第9章 代理与反射

本章内容

- ❑ 代理基础
- ❑ 代码捕获器与反射方法
- ❑ 代理模式

ECMAScript 6 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。具体地说，可以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。

对刚刚接触这个主题的开发者而言，代理是一个比较模糊的概念，而且还夹杂着很多新术语。其实只要看几个例子，就很容易理解了。

> **注意** 在 ES6 之前，ECMAScript 中并没有类似代理的特性。由于代理是一种新的基础性语言能力，很多转译程序都不能把代理行为转换为之前的 ECMAScript 代码，因为代理的行为实际上是无可替代的。为此，代理和反射只在百分之百支持它们的平台上有用。可以检测代理是否存在，不存在则提供后备代码。不过这会导致代码冗余，因此并不推荐。

9.1 代理基础

正如本章开头所介绍的，代理是目标对象的抽象。从很多方面看，代理类似 C++指针，因为它可以用作目标对象的替身，但又完全独立于目标对象。目标对象既可以直接被操作，也可以通过代理来操作。但直接操作会绕过代理施予的行为。

> **注意** ECMAScript 代理与 C++指针有重大区别，后面会再讨论。不过作为一种有助于理解的类比，指针在概念上还是比较合适的结构。

9.1.1 创建空代理

最简单的代理是空代理，即除了作为一个抽象的目标对象，什么也不做。默认情况下，在代理对象上执行的所有操作都会无障碍地传播到目标对象。因此，在任何可以使用目标对象的地方，都可以通过同样的方式来使用与之关联的代理对象。

代理是使用 `Proxy` 构造函数创建的。这个构造函数接收两个参数：目标对象和处理程序对象。缺少其中任何一个参数都会抛出 `TypeError`。要创建空代理，可以传一个简单的对象字面量作为处理程

序对象，从而让所有操作畅通无阻地抵达目标对象。

如下面的代码所示，在代理对象上执行的任何操作实际上都会应用到目标对象。唯一可感知的不同就是代码中操作的是代理对象。

```
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
console.log(target === proxy); // false
```

9.1.2 定义捕获器

使用代理的主要目的是可以定义**捕获器**（trap）。捕获器就是在处理程序对象中定义的“基本操作的拦截器”。每个处理程序对象可以包含零个或多个捕获器，每个捕获器都对应一种基本操作，可以直接或间接在代理对象上调用。每次在代理对象上调用这些基本操作时，代理可以在这些操作传播到目标对象之前先调用捕获器函数，从而拦截并修改相应的行为。

> **注意** 捕获器（trap）是从操作系统中借用的概念。在操作系统中，捕获器是程序流中的一个同步中断，可以暂停程序流，转而执行一段子例程，之后再返回原始程序流。

例如，可以定义一个 get()捕获器，在 ECMAScript 操作以某种形式调用 get()时触发。下面的例子定义了一个 get()捕获器：

```
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);
```

这样，当通过代理对象执行 get()操作时，就会触发定义的 get()捕获器。当然，get()不是 ECMAScript 对象可以调用的方法。这个操作在 JavaScript 代码中可以通过多种形式触发并被 get()捕获器拦截到。proxy[property]、proxy.property 或 Object.create(proxy)[property]等操作都会触发基本的 get()操作以获取属性。因此所有这些操作只要发生在代理对象上，就会触发 get()捕获器。注意，只有在代理对象上执行这些操作才会触发捕获器。在目标对象上执行这些操作仍然会产生正常的行为。

```
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
```

9.1.3　捕获器参数和反射 API

所有捕获器都可以访问相应的参数，基于这些参数可以重建被捕获方法的原始行为。比如，get()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。

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

const handler = {
```

```
  get(trapTarget, property, receiver) {
    console.log(trapTarget === target);
    console.log(property);
    console.log(receiver === proxy);
  }
};

const proxy = new Proxy(target, handler);

proxy.foo;
// true
// foo
// true
```

有了这些参数，就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
```

所有捕获器都可以基于自己的参数重建原始操作，但并非所有捕获器行为都像 `get()`那么简单。因此，通过手动写码如法炮制的想法是不现实的。实际上，开发者并不需要手动重建原始行为，而是可以通过调用全局 `Reflect` 对象上（封装了原始行为）的同名方法来轻松重建。

处理程序对象中所有可以捕获的方法都有对应的反射（Reflect）API 方法。这些方法与捕获器拦截的方法具有相同的名称和函数签名，而且也具有与被拦截方法相同的行为。因此，使用反射 API 也可以像下面这样定义出空代理对象：

