Proxy и Reflect

Объект `Proxy` «оборачивается» вокруг другого объекта и может перехватывать (и, при желании, самостоятельно обрабатывать) разные действия с ним, например чтение/запись свойств и другие. Далее мы будем называть такие объекты «прокси».

Прокси используются во многих библиотеках и некоторых браузерных фреймворках. В этой главе мы увидим много случаев применения прокси в решении реальных задач.

Синтаксис:

```
let proxy = new Proxy(target, handler);
```

- `target` – это объект, для которого нужно сделать прокси, может быть чем угодно, включая функции.
- `handler` – конфигурация прокси: объект с «ловушками» («traps»): методами, которые перехватывают разные операции, например, ловушка `get` – для чтения свойства из `target`, ловушка `set` – для записи свойства в `target` и так далее.

При операциях над `proxy`, если в `handler` имеется соответствующая «ловушка», то она срабатывает, и прокси имеет возможность по-своему обработать её, иначе операция будет совершена над оригинальным объектом `target`.

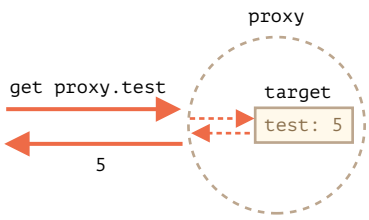
В качестве начального примера создадим прокси без всяких ловушек:

```
let target = {};
let proxy = new Proxy(target, {}); // пустой handler

proxy.test = 5; // записываем в прокси (1)
alert(target.test); // 5, свойство появилось в target!

alert(proxy.test); // 5, мы также можем прочитать его и

for(let key in proxy) alert(key); // test, итерация раб
```

Так как нет ловушек, то все операции на `proxy` применяются к оригинальному объекту `target`.

1. Запись свойства `proxy.test=` устанавливает значение на `target`.
2. Чтение свойства `proxy.test` возвращает значение из `target`.
3. Итерация по `proxy` возвращает значения из `target`.

Как мы видим, без ловушек `proxy` является прозрачной обёрткой над `target`.



`Proxy` – это особый, «экзотический», объект, у него нет собственных свойств. С пустым `handler` он просто перенаправляет все операции на `target`.

Чтобы активировать другие его возможности, добавим ловушки.

Что именно мы можем ими перехватить?

Для большинства действий с объектами в спецификации JavaScript есть так называемый «внутренний метод», который на самом низком уровне описывает, как его выполнять. Например, `[[Get]]` – внутренний метод для чтения свойства, `[[Set]]` – для записи свойства, и так далее. Эти методы используются только в спецификации, мы не можем обратиться напрямую к ним по имени.

Ловушки как раз перехватывают вызовы этих внутренних методов. Полный список методов, которые можно перехватывать, перечислен в спецификации Proxy, а также в таблице ниже.

Для каждого внутреннего метода в этой таблице указана ловушка, то есть имя метода, который мы можем добавить в параметр `handler` при создании `new Proxy`, чтобы перехватывать данную операцию:

Внутренний метод	Ловушка	Что вызывает
`[[Get]]`	`get`	чтение свойства
`[[Set]]`	`set`	запись свойства
`[[HasProperty]]`	`has`	оператор `in`
`[[Delete]]`	`deleteProperty`	оператор `delete`
`[[Call]]`	`apply`	вызов функции
`[[Construct]]`	`construct`	оператор `new`
`[[GetPrototypeOf]]`	`getPrototypeOf`	Object.getPrototypeOf
`[[SetPrototypeOf]]`	`setPrototypeOf`	Object.setPrototypeOf
`[[IsExtensible]]`	`isExtensible`	Object.isExtensible
`[[PreventExtensions]]`	`preventExtensions`	Object.preventExtensions
`[[DefineOwnProperty]]`	`defineProperty`	Object.defineProperty, Object.defineProperties
`[[GetOwnProperty]]`	`getOwnPropertyDescriptor`	Object.getOwnPropertyDescriptor, `for..in`, `Object.keys/values/entries`
`[[OwnPropertyKeys]]`	`ownKeys`	Object.getOwnPropertyNames, Object.getOwnPropertySymbols, `for..in`, `Object.keys/values/entries`

Инварианты

JavaScript налагает некоторые условия – инварианты на реализацию внутренних методов и ловушек.

