ownKeys' on proxy: trap result did not include 'a'

上面代码中，obj对象的a属性是不可配置的，这时ownKeys方法返回的数组之中，必须包含a，否则会报错。

另外，如果目标对象是不可扩展的（non-extensition），这时ownKeys方法返回的数组之中，必须包含原对象的所有属性，且不能包含多余的属性，否则报错。

var obj = {

a: 1};

Object.preventExtensions(obj);

var p = new Proxy(obj, {

ownKeys: function(target) {

return ['a', 'b'];

}});

Object.getOwnPropertyNames(p)

// Uncaught TypeError: 'ownKeys' on proxy: trap returned extra keys but proxy target is non-extensible

上面代码中，Obj对象是不可扩展的，这时ownKeys方法返回的数组之中，包含了obj对象的多余属性b，所以导致了报错。

### preventExtensions()

preventExtensions方法拦截Object.preventExtensions()。该方法必须返回一个布尔值，否则会被自动转为布尔值。

这个方法有一个限制，只有目标对象不可扩展时（即Object.isExtensible(proxy)为false），proxy.preventExtensions才能返回true，否则会报错。

var p = new Proxy({}, {

preventExtensions: function(target) {

return true;

}});

Object.preventExtensions(p) // 报错

上面代码中，proxy.preventExtensions方法返回true，但这时Object.isExtensible(proxy)会返回true，因此报错。

为了防止出现这个问题，通常要在proxy.preventExtensions方法里面，调用一次Object.preventExtensions。

var p = new Proxy({}, {

preventExtensions: function(target) {

console.log('called');

Object.preventExtensions(target);

return true;

}});

Object.preventExtensions(p)

// "called"// true

### setPrototypeOf()

setPrototypeOf方法主要用来拦截Object.setPrototypeOf方法。

下面是一个例子。

var handler = {

setPrototypeOf (target, proto) {

throw new Error('Changing the prototype is forbidden');

}};var proto = {};var target = function () {};var proxy = new Proxy(target, handler);

Object.setPrototypeOf(proxy, proto);

// Error: Changing the prototype is forbidden

上面代码中，只要修改target的原型对象，就会报错。

注意，该方法只能返回布尔值，否则会被自动转为布尔值。另外，如果目标对象不可扩展（extensible），setPrototypeOf方法不得改变目标对象的原型。

## Proxy.revocable()

Proxy.revocable方法返回一个可取消的 Proxy 实例。

let target = {};let handler = {};

let {proxy, revoke} = Proxy.revocable(target, handler);

proxy.foo = 123;

proxy.foo // 123revoke();

proxy.foo // TypeError: Revoked

Proxy.revocable方法返回一个对象，该对象的proxy属性是Proxy实例，revoke属性是一个函数，可以取消Proxy实例。上面代码中，当执行revoke函数之后，再访问Proxy实例，就会抛出一个错误。

Proxy.revocable的一个使用场景是，目标对象不允许直接访问，必须通过代理访问，一旦访问结束，就收回代理权，不允许再次访问。

## this 问题

虽然 Proxy 可以代理针对目标对象的访问，但它不是目标对象的透明代理，即不做任何拦截的情况下，也无法保证与目标对象的行为一致。主要原因就是在 Proxy 代理的情况下，目标对象内部的this关键字会指向 Proxy 代理。

const target = {

m: function () {

console.log(this === proxy);

}};

const handler = {};

const proxy = new Proxy(target, handler);

target.m() // falseproxy.m() // true

上面代码中，一旦proxy代理target.m，后者内部的this就是指向proxy，而不是target。

下面是一个例子，由于this指向的变化，导致 Proxy 无法代理目标对象。

const \_name = new WeakMap();

class Person {

constructor(name) {

\_name.set(this, name);

}

get name() {

return \_name.get(this);

}}

const jane = new Person('Jane');

jane.name // 'Jane'

const proxy = new Proxy(jane, {});

proxy.name // undefined

上面代码中，目标对象jane的name属性，实际保存在外部WeakMap对象\_name上面，通过this键区分。由于通过proxy.name访问时，this指向proxy，导致无法取到值，所以返回undefined。

此外，有些原生对象的内部属性，只有通过正确的this才能拿到，所以 Proxy 也无法代理这些原生对象的属性。

const target = new Date();

const handler = {};

const proxy = new Proxy(target, handler);

proxy.getDate();

// TypeError: this is not a Date object.

上面代码中，getDate方法只能在Date对象实例上面拿到，如果this不是Date对象实例就会报错。这时，this绑定原始对象，就可以解决这个问题。

const target = new Date('2015-01-01');

const handler = {

get(target, prop) {

if (prop === 'getDate') {

return target.getDate.bind(target);

}

return Reflect.get(target, prop);

}};

const proxy = new Proxy(target, handler);

proxy.getDate() // 1

## 实例：Web 服务的客户端

Proxy 对象可以拦截目标对象的任意属性，这使得它很合适用来写 Web 服务的客户端。

const service = createWebService('http://example.com/data');

service.employees().then(json => {

const employees = JSON.parse(json);

// ···});

上面代码新建了一个 Web 服务的接口，这个接口返回各种数据。Proxy 可以拦截这个对象的任意属性，所以不用为每一种数据写一个适配方法，只要写一个 Proxy 拦截就可以了。

function createWebService(baseUrl) {

return new Proxy({}, {

get(target, propKey, receiver) {

return () => httpGet(baseUrl+'/' + propKey);

}

});}

同理，Proxy 也可以用来实现数据库的 ORM 层。