Reflect

1. 概述

Reflect 并非一个构造函数，所以不能通过 new 运算符对其进行调用，或者将 Reflect 对象作为一个函数来调用。Reflect 的所有属性和方法都是静态的，就像 Math 对象。

Reflect 是 JavaScrip 标准内置对象，Proxy 也是。

`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` 对象的设计目的是：

> - （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。

```
// 老写法，Object.defineProperty(target, property, attributes) 在无法定义属性时，会抛出一个错误
try {
  Object.defineProperty(target, property, attributes);
  // success
} catch (e) {
  // failure
}

// 新写法，Reflect.defineProperty(target, property, attributes) 则会返回 false
if (Reflect.defineProperty(target, property, attributes)) {
  // success
} else {
  // failure
}
```

> - （3） 让Object操作都变成函数行为。

```
// 老写法，命令式
'assign' in Object // true
delete Object[name] // true

// 新写法，函数式
Reflect.has(Object, 'assign') // true
Reflect.delete(Object, 'name') // 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，总可以在 `Reflect` 上获取默认行为。

```
Proxy(target, {
  set: function(target, name, value, receiver) {
    let success = Reflect.set(target, name, value, receiver);
    if (success) {
      console.log('property ' + name + ' on ' + target + ' set to ' + value);
    }
    return success;
  }
});
```

上例中，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。

```
let 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);
  }
});
```

上例中，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，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。

> - Reflect.apply(target, thisArg, args) 对一个函数进行调用操作，同时可以传入一个数组作为调用参数。和 Function.prototype.apply() 功能类似。

> - `Reflect.construct(target, args)` 对构造函数进行 new 操作，相当于执行 `new target(...args)`。
> - `Reflect.get(target, name, receiver)` 获取对象身上某个属性的值，类似于 `target[name]`。
> - `Reflect.set(target, name, value, receiver)` 将值分配给属性的函数。返回一个 Boolean，如果更新成功，则返回 true。
> - `Reflect.defineProperty(target, name, desc)` 和 `Object.defineProperty()` 类似。如果设置成功就会返回 true。
> - `Reflect.deleteProperty(target, name)` 作为函数的delete操作符，相当于执行 `delete target[name]`。
> - `Reflect.has(target, name)` 判断一个对象是否存在某个属性，和 `in` 运算符 的功能完全相同。
> - `Reflect.ownKeys(target)` 返回一个包含所有自身属性（不包含继承属性）的数组。(类似于 `Object.keys()`, 但不会受 `enumerable` 影响)。
> - `Reflect.isExtensible(target)` 类似于 `Object.isExtensible()`，对象是否可扩展。
> - `Reflect.preventExtensions(target)` 类似于 `Object.preventExtensions()`。防止该对象被扩展，返回一个Boolean。
> - `Reflect.getOwnPropertyDescriptor(target, name)` 类似于 `Object.getOwnPropertyDescriptor()`。如果对象中存在该属性，则返回对应的属性描述符，否则返回 `undefined`。
> - `Reflect.getPrototypeOf(target)` 类似于 `Object.getPrototypeOf()`，获取对象原型的函数。
> - `Reflect.setPrototypeOf(target, prototype)` 设置对象原型的函数。返回一个 Boolean，如果更新成功，则返回 true。

2.1. 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`。

```
const o = { foo: 1 };
Reflect.get(o, 'foo') // 1
Reflect.get(o, 'bar') // undefined
```

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

```
const o = {
  foo: 1,
  bar: 2,
  get baz() {
    return this.foo + this.bar;
  },
};
const receiver = {
  foo: 4,
  bar: 4,
};
```

```
// o 对象的取值器 baz 中的 this，将绑定 Reflect.get() 方法中第三个参数 receiver
Reflect.get(o, 'baz', receiver); // 8
```

如果第一个参数不是对象，`Reflect.get()` 方法会报错。

```
Reflect.get(false, 'foo');
 // TypeError: Reflect.get called on non-object，调用非对象
```

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

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