Большинство из них касаются возвращаемых значений:

- Метод `[[Set]]` должен возвращать `true`, если значение было успешно записано, иначе `false`.
- Метод `[[Delete]]` должен возвращать `true`, если значение было успешно удалено, иначе `false`.
- ...и так далее, мы увидим больше в примерах ниже.

Есть и другие инварианты, например:

- Метод `[[GetPrototypeOf]]`, применённый к прокси, должен возвращать то же значение, что и метод `[[GetPrototypeOf]]`, применённый к оригинальному объекту. Другими словами, чтение прототипа объекта прокси всегда должно возвращать прототип оригинального объекта.

Ловушки могут перехватывать вызовы этих методов, но должны выполнять указанные условия.

Инварианты гарантируют корректное и последовательное поведение конструкций и методов языка. Полный список инвариантов можно найти в спецификации, хотя скорее всего вы не нарушите эти условия, если только не соберётесь делать что-то совсем уж странное.

Теперь давайте посмотрим, как это всё работает, на реальных примерах.

Значение по умолчанию с ловушкой «get»

Чаще всего используются ловушки на чтение/запись свойств.

Чтобы перехватить операцию чтения, `handler` должен иметь метод `get(target, property, receiver)`.

Он срабатывает при попытке прочитать свойство объекта, с аргументами:

- `target` – это оригинальный объект, который передавался первым аргументом в конструктор `new Proxy`,
- `property` – имя свойства,
- `receiver` – если свойство объекта является геттером, то `receiver` – это объект, который будет использован как `this` при его вызове. Обычно это сам объект прокси (или наследующий от него объект). Прямо сейчас нам не понадобится этот аргумент, подробнее разберём его позже.

Давайте применим ловушку `get`, чтобы реализовать «значения по умолчанию» для свойств объекта.

Например, сделаем числовой массив, так чтобы при чтении из него несуществующего элемента возвращался `0`.

Обычно при чтении из массива несуществующего свойства возвращается `undefined`, но мы обернём обычный массив в прокси, который перехватывает операцию чтения свойства из массива и возвращает `0`, если такого элемента нет:

```
let numbers = [0, 1, 2];

numbers = new Proxy(numbers, {
  get(target, prop) {
    if (prop in target) {
      return target[prop];
    } else {
      return 0; // значение по умолчанию
    }
  }
});

alert( numbers[1] ); // 1
alert( numbers[123] ); // 0 (нет такого элемента)
```

Как видно, это очень легко сделать при помощи ловушки `get`.

Мы можем использовать `Proxy` для реализации любой логики возврата значений по умолчанию.

Представим, что у нас есть объект-словарь с фразами на английском и их переводом на испанский:

```
let dictionary = {
  'Hello': 'Hola',
  'Bye': 'Adiós'
};

alert( dictionary['Hello'] ); // Hola
alert( dictionary['Welcome'] ); // undefined
```

Сейчас, если фразы в `dictionary` нет, при чтении возвращается `undefined`. Но на практике оставлять фразы непереведёнными лучше, чем использовать `undefined`. Поэтому давайте сделаем так, чтобы при отсутствии перевода возвращалась оригинальная фраза на английском вместо `undefined`.

