# Reflect

1. **[概述](http://es6.ruanyifeng.com/" \l "docs/reflect#概述)**
2. **[静态方法](http://es6.ruanyifeng.com/" \l "docs/reflect#静态方法)**
3. **[实例：使用 Proxy 实现观察者模式](http://es6.ruanyifeng.com/" \l "docs/reflect#实例：使用 Proxy 实现观察者模式)**

## 概述

Reflect对象与Proxy对象一样，也是 ES6 为了操作对象而提供的新 API。Reflect对象的设计目的有这样几个。

（1） 将Object对象的一些明显属于语言内部的方法（比如Object.defineProperty），放到Reflect对象上。现阶段，某些方法同时在Object和Reflect对象上部署，未来的新方法将只部署在Reflect对象上。也就是说，从Reflect对象上可以拿到语言内部的方法。

（2） 修改某些Object方法的返回结果，让其变得更合理。比如，Object.defineProperty(obj, name, desc)在无法定义属性时，会抛出一个错误，而Reflect.defineProperty(obj, name, desc)则会返回false。

// 老写法try {

Object.defineProperty(target, property, attributes);

// success} catch (e) {

// failure}

// 新写法if (Reflect.defineProperty(target, property, attributes)) {

// success} else {

// failure}

（3） 让Object操作都变成函数行为。某些Object操作是命令式，比如name in obj和delete obj[name]，而Reflect.has(obj, name)和Reflect.deleteProperty(obj, name)让它们变成了函数行为。

// 老写法'assign' in Object // true

// 新写法Reflect.has(Object, 'assign') // true

（4）Reflect对象的方法与Proxy对象的方法一一对应，只要是Proxy对象的方法，就能在Reflect对象上找到对应的方法。这就让Proxy对象可以方便地调用对应的Reflect方法，完成默认行为，作为修改行为的基础。也就是说，不管Proxy怎么修改默认行为，你总可以在Reflect上获取默认行为。

Proxy(target, {

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

var success = Reflect.set(target,name, value, receiver);

if (success) {

log('property ' + name + ' on ' + target + ' set to ' + value);

}

return success;

}});

上面代码中，Proxy方法拦截target对象的属性赋值行为。它采用Reflect.set方法将值赋值给对象的属性，确保完成原有的行为，然后再部署额外的功能。

下面是另一个例子。

var loggedObj = new Proxy(obj, {

get(target, name) {

console.log('get', target, name);

return Reflect.get(target, name);

},

deleteProperty(target, name) {

console.log('delete' + name);

return Reflect.deleteProperty(target, name);

},

has(target, name) {

console.log('has' + name);

return Reflect.has(target, name);

}});

上面代码中，每一个Proxy对象的拦截操作（get、delete、has），内部都调用对应的Reflect方法，保证原生行为能够正常执行。添加的工作，就是将每一个操作输出一行日志。

有了Reflect对象以后，很多操作会更易读。

// 老写法Function.prototype.apply.call(Math.floor, undefined, [1.75]) // 1

// 新写法Reflect.apply(Math.floor, undefined, [1.75]) // 1

## 静态方法

Reflect对象一共有13个静态方法。

* Reflect.apply(target, thisArg, args)
* Reflect.construct(target, args)
* Reflect.get(target, name, receiver)
* Reflect.set(target, name, value, receiver)
* Reflect.defineProperty(target, name, desc)
* Reflect.deleteProperty(target, name)
* Reflect.has(target, name)
* Reflect.ownKeys(target)
* Reflect.isExtensible(target)
* Reflect.preventExtensions(target)
* Reflect.getOwnPropertyDescriptor(target, name)
* Reflect.getPrototypeOf(target)
* Reflect.setPrototypeOf(target, prototype)

上面这些方法的作用，大部分与Object对象的同名方法的作用都是相同的，而且它与Proxy对象的方法是一一对应的。下面是对它们的解释。

### Reflect.get(target, name, receiver)

Reflect.get方法查找并返回target对象的name属性，如果没有该属性，则返回undefined。

var myObject = {

foo: 1,

bar: 2,

get baz() {

return this.foo + this.bar;

},}

Reflect.get(myObject, 'foo') // 1Reflect.get(myObject, 'bar') // 2Reflect.get(myObject, 'baz') // 3

如果name属性部署了读取函数（getter），则读取函数的this绑定receiver。

var myObject = {

foo: 1,

bar: 2,

get baz() {

return this.foo + this.bar;

},};

var myReceiverObject = {

foo: 4,

bar: 4,};

Reflect.get(myObject, 'baz', myReceiverObject) // 8

如果第一个参数不是对象，Reflect.get方法会报错。

Reflect.get(1, 'foo') // 报错Reflect.get(false, 'foo') // 报错

### Reflect.set(target, name, value, receiver)

Reflect.set方法设置target对象的name属性等于value。

var myObject = {

foo: 1,

set bar(value) {

return this.foo = value;

