12 Прокси-объекты и Reflection API

Одной из целей обеих спецификаций, ECMAScript 5 и ECMAScript 6, было снятие завесы таинственности с функциональных возможностей JavaScript. Например, до ECMAScript 5 встроенные объекты JavaScript могли иметь свойства, неперечислимые и недоступные для записи, но разработчики не имели возможности определять собственные свойства как неперечислимые или недоступные для записи. В ECMAScript 5 появился метод `Object.defineProperty()`, который позволил разработчикам делать то, что движки JavaScript давно уже умели делать.

Спецификация ECMAScript 6 дает разработчикам еще более полный доступ к возможностям движка JavaScript, добавляя встроенные объекты. Чтобы позволить разработчикам создавать встроенные объекты, язык открывает доступ к внутренним механизмам объектов посредством *прокси-объектов* (*proxies*) — оберток, которые могут перехватывать и изменять порядок выполнения низкоуровневых операций движком JavaScript. В этой главе сначала подробно описывается проблема, решить которую призваны прокси-объекты, а затем обсуждаются приемы создания и эффективного использования прокси-объектов.

Проблема с массивами

До выхода ECMAScript 6 разработчики не могли имитировать работу объектов массивов в собственных объектах. Свойство `length` массива изменяется автоматически, когда программа присваивает значения определенным элементам этого массива, а изменяя свойство `length`, можно оказывать влияние на элементы массива. Например:

```
let colors = ["red", "green", "blue"];

console.log(colors.length); // 3

colors[3] = "black";

console.log(colors.length); // 4
```

```
console.log(colors[3]);     // "black"

colors.length = 2;

console.log(colors.length); // 2
console.log(colors[3]);     // undefined
console.log(colors[2]);     // undefined
console.log(colors[1]);     // "green"
```

Первоначально массив `colors` имеет три элемента. Операция присваивания строки `"black"` элементу `colors[3]` автоматически увеличивает свойство `length` до 4. Присваивание числа 2 свойству `length` удаляет два последних элемента из массива, оставляя два первых. В ECMAScript 5 отсутствовали механизмы, которые позволили бы разработчикам реализовать аналогичное поведение, но прокси-объекты изменили ситуацию.

ПРИМЕЧАНИЕ

Такое нестандартное поведение свойств с числовыми именами и свойства `length` объясняет, почему в ECMAScript 6 массивы считаются экзотическими объектами.

Введение в прокси-объекты и Reflection API

Вызов `new Proxy()` создает прокси-объект для использования вместо другого объекта (называется *целевым*). Прокси *виртуализирует* цель так, что оба объекта, прокси и целевой, выглядят функционально одинаковыми.

Прокси-объекты позволяют перехватывать низкоуровневые операции целевого объекта, которые иначе являются внутренними для движка JavaScript. Перехват этих низкоуровневых операций производится с помощью *ловушки* — функции, отвечающей за определенную операцию.

Программный интерфейс механизма рефлексии (Reflection API) представлен объектом `Reflect` — коллекцией методов, которые предоставляют поведение по умолчанию к тем же низкоуровневым операциям, которые могут переопределять прокси-объекты. Каждой ловушке в прокси соответствует метод в `Reflect`. Эти методы имеют те же имена и принимают те же аргументы, что и соответствующие им ловушки в прокси. В табл. 12.1 приводится краткая сводка по ловушкам в прокси.

Каждая ловушка переопределяет некоторую встроенную операцию с объектами JavaScript, позволяя вам перехватывать и изменять их. Если понадобится использовать неизмененные встроенные операции, они всегда доступны в виде соответствующих методов Reflection API. Связь между прокси-объектами и Reflection API станет понятнее, когда вы начнете создавать прокси, поэтому продолжайте читать и исследуйте предлагаемые примеры.

Таблица 12.1. Прокси-ловушки в JavaScript

Ловушка в прокси	Переопределяемая операция	Метод по умолчанию
`get`	Чтение значения свойства	`Reflect.get()`
`set`	Запись значения в свойство	`Reflect.set()`
`has`	Оператор `in`	`Reflect.has()`
`deleteProperty`	Оператор `delete`	`Reflect.deleteProperty()`
`getPrototypeOf`	`Object.getPrototypeOf()`	`Reflect.getPrototypeOf`
`setPrototypeOf`	`Object.setPrototypeOf()`	`Reflect.setPrototypeOf()`
`isExtensible`	`Object.isExtensible()`	`Reflect.isExtensible()`
`preventExtensions`	`Object.preventExtensions()`	`Reflect.preventExtensions()`
`getOwnProperty-Descriptor`	`Object.getOwnPropertyDescriptor()`	`Reflect.getOwnPropertyDescriptor()`
`defineProperty`	`Object.defineProperty()`	`Reflect.defineProperty`
`ownKeys`	`Object.keys()`, `Object.getOwnProperty-Names()` и `Object.getOwnProperty-Symbols()`	`Reflect.ownKeys()`
`apply`	Вызов функции	`Reflect.apply()`
`construct`	Вызов функции с ключевым словом `new`	`Reflect.construct()`