Чтобы достичь этого, обернём `dictionary` в прокси, перехватывающий операцию чтения:

```
let dictionary = {
  'Hello': 'Hola',
  'Bye': 'Adiós'
};

dictionary = new Proxy(dictionary, {
  get(target, phrase) { // перехватываем чтение свойств
    if (phrase in target) { // если перевод для фразы е
      return target[phrase]; // возвращаем его
    } else {
      // иначе возвращаем непереведённую фразу
      return phrase;
    }
  }
});

// Запросим перевод произвольного выражения в словаре!
// В худшем случае оно не будет переведено
alert( dictionary['Hello'] ); // Hola
alert( dictionary['Welcome to Proxy']); // Welcome to P
```

Прокси следует использовать везде вместо `target`

Пожалуйста, обратите внимание: прокси перезаписывает переменную:

```
dictionary = new Proxy(dictionary, ...);
```

Прокси должен заменить собой оригинальный объект повсюду. Никто не должен ссылаться на оригинальный объект после того, как он был проксирован. Иначе очень легко запутаться.

Валидация с ловушкой «set»

Допустим, мы хотим сделать массив исключительно для чисел. Если в него добавляется значение иного типа, то это должно приводить к ошибке.

Ловушка `set` срабатывает, когда происходит запись свойства.

`set(target, property, value, receiver)`:

- `target` – это оригинальный объект, который передавался первым аргументом в конструктор `new Proxy`,
- `property` – имя свойства,
- `value` – значение свойства,
- `receiver` – аналогично ловушке `get`, этот аргумент имеет значение, только если свойство – сеттер.

Ловушка `set` должна вернуть `true`, если запись прошла успешно, и `false` в противном случае (будет сгенерирована ошибка `TypeError`).

Давайте применим её для проверки новых значений:

```
let numbers = [];

numbers = new Proxy(numbers, { // (*)
  set(target, prop, val) { // для перехвата записи свой
    if (typeof val == 'number') {
      target[prop] = val;
      return true;
    } else {
      return false;
    }
  }
});

numbers.push(1); // добавилось успешно
```


```
numbers.push(2); // добавилось успешно
alert("Длина: " + numbers.length); // 2

numbers.push("тест"); // TypeError (ловушка set на прок

alert("Интерпретатор никогда не доходит до этой строки
```

Обратите внимание, что встроенная функциональность массива по-прежнему работает! Значения добавляются методом `push`. Свойство `length` при этом увеличивается. Наш прокси ничего не ломает.

Нам не нужно переопределять методы массива `push` и `unshift` и другие, чтобы добавлять туда проверку на тип, так как внутри себя они используют операцию `[[Set]]`, которая перехватывается прокси.

Таким образом, код остаётся чистым и прозрачным.

> **Не забывайте вернуть `true`**
>
> Как сказано ранее, нужно соблюдать инварианты.
>
> Для `set` реализация ловушки должна возвращать `true` в случае успешной записи свойства.
>
> Если забыть это сделать или возвратить любое ложное значение, это приведёт к ошибке `TypeError`.

Перебор при помощи «ownKeys» и «getOwnPropertyDescriptor»

`Object.keys`, цикл `for..in` и большинство других методов, которые работают со списком свойств объекта, используют внутренний метод `[[OwnPropertyKeys]]` (перехватываемый ловушкой `ownKeys`) для их получения.

Такие методы различаются в деталях:

- `Object.getOwnPropertyNames(obj)` возвращает не-символьные ключи.
- `Object.getOwnPropertySymbols(obj)` возвращает символьные ключи.
- `Object.keys/values()` возвращает не-символьные ключи/значения с флагом `enumerable` (подробнее про флаги свойств было в главе Флаги и дескрипторы свойств).
- `for..in` перебирает не-символьные ключи с флагом `enumerable`, а также ключи прототипов.

...Но все они начинают с этого списка.

В примере ниже мы используем ловушку `ownKeys`, чтобы цикл `for..in` по объекту, равно как `Object.keys` и `Object.values` пропускали свойства, начинающиеся с подчёркивания `_`:

```
let user = {
  name: "Вася",
  age: 30,
  _password: "***"
};

user = new Proxy(user, {
  ownKeys(target) {
    return Object.keys(target).filter(key => !key.start
  }
});

// ownKeys исключил _password
for(let key in user) alert(key); // name, затем: age

// аналогичный эффект для этих методов:
```

```
17  alert( Object.keys(user) ); // name,age
18  alert( Object.values(user) ); // Вася,30
```


Поделиться

Редактировать на GitHub

Как видно, работает.