},}

myObject.foo // 1

Reflect.set(myObject, 'foo', 2);

myObject.foo // 2

Reflect.set(myObject, 'bar', 3)

myObject.foo // 3

如果name属性设置了赋值函数，则赋值函数的this绑定receiver。

var myObject = {

foo: 4,

set bar(value) {

return this.foo = value;

},};

var myReceiverObject = {

foo: 0,};

Reflect.set(myObject, 'bar', 1, myReceiverObject);

myObject.foo // 4myReceiverObject.foo // 1

注意，如果 Proxy 对象和 Reflect 对象联合使用，前者拦截赋值操作，后者完成赋值的默认行为，而且传入了receiver，那么Reflect.set会触发Proxy.defineProperty拦截。

let p = {

a: 'a'};

let handler = {

set(target, key, value, receiver) {

console.log('set');

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

},

defineProperty(target, key, attribute) {

console.log('defineProperty');

Reflect.defineProperty(target, key, attribute);

}};

let obj = new Proxy(p, handler);

obj.a = 'A';

// set// defineProperty

上面代码中，Proxy.set拦截里面使用了Reflect.set，而且传入了receiver，导致触发Proxy.defineProperty拦截。这是因为Proxy.set的receiver参数总是指向当前的 Proxy 实例（即上例的obj），而Reflect.set一旦传入receiver，就会将属性赋值到receiver上面（即obj），导致触发defineProperty拦截。如果Reflect.set没有传入receiver，那么就不会触发defineProperty拦截。

let p = {

a: 'a'};

let handler = {

set(target, key, value, receiver) {

console.log('set');

Reflect.set(target, key, value)

},

defineProperty(target, key, attribute) {

console.log('defineProperty');

Reflect.defineProperty(target, key, attribute);

}};

let obj = new Proxy(p, handler);

obj.a = 'A';

// set

如果第一个参数不是对象，Reflect.set会报错。

Reflect.set(1, 'foo', {}) // 报错Reflect.set(false, 'foo', {}) // 报错

### Reflect.has(obj, name)

Reflect.has方法对应name in obj里面的in运算符。

var myObject = {

foo: 1,};

// 旧写法'foo' in myObject // true

// 新写法Reflect.has(myObject, 'foo') // true

如果第一个参数不是对象，Reflect.has和in运算符都会报错。

### Reflect.deleteProperty(obj, name)

Reflect.deleteProperty方法等同于delete obj[name]，用于删除对象的属性。

const myObj = { foo: 'bar' };

// 旧写法delete myObj.foo;

// 新写法Reflect.deleteProperty(myObj, 'foo');

该方法返回一个布尔值。如果删除成功，或者被删除的属性不存在，返回true；删除失败，被删除的属性依然存在，返回false。

### Reflect.construct(target, args)

Reflect.construct方法等同于new target(...args)，这提供了一种不使用new，来调用构造函数的方法。

function Greeting(name) {

this.name = name;}

// new 的写法const instance = new Greeting('张三');

// Reflect.construct 的写法const instance = Reflect.construct(Greeting, ['张三']);

### Reflect.getPrototypeOf(obj)

Reflect.getPrototypeOf方法用于读取对象的\_\_proto\_\_属性，对应Object.getPrototypeOf(obj)。

const myObj = new FancyThing();

// 旧写法Object.getPrototypeOf(myObj) === FancyThing.prototype;

// 新写法Reflect.getPrototypeOf(myObj) === FancyThing.prototype;

Reflect.getPrototypeOf和Object.getPrototypeOf的一个区别是，如果参数不是对象，Object.getPrototypeOf会将这个参数转为对象，然后再运行，而Reflect.getPrototypeOf会报错。

Object.getPrototypeOf(1) // Number {[[PrimitiveValue]]: 0}Reflect.getPrototypeOf(1) // 报错

### Reflect.setPrototypeOf(obj, newProto)

Reflect.setPrototypeOf方法用于设置对象的\_\_proto\_\_属性，返回第一个参数对象，对应Object.setPrototypeOf(obj, newProto)。

const myObj = new FancyThing();

// 旧写法Object.setPrototypeOf(myObj, OtherThing.prototype);

// 新写法Reflect.setPrototypeOf(myObj, OtherThing.prototype);

如果第一个参数不是对象，Object.setPrototypeOf会返回第一个参数本身，而Reflect.setPrototypeOf会报错。

Object.setPrototypeOf(1, {})

// 1

Reflect.setPrototypeOf(1, {})

// TypeError: Reflect.setPrototypeOf called on non-object

如果第一个参数是undefined或null，Object.setPrototypeOf和Reflect.setPrototypeOf都会报错。

Object.setPrototypeOf(null, {})

// TypeError: Object.setPrototypeOf called on null or undefined

Reflect.setPrototypeOf(null, {})

// TypeError: Reflect.setPrototypeOf called on non-object

### Reflect.apply(func, thisArg, args)

Reflect.apply方法等同于Function.prototype.apply.call(func, thisArg, args)，用于绑定this对象后执行给定函数。

