Reflect

---

1. 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) {
      console.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
```

---

2. 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') // 1
Reflect.get(myObject, 'bar') // 2
Reflect.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 // 4
myReceiverObject.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` 方法用于设置目标对象的原型（prototype），对应 `Object.setPrototypeOf(obj, newProto)` 方法。它返回一个布尔值，表示是否设置成功。

```
const myObj = {};

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

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

myObj.length // 0
```

如果无法设置目标对象的原型（比如，目标对象禁止扩展）， `Reflect.setPrototypeOf` 方法返回 `false` 。

```
Reflect.setPrototypeOf({}, null)
// true
Reflect.setPrototypeOf(Object.freeze({}), null)
// false
```

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')`。

这个方法可以与`Proxy.defineProperty`配合使用。

```
const p = new Proxy({}, {
  defineProperty(target, prop, descriptor) {
    console.log(descriptor);
    return Reflect.defineProperty(target, prop, descriptor);
  }
});

p.foo = 'bar';
// {value: "bar", writable: true, enumerable: true, configurable: true}

p.foo // "bar"
```

上面代码中，`Proxy.defineProperty`对属性赋值设置了拦截，然后使用`Reflect.defineProperty`完成了赋值。

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) // false
Reflect.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)]
```

---

3. 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。