Впрочем, если мы попробуем возвратить ключ, которого в объекте на самом деле нет, то `Object.keys` его не выдаст:

```
1  let user = { };
2
3  user = new Proxy(user, {
4    ownKeys(target) {
5      return ['a', 'b', 'c'];
6    }
7  });
8
9  alert( Object.keys(user) ); // <пусто>
```

Почему? Причина проста: `Object.keys` возвращает только свойства с флагом `enumerable`. Для того, чтобы определить, есть ли этот флаг, он для каждого свойства вызывает внутренний метод `[[GetOwnProperty]]`, который получает его дескриптор. А в данном случае свойство отсутствует, его дескриптор пуст, флага `enumerable` нет, поэтому оно пропускается.

Чтобы `Object.keys` возвращал свойство, нужно либо чтобы свойство в объекте физически было, причём с флагом `enumerable`, либо перехватить вызовы `[[GetOwnProperty]]` (это делает ловушка `getOwnPropertyDescriptor`), и там вернуть дескриптор с `enumerable: true`.

Вот так будет работать:

```
1  let user = { };
2
3  user = new Proxy(user, {
4    ownKeys(target) { // вызывается 1 раз для получения с
5      return ['a', 'b', 'c'];
6    },
7
8    getOwnPropertyDescriptor(target, prop) { // вызываетс
9      return {
10       enumerable: true,
11       configurable: true
12       /* ...другие флаги, возможно, "value: ..." */
13     };
14   }
15
16 });
17
18 alert( Object.keys(user) ); // a, b, c
```

Ещё раз заметим, что получение дескриптора нужно перехватывать только если свойство отсутствует в самом объекте.

Защищённые свойства с ловушкой «deleteProperty» и другими

Существует широко распространённое соглашение о том, что свойства и методы, название которых начинается с символа подчёркивания `_`, следует считать внутренними. К ним не следует обращаться снаружи объекта.

Однако технически это всё равно возможно:

```
1  let user = {
2    name: "Вася",
3    _password: "secret"
4  };
```

```
5
6  alert(user._password); // secret
```


Давайте применим прокси, чтобы защитить свойства, начинающиеся на `_`, от доступа извне.

Нам будут нужны следующие ловушки:

- `get` – для того, чтобы сгенерировать ошибку при чтении такого свойства,
- `set` – для того, чтобы сгенерировать ошибку при записи,
- `deleteProperty` – для того, чтобы сгенерировать ошибку при удалении,
- `ownKeys` – для того, чтобы исключить такие свойства из `for..in` и методов типа `Object.keys`.

Вот соответствующий код:

```
1  let user = {
2    name: "Вася",
3    _password: "***"
4  };
5
6  user = new Proxy(user, {
7    get(target, prop) {
8      if (prop.startsWith('_')) {
9        throw new Error("Отказано в доступе");
10     } else {
11       let value = target[prop];
12       return (typeof value === 'function') ? value.bind
13     }
14   },
15   set(target, prop, val) { // перехватываем запись свой
16     if (prop.startsWith('_')) {
17       throw new Error("Отказано в доступе");
18     } else {
19       target[prop] = val;
20       return true;
21     }
22   },
23   deleteProperty(target, prop) { // перехватываем удале
24     if (prop.startsWith('_')) {
25       throw new Error("Отказано в доступе");
26     } else {
27       delete target[prop];
28       return true;
29     }
30   },
31   ownKeys(target) { // перехватываем попытку итерации
32     return Object.keys(target).filter(key => !key.start
33   }
34 });
35
36 // "get" не позволяет прочитать _password
37 try {
38   alert(user._password); // Error: Отказано в доступе
39 } catch(e) { alert(e.message); }
40
41 // "set" не позволяет записать _password
42 try {
43   user._password = "test"; // Error: Отказано в доступе
44 } catch(e) { alert(e.message); }
45
46 // "deleteProperty" не позволяет удалить _password
47 try {
48   delete user._password; // Error: Отказано в доступе
49 } catch(e) { alert(e.message); }
50
51 // "ownKeys" исключает _password из списка видимых для
52 for(let key in user) alert(key); // name
```