ПРИМЕЧАНИЕ

В первоначальной версии спецификации ECMAScript 6 имелась еще одна ловушка — `enumerate`, предназначавшаяся для изменения поведения перечисления свойств объекта в цикле `for-in` и методе `Object.keys()`. Однако она была убрана в спецификации ECMAScript 7 (ECMAScript 2016), потому что в ходе реализации вскрылись некоторые сложности. Ловушка `enumerate` не поддерживается ни в одном окружении JavaScript и потому не рассматривается в этой главе.

Создание простого прокси-объекта

Когда вызывается конструктор `Proxy`, чтобы создать прокси-объект, ему передаются два аргумента: цель и обработчик. *Обработчик* — это объект, определяющий одну или несколько ловушек. Прокси-объект использует реализации по умолчанию для

всех операций, кроме тех, для которых определены ловушки. Чтобы создать простейший прокси-объект, который лишь передает вызовы реализации по умолчанию, можно использовать обработчик без ловушек, например:

```
let target = {};

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

proxy.name = "proxy";
console.log(proxy.name);  // "proxy"
console.log(target.name); // "proxy"

target.name = "target";
console.log(proxy.name);  // "target"
console.log(target.name); // "target"
```

В этом примере прокси-объект `proxy` делегирует выполнение всех операций непосредственно целевому объекту `target`. Когда свойству `proxy.name` присваивается значение `"proxy"`, в объекте `target` создается свойство `name`. Сам объект `proxy` не хранит это свойство; он просто передает операцию объекту `target`. Аналогично обращения к `proxy.name` и `target.name` возвращают одно и то же значение, потому что оба они в конечном итоге ссылаются на свойство `target.name`. Это также означает, что если присвоить свойству `target.name` новое значение, оно станет также доступно через `proxy.name`. Конечно, прокси-объекты без ловушек не представляют особого интереса, поэтому давайте посмотрим, что получится, если определить какую-нибудь ловушку.

Проверка свойств с помощью ловушки set

Предположим, что требуется создать объект, свойства которого могут принимать только числовые значения. Это означает необходимость проверки каждого нового свойства, добавляемого в объект, и возбуждения ошибки, если присваиваемое ему значение не является числом. Для решения этой задачи можно определить ловушку `set`, переопределяющую поведение по умолчанию операции присваивания значения. Ловушка `set` принимает четыре аргумента:

- `trapTarget`. Объект, в который добавляется свойство (цель для прокси-объекта).
- `key`. Ключ свойства (строка или символ) для записи.
- `value`. Значение, записываемое в свойство.
- `receiver`. Объект, в котором определена выполняемая операция (обычно прокси-объект).

Ловушке `set` соответствует метод `Reflect.set()`, который определяет поведение по умолчанию данной операции. Метод `Reflect.set()` принимает те же четыре

аргумента, что и ловушка `set`, что дает возможность использовать этот метод внутри ловушки. Ловушка должна вернуть `true`, если значение было присвоено свойству, и `false` — в противном случае. (Метод `Reflect.set()` возвращает правильное значение в зависимости от успеха операции.)

Для проверки значений, присваиваемых свойствам, можно использовать ловушку `set`. Например:

```
let target = {
    name: "target"
};

let proxy = new Proxy(target, {
    set(trapTarget, key, value, receiver) {

        // игнорировать существующие свойства объекта,
        // на которые действие ловушки не распространяется
        if (!trapTarget.hasOwnProperty(key)) {
            if (isNaN(value)) {
                throw new TypeError("Property must be a number.");
            }
        }

        // добавить свойство
        return Reflect.set(trapTarget, key, value, receiver);
    }
});

// добавление нового свойства
proxy.count = 1;
console.log(proxy.count);     // 1
console.log(target.count);    // 1

// свойству name можно присвоить строку,
// потому что оно уже существует
proxy.name = "proxy";
console.log(proxy.name);     // "proxy"
console.log(target.name);    // "proxy"

// следующая операция вызовет ошибку
proxy.anotherName = "proxy";
```

В этом примере определяется ловушка в прокси-объекте, которая проверяет значение любого нового свойства, добавляемого в объект `target`. Когда выполняется инструкция `proxy.count = 1`, вызывается ловушка `set`. В аргументе `trapTarget` передается ссылка на `target`, в аргументе `key` — строка `"count"`, в аргументе `value` — значение 1 и в аргументе `receiver` (в этом примере не используется) — ссылка на `proxy`. В объекте `target` отсутствует свойство с именем `count`, поэтому ловушка проверяет значение `value`, передавая его в вызов `isNaN()`. Если в результате будет получено значение `NaN`, значит, производится попытка присвоить свойству нечисловое

значение, и поэтому возбуждается ошибка. Так как в этом примере свойству `count` присваивается 1, ловушка вызывает `Reflect.set()` с теми же четырьмя аргументами, которые получила сама, чтобы добавить новое свойство.

