代理与反射

ES6 新增的代理和反射为开发者提供了拦截并向基本操作嵌入额外行为的能力

给目标对象定义一个关联的代理对象，而这个代理对象可以作为抽象的目标对象来使用。在对目标对象的各种操作影响目标对象之前，可以在代理对象中对这些操作加以控制

代理基础

代理是目标对象的抽象
作用与目标对象，但又独立于目标对象
目标对象可以直接操作，也可以通过代理操作
直接操作会绕过代理施予的行为

创建空代理

只作为一个抽象的目标对象
不拦截任何操作
代理是使用Proxy构造函数创建的。接收两个参数：目标对象和处理程序对象
Proxy.prototype是undefined，所以不能使用instanceof操作符

```
const target = {
    id: 'target'
};
const handler = {};
const proxy = new Proxy(target, handler);
// id 属性会访问同一个值
console.log(target.id); // target
console.log(proxy.id); // target
// 给目标属性赋值会反映在两个对象上
// 因为两个对象访问的是同一个值
target.id = 'foo';
console.log(target.id); // foo
console.log(proxy.id); // foo
// 给代理属性赋值会反映在两个对象上
// 因为这个赋值会转移到目标对象
proxy.id = 'bar';
console.log(target.id); // bar
console.log(proxy.id); // bar
// hasOwnProperty()方法在两个地方
// 都会应用到目标对象
console.log(target.hasOwnProperty('id')); // true
console.log(proxy.hasOwnProperty('id')); // true
// Proxy.prototype 是 undefined
// 因此不能使用 instanceof 操作符
console.log(target instanceof Proxy); // TypeError: Function has non-object prototype
'undefined' in instanceof check
console.log(proxy instanceof Proxy); // TypeError: Function has non-object prototype
'undefined' in instanceof check
// 严格相等可以用来区分代理和目标
```

```
// 不严格相等也能区分
console.log(target === proxy); // false
```

定义捕获器

代理的主要目的就是定义捕获器。
捕获器就是基本操作拦截器
在代理对象调用基本操作，先会调用捕获器函数然后再作用到目标函数

```
const target = {
    foo: 'bar'
};
const handler = {
    // 捕获器在处理程序对象中以方法名为键
    get() {
        return 'handler override';
    }
};
const proxy = new Proxy(target, handler);
console.log(target.foo); // bar
console.log(proxy.foo); // handler override
console.log(target['foo']); // bar
console.log(proxy['foo']); // handler override
console.log(Object.create(target)['foo']); // bar
console.log(Object.create(proxy)['foo']); // handler override
```

捕获器(又称为劫持)有

- handle.apply() 函数调用劫持
- handle.construct() new操作符劫持
- handle.defineProperty() Object.defineProperty调用劫持
- handle.deleteProperty() delete操作符劫持
- handle.get() 获取属性值劫持
- handle.getOwnPropertyDescriptor() Object.getOwnPropertyDescriptor 调用劫持
- handle.getPrototyoeOf() Object.getPrototypeOf调用劫持
- handle.has() in 操作符劫持
- handle.isExtensible() Object.isExtensible调用劫持
- handle.ownKeys() Object.getOwnPropertyNames 和Object.getOwnPropertySymbols调用劫持
- handler.preventExtensions() Object.preventExtensions调用劫持
- handler.set() 设置属性值劫持
- handler.setPrototypeOf() Object.setPrototypeOf调用劫持

捕获器参数和反射API

捕获器会接收到目标对象、要查询的属性和代理对象三个参数
根据这三个参数，可以重建被捕获方法的原始行为

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

```
const handler = {
 get(trapTarget, property, receiver) {
 return trapTarget[property];
 }
};
const proxy = new Proxy(target, handler);
console.log(proxy.foo); // bar
console.log(target.foo); // bar
```

处理程序对象中所有可以捕获的方法，都有对应的反射API方法（Reflect API）

捕获器不变式

使用捕获器几乎可以改变所有基本方法的行为
捕获器不变式：因方法不同而异，防止捕获器定义过于反常的行为

```
const target = {};
Object.defineProperty(target, 'foo', {
    configurable: false,
    writable: false,
    value: 'bar'
});
const handler = {
    get() {
        return 'qux';
    }
};
const proxy = new Proxy(target, handler);
console.log(proxy.foo);
// TypeError
```

可撤销代理

new Proxy() 这种联系会在代理对象的生命周期内一直持续存在
Proxy有revocable()方法，用这个方法创建的代理支持撤销代理对象和目标对象的联系。
撤销函数revoke()，不可逆
撤销函数调用多少次结果都一样。
撤销后再调用就代理会报错 TypeError

```
const target = {
    foo: 'bar'
};
const handler = {
    get() {
        return 'intercepted';
    }
};
const { proxy, revoke } = Proxy.revocable(target, handler);
console.log(proxy.foo); // intercepted
console.log(target.foo); // bar
revoke();
console.log(proxy.foo); // TypeError
```

实用反射API

某些情况应该优先使用反射API

- 反射API与对象API
  使用反射API时要记住：

  - 反射API并不限与捕获处理程序对象
  - 大多数反射API方法再Object类型上有对应的方法。

  通常，Object 上的方法适用于通用程序，而反射方法适用于细粒度的对象控制与操作。

- 状态标记
  很多反射方法可以返回布尔值表示执行是否成功，这就是状态标记
  以下反射方法都会提供状态标记

  - Reflect.defineProperty()
  - Reflect.preventExtensions()
  - Reflect.setPrototypeOf()
  - Reflect.set()
  - Reflect.deleteProperty()

- 用一等函数代替操作符
  以下反射方法提供只有通过操作符才能完成的操作

  - Reflect.get()：可以替代对象属性访问操作符。
  - Reflect.set()：可以替代=赋值操作符。
  - Reflect.has()：可以替代 in 操作符或 with()。
  - Reflect.deleteProperty()：可以替代 delete 操作符。
  - Reflect.construct()：可以替代 new 操作符。

- 安全地应用函数
  通过apply调用函数时，函数可能自己也有apply属性，这样就只能用
  Function.prototype.apply.call(myFunc, thisVal, argumentList)
  用 Reflect.apply视觉上更短。

代理另一个代理

```
const target = {
    foo: 'bar'
};
const firstProxy = new Proxy(target, {
    get() {
        console.log('first proxy');
        return Reflect.get(...arguments);
    }
});
const secondProxy = new Proxy(firstProxy, {
    get() {
        console.log('second proxy');
        return Reflect.get(...arguments);
    }
```

```
});
console.log(secondProxy.foo);
   // second proxy
   // first proxy
   // bar
```

代理地问题与不足

某些情况下代理不能与ES的机制很好的协同

- 代理中的this
  代理中的this与目标对象的this不同

```
const wm = new WeakMap();
class User {
    constructor(userId) {
        wm.set(this, userId);
    }
    set id(userId) {
        wm.set(this, userId);
    }
    get id() {
        return wm.get(this);
    }
}
const user = new User(123);
console.log(user.id); // 123
const userInstanceProxy = new Proxy(user, {});
console.log(userInstanceProxy.id); // undefined
```

- 代理与内部槽位
  有些内置类型可能依赖代理无法控制的机制，会让代理调用某些方法出错
  Date依赖this上的内部槽位[[NumberDate]]，且不能通过普通的get(),set()访问。代理对象上没有[[NumberDate]]
  总结：有些对象的内部属性只能通过正确的this拿到，所以不能用Proxy去代理

```
const target = new Date();
const proxy = new Proxy(target, {});
console.log(proxy instanceof Date); // true
proxy.getDate(); // TypeError: 'this' is not a Date object
```