Обратите внимание на важную деталь в ловушке `get` на строке `(*)`:

```
1  get(target, prop) {
2    // ...
3    let value = target[prop];
4    return (typeof value === 'function') ? value.bind(tar
5  }
```

Зачем для функции вызывать `value.bind(target)`?

Всё дело в том, что метод самого объекта, например `user.checkPassword()`, должен иметь доступ к свойству `_password`:

```
1  user = {
2    // ...
3    checkPassword(value) {
4      // метод объекта должен иметь доступ на чтение _pas
5      return value === this._password;
6    }
7  }
```

Вызов `user.checkPassword()` получает проксированный объект `user` в качестве `this` (объект перед точкой становится `this`), так что когда такой вызов обращается к `this._password`, ловушка `get` вступает в действие (она срабатывает при любом чтении свойства), и выбрасывается ошибка.

Поэтому мы привязываем контекст к методам объекта – оригинальный объект `target` в строке `(*)`. Тогда их дальнейшие вызовы будут использовать `target` в качестве `this`, без всяких ловушек.

Такое решение обычно работает, но не является идеальным, поскольку метод может передать оригинальный объект куда-то ещё, и возможна путаница: где изначальный объект, а где – проксированный.

К тому же, объект может проксироваться несколько раз (для добавления различных возможностей), и если передавать методу исходный, то могут быть неожиданности.

Так что везде использовать такой прокси не стоит.

Приватные свойства в классах

Современные интерпретаторы JavaScript поддерживают приватные свойства в классах. Названия таких свойств должны начинаться с символа `#`. Они подробно описаны в главе Приватные и защищённые методы и свойства. Для них не нужны подобные прокси.

Впрочем, приватные свойства имеют свои недостатки. В частности, они не наследуются.

«В диапазоне» с ловушкой «has»

Давайте посмотрим ещё примеры.

Предположим, у нас есть объект `range`, описывающий диапазон:

```
1  let range = {
2    start: 1,
3    end: 10
4  };
```

Мы бы хотели использовать оператор `in`, чтобы проверить, что некоторое число находится в указанном диапазоне.

Ловушка `has` перехватывает вызовы `in`.

```
has(target, property)
```

- `target` – это оригинальный объект, который передавался первым аргументом в конструктор `new Proxy`,
- `property` – имя свойства

Вот демо:

```
let range = {
  start: 1,
  end: 10
};

range = new Proxy(range, {
  has(target, prop) {
    return prop >= target.start && prop <= target.end
  }
});

alert(5 in range); // true
alert(50 in range); // false
```

Отлично выглядит, не правда ли? И очень просто в реализации.

Оборачиваем функции: «apply»

Мы можем оборачивать в прокси и функции.

Ловушка `apply(target, thisArg, args)` активируется при вызове прокси как функции:

- `target` – это оригинальный объект (как мы помним, функция – это объект в языке JavaScript),
- `thisArg` – это контекст `this`.
- `args` – список аргументов.

Например, давайте вспомним декоратор `delay(f, ms)`, созданный нами в главе Декораторы и переадресация вызова, call/apply.

Тогда мы обошлись без создания прокси. Вызов `delay(f, ms)` возвращал функцию, которая передавала вызовы `f` после `ms` миллисекунд.

