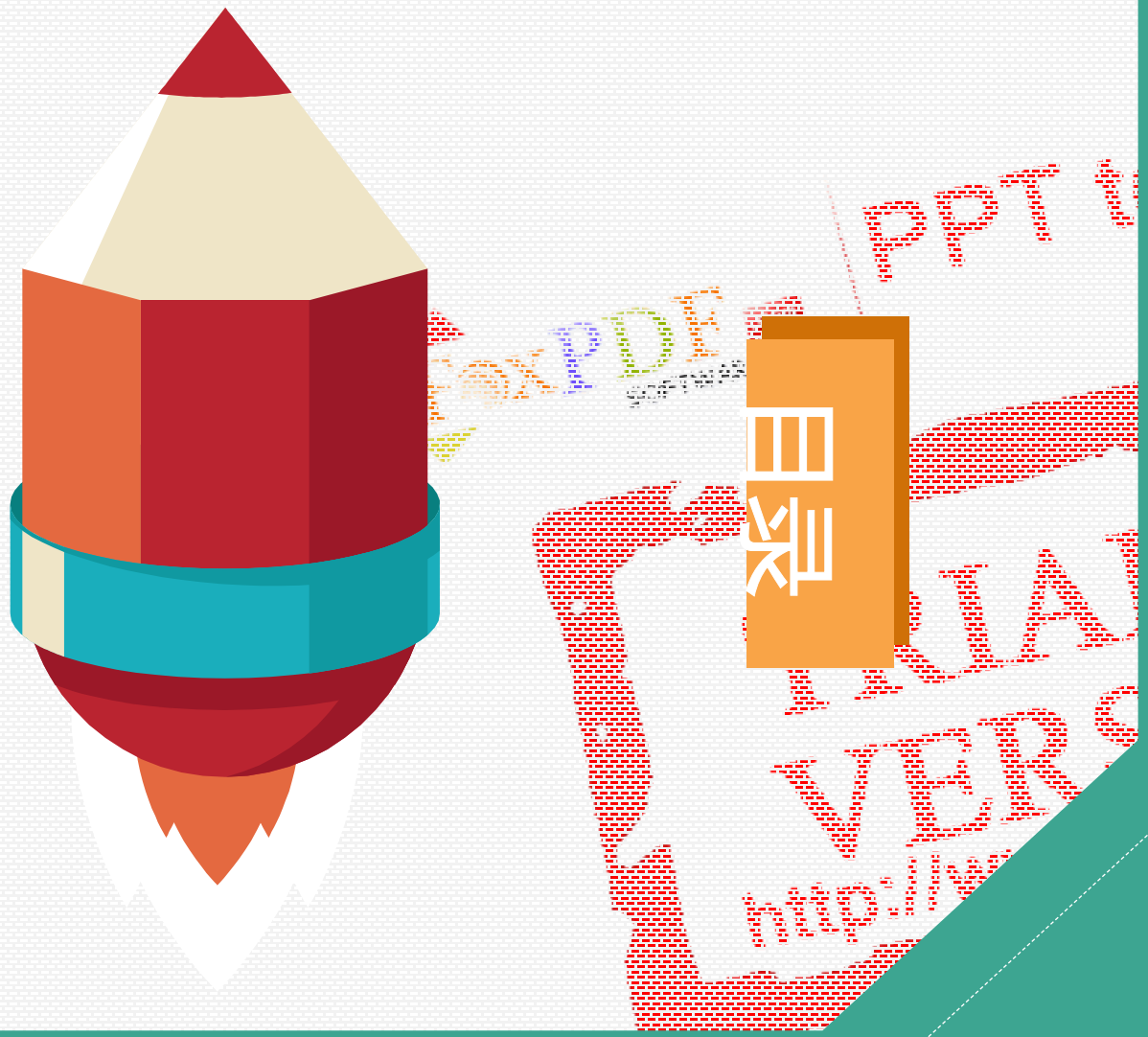

第42课：Reflect
主讲老师：万章



Reflect的静态方法

Reflect的核心总结

设计模式示例

Reflect的静态方法

Reflect的基本概念

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没有构造函数,所以不能用new 的方式来创造新的Reflect对象， Reflect有点像是Math这种全局的对象，给我们提供了一个静态API的接口.

Reflect对象一共有 **13 个静态方法!!!! !!**

Reflect.get(target, name, receiver)方法

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。

如果第一个参数不是对象，Reflect.get方法会报错。

```
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
```

```
let obj = {
    //属性msg部署了getter读取函数
    get msg() {
        //this返回的是Reflect.get的receiver参数对象
        return this.name + this.age
    }
}

let receiver = {
    name: "万章",
    age: "18",
}

let result = Reflect.get(obj, "msg", receiver)

console.log(result) //shen18
```

如果遇到 getter，getter函数里面有this，这个this就是receiver的对象

Reflect.defineProperty(target, propertyKey, attributes)方法

```
function MyDate() {
  /*…*/
}

// 旧写法
Object.defineProperty(MyDate, 'now', {
  value: () => Date.now()
});

// 新写法
Reflect.defineProperty(MyDate, 'now', {
  value: () => Date.now()
});
```

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。

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.set(target, name, value, receiver)方法

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
```

```
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
```

如果name属性设置了set 赋值函数，则赋值函数的this绑定receiver

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

```
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

console.log(obj.a);// A
```

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。

Reflect.has(obj, name)方法

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Reflect.has()方法的第一个参数不是对象，会报错。

Reflect.deleteProperty(obj, name)方法

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。

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

Reflect.construct(target, args)方法

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。

Reflect.getPrototypeOf(obj) 方法

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;
```

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

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。

Reflect.setPrototypeOf(obj, newProto)方法

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
```

Reflect.setPrototypeOf(obj, newProto)方法

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

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。

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
```

```
Object.setPrototypeOf(null, {})
// TypeError: Object.setPrototypeOf called on null or undefined

Reflect.setPrototypeOf(null, {})
// 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。

Reflect.apply(func, thisArg, args)方法

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。

```
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, []);
```

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。

Reflect.getOwnPropertyDescriptor(target, propertyKey)方法

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)方法

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)方法

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)方法

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)]
```

如果Reflect.ownKeys()方法的第一个参数不是对象，会报错。

Reflect的核心总结

Reflect的核心总结

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
}
```

Reflect的核心总结

（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
```

```
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

（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。

Reflect的核心总结

```
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
```

观察者设计模式示例

观察者设计模式示例

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。

（ vue里面有一个数据绑定的功能，该功能的基础设计哲学与观察者模式类似 ）

```
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。

```
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，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。