Когда свойству `proxy.name` присваивается строка, операция выполняется успешно. Так как объект `target` уже имеет свойство `name`, оно исключается из проверки вызовом метода `trapTarget.hasOwnProperty()`. Это гарантирует поддержку свойств, которые прежде имели нечисловые значения.

Однако попытка присвоить строку свойству `proxy.anotherName` вызывает ошибку. Свойство `anotherName` не существовало в объекте `target` прежде, поэтому оно подвергается проверке. В процессе проверки значения возбуждается ошибка, потому что `"proxy"` не является числовым значением.

Ловушка `set` позволяет перехватывать операции записи в свойства, а ловушка `get` — операции чтения свойств.

Проверка формы объектов с помощью ловушки get

Одной из необычных особенностей JavaScript, которая иногда может вводить в заблуждение, является операция чтения свойств, которая не вызывает ошибку, если предпринимается попытка прочитать несуществующее свойство. Вместо ошибки она просто возвращает значение `undefined`, как в следующем примере:

```
let target = {};

console.log(target.name);   // undefined
```

В большинстве других языков попытка прочитать `target.name` вызовет ошибку, потому что свойство не существует. Но в JavaScript в качестве значения свойства `target.name` будет возвращено `undefined`. Если вам доводилось работать с большими программами, вы наверняка знаете, что подобное поведение может вызывать существенные проблемы, особенно когда в имени свойства допущена опечатка. Прокси-объекты помогают решить эту проблему за счет организации проверки формы объекта.

Форма объекта (*object shape*) — это коллекция свойств и методов, доступных в объекте. Движки JavaScript используют формы объектов для оптимизации кода, часто создавая классы, представляющие объекты. Если можно с уверенностью сказать, что объект всегда будет иметь один и тот же набор свойств и методов (такое поведение можно гарантировать с помощью методов `Object.preventExtensions()`, `Object.seal()` или `Object.freeze()`), тогда возбуждение ошибки при попытке обратиться к несуществующему свойству может оказаться полезным приобретением. Прокси-объекты позволяют легко проверить форму объекта.

Так как проверка свойства должна выполняться только при чтении свойств, ее можно реализовать в ловушке `get`. Ловушка `get` вызывается, когда производится операция чтения свойства, даже если это свойство отсутствует в объекте. Она принимает три аргумента:

`trapTarget`. Объект, свойство которого читается (цель для прокси-объекта).

`key`. Ключ свойства (строка или символ) для чтения.

`receiver`. Объект, в котором определена выполняемая операция (обычно прокси-объект).

Эти аргументы повторяют аргументы ловушки `set` с одним заметным исключением: отсутствует аргумент `value`, потому что ловушка `get` не выполняет запись значения. Ловушке `get` соответствует метод `Reflect.get()`, который принимает те же три аргумента, что и ловушка `get`, и возвращает значение свойства.

Ловушку `get` и метод `Reflect.get()` можно использовать, чтобы возбуждать ошибку при любой попытке прочитать свойство, отсутствующее в целевом объекте, как показано ниже:

```
let proxy = new Proxy({}, {
    get(trapTarget, key, receiver) {
        if (!(key in receiver)) {
            throw new TypeError("Property " + key + " doesn't exist.");
        }

        return Reflect.get(trapTarget, key, receiver);
    }
});

// допускается добавление свойств
proxy.name = "proxy";
console.log(proxy.name);     // "proxy"

// обращение к несуществующему свойству вызывает ошибку
console.log(proxy.nme);      // throws an error
```

Ловушка `get` в этом примере перехватывает операции чтения свойств. Оператор `in` проверяет наличие свойства в объекте `receiver`. Здесь в операторе `in` используется `receiver` вместо `trapTarget` на тот случай, если в прокси-объекте `receiver` определена ловушка `has`, о которой я расскажу в следующем разделе. Использование `trapTarget` в такой ситуации приведет к вызову ловушки `has` и теоретически может дать неверный результат. Если свойство отсутствует, возбуждается ошибка; в противном случае используется поведение по умолчанию.

Это решение позволяет добавлять новые свойства, такие как `proxy.name`, записывать значения и читать их. Последняя строка в примере содержит опечатку: под `proxy.nme` программист, вероятно, подразумевал `proxy.name`. Она вызовет ошибку, потому что свойство `nme` отсутствует в объекте.