Вот предыдущая реализация, на основе функции:

```
function delay(f, ms) {
  // возвращает обёртку, которая вызывает функцию f чер
  return function() { // (*)
    setTimeout(() => f.apply(this, arguments), ms);
  };
}

function sayHi(user) {
  alert(`Привет, ${user}!`);
}

// после обёртки вызовы sayHi будут срабатывать с задер
sayHi = delay(sayHi, 3000);

sayHi("Вася"); // Привет, Вася! (через 3 секунды)
```

Как мы уже видели, это в целом работает. Функция-обёртка в строке `(*)` вызывает нужную функцию с указанной задержкой.

Но наша функция-обёртка не перенаправляет операции чтения/записи свойства и другие. После обёртывания доступ к свойствам оригинальной функции, таким как `name`, `length`, и другим, будет потерян.

```
function delay(f, ms) {
  return function() {
    setTimeout(() => f.apply(this, arguments), ms);
```

```
  };
}

function sayHi(user) {
  alert(`Привет, ${user}!`);
}

alert(sayHi.length); // 1 (в функции length - это число

sayHi = delay(sayHi, 3000);

alert(sayHi.length); // 0 (в объявлении функции-обёртки
```

Прокси куда более мощные в этом смысле, поскольку они перенаправляют всё к оригинальному объекту.

Давайте используем прокси вместо функции-обёртки:

```
function delay(f, ms) {
  return new Proxy(f, {
    apply(target, thisArg, args) {
      setTimeout(() => target.apply(thisArg, args), ms)
    }
  });
}

function sayHi(user) {
  alert(`Привет, ${user}!`);
}

sayHi = delay(sayHi, 3000);

alert(sayHi.length); // 1 (*) прокси перенаправляет чте

sayHi("Вася"); // Привет, Вася! (через 3 секунды)
```

Результат такой же, но сейчас не только вызовы, но и другие операции на прокси перенаправляются к оригинальной функции. Таким образом, операция чтения свойства `sayHi.length` возвращает корректное значение в строке `(*)` после проксирования.

Мы получили лучшую обёртку.

Существуют и другие ловушки: полный список есть в начале этой главы. Использовать их можно по аналогии с вышеописанными.

Reflect

`Reflect` – встроенный объект, упрощающий создание прокси.

Ранее мы говорили о том, что внутренние методы, такие как `[[Get]]`, `[[Set]]` и другие, существуют только в спецификации, что к ним нельзя обратиться напрямую.

Объект `Reflect` делает это возможным. Его методы – минимальные обёртки вокруг внутренних методов.

Вот примеры операций и вызовы `Reflect`, которые делают то же самое:

Операция	Вызов `Reflect`	Внутренний метод
`obj[prop]`	`Reflect.get(obj, prop)`	`[[Get]]`
`obj[prop] = value`	`Reflect.set(obj, prop, value)`	`[[Set]]`
`delete obj[prop]`	`Reflect.deleteProperty(obj, prop)`	`[[Delete]]`
`new F(value)`	`Reflect.construct(F, value)`	`[[Construct]]`
...	...	...

Редактировать на GitHub

Например:

```
let user = {};

Reflect.set(user, 'name', 'Вася');

alert(user.name); // Вася
```

В частности, `Reflect` позволяет вызвать операторы (`new`, `delete` ...) как функции (`Reflect.construct`, `Reflect.deleteProperty`, ...). Это интересная возможность, но здесь нам важно другое.

Для каждого внутреннего метода, перехватываемого `Proxy`, есть соответствующий метод в `Reflect`, который имеет такое же имя и те же аргументы, что и у ловушки `Proxy`.

Поэтому мы можем использовать `Reflect`, чтобы перенаправить операцию на исходный объект.

В этом примере обе ловушки `get` и `set` прозрачно (как будто их нет) перенаправляют операции чтения и записи на объект, при этом выводя сообщение:

```
let user = {
  name: "Вася",
};

user = new Proxy(user, {
  get(target, prop, receiver) {
    alert(`GET ${prop}`);
    return Reflect.get(target, prop, receiver); // (1)
  },
  set(target, prop, val, receiver) {
    alert(`SET ${prop}=${val}`);
    return Reflect.set(target, prop, val, receiver); //
  }
});

let name = user.name; // выводит "GET name"
user.name = "Петя"; // выводит "SET name=Петя"
```

Здесь:

1. `Reflect.get` читает свойство объекта.
2. `Reflect.set` записывает свойство и возвращает `true` при успехе, иначе `false`.