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

const handler = {
  get() {
    return Reflect.get(...arguments);
  }
};

const proxy = new Proxy(target, handler);

console.log(proxy.foo);   // bar
console.log(target.foo);  // bar
```

甚至还可以写得更简洁一些：

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

```
const handler = {
  get: Reflect.get
};

const proxy = new Proxy(target, handler);

console.log(proxy.foo);  // bar
console.log(target.foo); // bar
```

事实上，如果真想创建一个可以捕获所有方法，然后将每个方法转发给对应反射 API 的空代理，那么甚至不需要定义处理程序对象：

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

const proxy = new Proxy(target, Reflect);

console.log(proxy.foo);   // bar
console.log(target.foo);  // bar
```

反射 API 为开发者准备好了样板代码，在此基础上开发者可以用最少的代码修改捕获的方法。比如，下面的代码在某个属性被访问时，会对返回的值进行一番修饰：

```
const target = {
  foo: 'bar',
  baz: 'qux'
};

const handler = {
  get(trapTarget, property, receiver) {
    let decoration = '';
    if (property === 'foo') {
      decoration = '!!!';
    }

    return Reflect.get(...arguments) + decoration;
  }
};

const proxy = new Proxy(target, handler);

console.log(proxy.foo);   // bar!!!
console.log(target.foo);  // bar

console.log(proxy.baz);   // qux
console.log(target.baz);  // qux
```

9.1.4　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，但也不是没有限制。根据 ECMAScript 规范，每个捕获的方法都知道目标对象上下文、捕获函数签名，而捕获处理程序的行为必须遵循“捕获器不变式”（trap invariant）。捕获器不变式因方法不同而异，但通常都会防止捕获器定义出现过于反常的行为。

比如，如果目标对象有一个不可配置且不可写的数据属性，那么在捕获器返回一个与该属性不同的值时，会抛出 `TypeError`：

```
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
```

9.1.5 可撤销代理

有时候可能需要中断代理对象与目标对象之间的联系。对于使用 new Proxy()创建的普通代理来说，这种联系会在代理对象的生命周期内一直持续存在。

Proxy 也暴露了 revocable()方法，这个方法支持撤销代理对象与目标对象的关联。撤销代理的操作是不可逆的。而且，撤销函数（revoke()）是幂等的，调用多少次的结果都一样。撤销代理之后再调用代理会抛出 TypeError。

撤销函数和代理对象是在实例化时同时生成的：

```
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
```

9.1.6 实用反射 API

某些情况下应该优先使用反射 API，这是有一些理由的。

1. 反射 API 与对象 API

在使用反射 API 时，要记住：

(1) 反射 API 并不限于捕获处理程序；

(2) 大多数反射 API 方法在 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。

2. 状态标记

很多反射方法返回称作“状态标记”的布尔值，表示意图执行的操作是否成功。有时候，状态标记比那些返回修改后的对象或者抛出错误（取决于方法）的反射 API 方法更有用。例如，可以使用反射 API 对下面的代码进行重构：

```
// 初始代码

const o = {};

try {
  Object.defineProperty(o, 'foo', 'bar');
  console.log('success');
} catch(e) {
  console.log('failure');
}
```

在定义新属性时如果发生问题，Reflect.defineProperty()会返回 false，而不是抛出错误。因此使用这个反射方法可以这样重构上面的代码：

```
// 重构后的代码

const o = {};

if(Reflect.defineProperty(o, 'foo', {value: 'bar'})) {
  console.log('success');
} else {
  console.log('failure');
}
```

以下反射方法都会提供状态标记：

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

3. 用一等函数替代操作符

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。

4. 安全地应用函数

在通过 apply 方法调用函数时，被调用的函数可能也定义了自己的 apply 属性（虽然可能性极小）。为绕过这个问题，可以使用定义在 Function 原型上的 apply 方法，比如：

```
Function.prototype.apply.call(myFunc, thisVal, argumentList);
```

这种可怕的代码完全可以使用 Reflect.apply 来避免：

```
Reflect.apply(myFunc, thisVal, argumentsList);
```

9.1.7 代理另一个代理

代理可以拦截反射 API 的操作，而这意味着完全可以创建一个代理，通过它去代理另一个代理。这样就可以在一个目标对象之上构建多层拦截网：

```
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
```