```
let o = {
  foo: 1,
  set bar(value) {
    return this.foo = value;
  },
}

o.foo; // 1

Reflect.set(o, 'foo', 2);
o.foo; // 2

Reflect.set(o, 'bar', 3)
o.foo; // 3
```

如果 `name` 属性设置了赋值函数，则赋值函数的 `this` 绑定 `receiver`。

```
let o = {
  foo: 4,
  set bar(value) {
    return this.foo = value;
  },
};
let receiver = {
  foo: 0,
};
// o 对象的存值器 bar 中 this，绑定 Reflect.set 方法第四个参数 receiver
Reflect.set(o, 'bar', 1, receiver);
o.foo; // 4
receiver.foo; // 1
```

如果第一个参数不是对象，`Reflect.set()` 会报错。

```
Reflect.set(false, 'foo', {});
// Reflect.set called on non-object, 调用非对象
```

2.3. Reflect.has(obj, name)

`Reflect.has()` 方法对应 `name in obj` 里面的 `in` 运算符。

```
let o = {
  foo: 1,
};

// 旧写法
'foo' in o; // true

// 新写法
Reflect.has(o, 'foo'); // true
```

如果 `Reflect.has()` 方法的第一个参数不是对象，会报错。

2.4. Reflect.deleteProperty(obj, name)

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

```
const o = { foo: 'bar' };

// 旧写法
delete o.foo;

// 新写法
Reflect.deleteProperty(o, '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`。

如果 `Reflect.deleteProperty()` 方法的第一个参数不是对象，会报错。

```
Reflect.deleteProperty(false, 'foo');
// Reflect.set called on non-object, 调用非对象
```

2.5. 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']);
```

如果 `Reflect.construct()` 方法的第一个参数不是函数，会报错。

2.6. Reflect.getPrototypeOf(obj)

`Reflect.getPrototypeOf()` 方法用于读取对象的 `__proto__` 属性，对应 `Object.getPrototypeOf(obj)`。

```
function Greeting(name) {
  this.name = name;
}
const o = new Greeting();

// 旧写法
Object.getPrototypeOf(o) === Greeting.prototype;
Greeting.prototype === o.__proto__; // 类（构造函数）的显示原型等于实例的隐示原型

// 新写法
Reflect.getPrototypeOf(o) === Greeting.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); // Reflect.getPrototypeOf called on non-object
```

2.7. 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 o = {};

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

// 新写法
```

```
Reflect.setPrototypeOf(o, Array.prototype);
o.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
```

2.8. 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, []);
```

2.9. 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"
```

上例中，`Proxy.defineProperty()` 对属性赋值设置了拦截，然后使用 `Reflect.defineProperty` 完成了赋值。

2.10. 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。

```
let o = {};
Object.defineProperty(o, 'hidden', {
  value: true,
  enumerable: false,
});

// 旧写法
let theDescriptor = Object.getOwnPropertyDescriptor(o, 'hidden');

// 新写法
let theDescriptor = Reflect.getOwnPropertyDescriptor(o, '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。

2.11. 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 o = {};

// 旧写法
Object.isExtensible(o) // true

// 新写法
Reflect.isExtensible(o) // 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
```

2.12. 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。

```
let o = {};

// 旧写法
Object.preventExtensions(o) // Object {}
```

```
// 新写法
Reflect.preventExtensions(o) // 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
```

2.13. 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。

```
let o = {
  foo: 1,
  bar: 2,
  [Symbol.for('baz')]: 3,
  [Symbol.for('bing')]: 4,
};

// 旧写法
Object.getOwnPropertyNames(o)
// ['foo', 'bar']

Object.getOwnPropertySymbols(o)
//[Symbol(baz), Symbol(bing)]

// 新写法
Reflect.ownKeys(o)
// ['foo', 'bar', Symbol(baz), Symbol(bing)]
```

如果 `Reflect.ownKeys()` 方法的第一个参数不是对象，会报错。