То есть, всё очень просто – если ловушка хочет перенаправить вызов на объект, то достаточно вызвать `Reflect.<метод>` с теми же аргументами.

В большинстве случаев мы можем сделать всё то же самое и без `Reflect`, например, чтение свойства `Reflect.get(target, prop, receiver)` можно заменить на `target[prop]`. Но некоторые нюансы легко упустить.

Прокси для геттера

Рассмотрим конкретный пример, демонстрирующий, чем лучше `Reflect.get`, и заодно разберёмся, зачем в `get/set` нужен третий аргумент `receiver`, мы его ранее не использовали.

Допустим, у нас есть объект `user` со свойством `_name` и геттером для него.

Сделаем вокруг `user` прокси:

```
let user = {
  _name: "Гость",
  get name() {
    return this._name;
```

```
  }
};

let userProxy = new Proxy(user, {
  get(target, prop, receiver) {
    return target[prop];
  }
});

alert(userProxy.name); // Гость
```

Ловушка `get` здесь «прозрачная», она возвращает свойство исходного объекта и больше ничего не делает. Для нашего примера этого вполне достаточно.

Казалось бы, всё в порядке. Но давайте немного усложним пример.

Если мы унаследуем от проксированного `user` объект `admin`, то мы увидим, что он ведёт себя некорректно:

```
let user = {
  _name: "Гость",
  get name() {
    return this._name;
  }
};

let userProxy = new Proxy(user, {
  get(target, prop, receiver) {
    return target[prop]; // (*) target = user
  }
});

let admin = {
  __proto__: userProxy,
  _name: "Админ"
};

// Ожидается: Админ
alert(admin.name); // выводится Гость (?!?)
```

Обращение к свойству `admin.name` должно возвращать строку `"Админ"`, а выводит `"Гость"`!

В чём дело? Может быть, мы делаем что-то не так с наследованием?

Но если убрать прокси, то всё будет работать как ожидается.

На самом деле, проблема в прокси, в строке `(*)`.

1. При чтении `admin.name`, так как в объекте `admin` нет свойства `name`, оно ищется в прототипе.
2. Прототипом является прокси `userProxy`.
3. При чтении из прокси свойства `name` срабатывает ловушка `get` и возвращает его из исходного объекта как `target[prop]` в строке `(*)`.

   Вызов `target[prop]`, если `prop` – это геттер, запускает его код в контексте `this=target`. Поэтому результатом является `this._name` из исходного объекта `target`, то есть из `user`.

Именно для исправления таких ситуаций нужен `receiver`, третий аргумент ловушки `get`. В нём хранится ссылка на правильный контекст `this`, который нужно передать геттеру. В данном случае это `admin`.

Как передать геттеру контекст? Для обычной функции мы могли бы использовать `call/apply`, но это же геттер, его не вызывают, просто читают значение.

Это может сделать `Reflect.get`. Всё будет работать верно, если использовать его.

Вот исправленный вариант:

```
let user = {
  _name: "Гость",
  get name() {
    return this._name;
  }
};

let userProxy = new Proxy(user, {
  get(target, prop, receiver) { // receiver = admin
    return Reflect.get(target, prop, receiver); // (*)
  }
});


let admin = {
  __proto__: userProxy,
  _name: "Админ"
};

alert(admin.name); // Админ
```

Сейчас `receiver`, содержащий ссылку на корректный `this` (то есть на `admin`), передаётся геттеру посредством `Reflect.get` в строке `(*)`.

Можно переписать ловушку и короче:

```
get(target, prop, receiver) {
  return Reflect.get(...arguments);
}
```

Методы в `Reflect` имеют те же названия, что и соответствующие ловушки, и принимают такие же аргументы. Это было специально задумано при разработке спецификации JavaScript.