9.1.8 代理的问题与不足

代理是在 ECMAScript 现有基础之上构建起来的一套新 API，因此其实现已经尽力做到最好了。很大程度上，代理作为对象的虚拟层可以正常使用。但在某些情况下，代理也不能与现在的 ECMAScript 机制很好地协同。

1. 代理中的 this

代理潜在的一个问题来源是 this 值。我们知道，方法中的 this 通常指向调用这个方法的对象：

```
const target = {
  thisValEqualsProxy() {
    return this === proxy;
  }
}

const proxy = new Proxy(target, {});

console.log(target.thisValEqualsProxy());  // false
console.log(proxy.thisValEqualsProxy());   // true
```

从直觉上讲，这样完全没有问题：调用代理上的任何方法，比如 proxy.outerMethod()，而这个方法进而又会调用另一个方法，如 this.innerMethod()，实际上都会调用 proxy.innerMethod()。多数情况下，这是符合预期的行为。可是，如果目标对象依赖于对象标识，那就可能碰到意料之外的问题。

还记得第 6 章中通过 WeakMap 保存私有变量的例子吧，以下是它的简化版：

```
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);
  }
}
```

由于这个实现依赖 User 实例的对象标识，在这个实例被代理的情况下就会出问题：

```
const user = new User(123);
console.log(user.id); // 123

const userInstanceProxy = new Proxy(user, {});
console.log(userInstanceProxy.id); // undefined
```

这是因为 User 实例一开始使用目标对象作为 WeakMap 的键，代理对象却尝试从**自身**取得这个实例。要解决这个问题，就需要重新配置代理，把代理 User 实例改为代理 User 类本身。之后再创建代理的实例就会以代理实例作为 WeakMap 的键了：

```
const UserClassProxy = new Proxy(User, {});
const proxyUser = new UserClassProxy(456);
console.log(proxyUser.id);
```

2. 代理与内部槽位

代理与内置引用类型（比如 Array）的实例通常可以很好地协同，但有些 ECMAScript 内置类型可能会依赖代理无法控制的机制，结果导致在代理上调用某些方法会出错。

一个典型的例子就是 Date 类型。根据 ECMAScript 规范，Date 类型方法的执行依赖 this 值上的内部槽位[[NumberDate]]。代理对象上不存在这个内部槽位，而且这个内部槽位的值也不能通过普通的 get()和 set()操作访问到，于是代理拦截后本应转发给目标对象的方法会抛出 TypeError：

```
const target = new Date();
const proxy = new Proxy(target, {});

console.log(proxy instanceof Date);  // true

proxy.getDate();  // TypeError: 'this' is not a Date object
```

9.2　代理捕获器与反射方法

代理可以捕获 13 种不同的基本操作。这些操作有各自不同的反射 API 方法、参数、关联 ECMAScript 操作和不变式。

正如前面示例所展示的，有几种不同的 JavaScript 操作会调用同一个捕获器处理程序。不过，对于在代理对象上执行的任何一种操作，只会有一个捕获处理程序被调用。不会存在重复捕获的情况。

只要在代理上调用，所有捕获器都会拦截它们对应的反射 API 操作。

9.2.1 get()

get()捕获器会在获取属性值的操作中被调用。对应的反射 API 方法为 Reflect.get()。

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
  get(target, property, receiver) {
    console.log('get()');
    return Reflect.get(...arguments)
  }
});

proxy.foo;
// get()
```

1. 返回值

返回值无限制。

2. 拦截的操作

- proxy.property
- proxy[property]
- Object.create(proxy)[property]
- Reflect.get(proxy, property, receiver)

3. 捕获器处理程序参数

- target：目标对象。
- property：引用的目标对象上的字符串键属性。[①]
- receiver：代理对象或继承代理对象的对象。

4. 捕获器不变式

如果 target.property 不可写且不可配置，则处理程序返回的值必须与 target.property 匹配。

如果 target.property 不可配置且[[Get]]特性为 undefined，处理程序的返回值也必须是 undefined。

9.2.2 set()

set()捕获器会在设置属性值的操作中被调用。对应的反射 API 方法为 Reflect.set()。

```
const myTarget = {};

const proxy = new Proxy(myTarget, {
  set(target, property, value, receiver) {
    console.log('set()');
    return Reflect.set(...arguments)
  }
});

proxy.foo = 'bar';
// set()
```

1. 返回值

返回 true 表示成功；返回 false 表示失败，严格模式下会抛出 TypeError。

① 严格来讲，property 参数除了字符串键，也可能是符号（symbol）键。后面几处也一样。——译者注