一般来说，如果要绑定一个函数的this对象，可以这样写fn.apply(obj, args)，但是如果函数定义了自己的apply方法，就只能写成Function.prototype.apply.call(fn, obj, args)，采用Reflect对象可以简化这种操作。

const ages = [11, 33, 12, 54, 18, 96];

// 旧写法const youngest = Math.min.apply(Math, ages);

const oldest = Math.max.apply(Math, ages);

const type = Object.prototype.toString.call(youngest);

// 新写法const youngest = Reflect.apply(Math.min, Math, ages);

const oldest = Reflect.apply(Math.max, Math, ages);

const type = Reflect.apply(Object.prototype.toString, youngest, []);

### Reflect.defineProperty(target, propertyKey, attributes)

Reflect.defineProperty方法基本等同于Object.defineProperty，用来为对象定义属性。未来，后者会被逐渐废除，请从现在开始就使用Reflect.defineProperty代替它。

function MyDate() {

/\*…\*/}

// 旧写法Object.defineProperty(MyDate, 'now', {

value: () => Date.now()});

// 新写法Reflect.defineProperty(MyDate, 'now', {

value: () => Date.now()});

如果Reflect.defineProperty的第一个参数不是对象，就会抛出错误，比如Reflect.defineProperty(1, 'foo')。

### Reflect.getOwnPropertyDescriptor(target, propertyKey)

Reflect.getOwnPropertyDescriptor基本等同于Object.getOwnPropertyDescriptor，用于得到指定属性的描述对象，将来会替代掉后者。

var myObject = {};

Object.defineProperty(myObject, 'hidden', {

value: true,

enumerable: false,});

// 旧写法var theDescriptor = Object.getOwnPropertyDescriptor(myObject, 'hidden');

// 新写法var theDescriptor = Reflect.getOwnPropertyDescriptor(myObject, 'hidden');

Reflect.getOwnPropertyDescriptor和Object.getOwnPropertyDescriptor的一个区别是，如果第一个参数不是对象，Object.getOwnPropertyDescriptor(1, 'foo')不报错，返回undefined，而Reflect.getOwnPropertyDescriptor(1, 'foo')会抛出错误，表示参数非法。

### Reflect.isExtensible (target)

Reflect.isExtensible方法对应Object.isExtensible，返回一个布尔值，表示当前对象是否可扩展。

const myObject = {};

// 旧写法Object.isExtensible(myObject) // true

// 新写法Reflect.isExtensible(myObject) // true

如果参数不是对象，Object.isExtensible会返回false，因为非对象本来就是不可扩展的，而Reflect.isExtensible会报错。

Object.isExtensible(1) // falseReflect.isExtensible(1) // 报错

### Reflect.preventExtensions(target)

Reflect.preventExtensions对应Object.preventExtensions方法，用于让一个对象变为不可扩展。它返回一个布尔值，表示是否操作成功。

var myObject = {};

// 旧写法Object.preventExtensions(myObject) // Object {}

// 新写法Reflect.preventExtensions(myObject) // true

如果参数不是对象，Object.preventExtensions在 ES5 环境报错，在 ES6 环境返回传入的参数，而Reflect.preventExtensions会报错。

// ES5 环境Object.preventExtensions(1) // 报错

// ES6 环境Object.preventExtensions(1) // 1

// 新写法Reflect.preventExtensions(1) // 报错

### Reflect.ownKeys (target)

Reflect.ownKeys方法用于返回对象的所有属性，基本等同于Object.getOwnPropertyNames与Object.getOwnPropertySymbols之和。

var myObject = {

foo: 1,

bar: 2,

[Symbol.for('baz')]: 3,

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

// 旧写法Object.getOwnPropertyNames(myObject)

// ['foo', 'bar']

Object.getOwnPropertySymbols(myObject)

//[Symbol(baz), Symbol(bing)]

// 新写法Reflect.ownKeys(myObject)

// ['foo', 'bar', Symbol(baz), Symbol(bing)]

## 实例：使用 Proxy 实现观察者模式

观察者模式（Observer mode）指的是函数自动观察数据对象，一旦对象有变化，函数就会自动执行。

const person = observable({

name: '张三',

age: 20});

function print() {

console.log(`${person.name}, ${person.age}`)}

observe(print);

person.name = '李四';

// 输出// 李四, 20

上面代码中，数据对象person是观察目标，函数print是观察者。一旦数据对象发生变化，print就会自动执行。

下面，使用 Proxy 写一个观察者模式的最简单实现，即实现observable和observe这两个函数。思路是observable函数返回一个原始对象的 Proxy 代理，拦截赋值操作，触发充当观察者的各个函数。

const queuedObservers = new Set();

const observe = fn => queuedObservers.add(fn);

const observable = obj => new Proxy(obj, {set});

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

const result = Reflect.set(target, key, value, receiver);

queuedObservers.forEach(observer => observer());

return result;}

上面代码中，先定义了一个Set集合，所有观察者函数都放进这个集合。然后，observable函数返回原始对象的代理，拦截赋值操作。拦截函数set之中，会自动执行所有观察者。