Так что `return Reflect...` даёт простую и безопасную возможность перенаправить операцию на оригинальный объект и при этом предохраняет нас от возможных ошибок, связанных с этим действием.

Ограничения прокси

Прокси – уникальное средство для настройки поведения объектов на самом низком уровне. Но они не идеальны, есть некоторые ограничения.

Встроенные объекты: внутренние слоты

Многие встроенные объекты, например `Map`, `Set`, `Date`, `Promise` и другие используют так называемые «внутренние слоты».

Это как свойства, но только для внутреннего использования в самой спецификациии. Например, `Map` хранит элементы во внутреннем слоте `[[MapData]]`. Встроенные методы обращаются к слотам напрямую, не через `[[Get]]/[[Set]]`. Таким образом, прокси не может перехватить их.

Почему это имеет значение? Они же всё равно внутренние!

Есть один нюанс. Если встроенный объект проксируется, то в прокси не будет этих «внутренних слотов», так что попытка вызвать на таком прокси встроенный метод приведёт к ошибке.

Пример:

```
let map = new Map();

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

proxy.set('test', 1); // будет ошибка
```

Внутри себя объект типа `Map` хранит все данные во внутреннем слоте `[[MapData]]`. Прокси не имеет такого слота. Встроенный метод `Map.prototype.set` пытается получить доступ к своему внутреннему свойству `this.[[MapData]]`, но так как `this=proxy`, то не может его найти и завершается с ошибкой.

К счастью, есть способ исправить это:

```
let map = new Map();

let proxy = new Proxy(map, {
  get(target, prop, receiver) {
    let value = Reflect.get(...arguments);
    return typeof value == 'function' ? value.bind(targ
  }
});

proxy.set('test', 1);
alert(proxy.get('test')); // 1 (работает!)
```

Сейчас всё сработало, потому что `get` привязывает свойства-функции, такие как `map.set`, к оригинальному объекту `map`. Таким образом, когда реализация метода `set` попытается получить доступ к внутреннему слоту `this.[[MapData]]`, то всё пройдёт благополучно.

> **Объект `Array` не использует внутренние слоты**
>
> Важным исключением является встроенный объект `Array`: он не использует внутренние слоты. Так сложилось исторически, ведь массивы были добавлены в язык очень давно.
>
> То есть описанная выше проблема не возникает при проксировании массивов.

Приватные поля

Нечто похожее происходит и с приватными полями классов.

Например, метод `getName()` осуществляет доступ к приватному полю `#name`, после проксирования он перестаёт работать:

```
class User {
  #name = "Гость";

  getName() {
    return this.#name;
  }
}

let user = new User();

user = new Proxy(user, {});

alert(user.getName()); // Ошибка
```

Причина всё та же: приватные поля реализованы с использованием внутренних слотов. JavaScript не использует `[[Get]]/[[Set]]` при доступе к ним.

В вызове `getName()` значением `this` является проксированный `user`, в котором нет внутреннего слота с приватными полями.

Решением, как и в предыдущем случае, является привязка контекста к методу:

```
class User {
  #name = "Гость";
```

```
  getName() {
    return this.#name;
  }
}

let user = new User();

user = new Proxy(user, {
  get(target, prop, receiver) {
    let value = Reflect.get(...arguments);
    return typeof value == 'function' ? value.bind(targ
  }
});

alert(user.getName()); // Гость
```

Однако, такое решение имеет ряд недостатков, о которых уже говорилось: методу передаётся оригинальный объект, который может быть передан куда-то ещё, и это может поломать всю функциональность проксирования.

Прокси != оригинальный объект

Прокси и объект, который проксируется, являются двумя разными объектами. Это естественно, не правда ли?

Если мы используем оригинальный объект как ключ, а затем проксируем его, то прокси не будет найден:

```
let allUsers = new Set();

class User {
  constructor(name) {
    this.name = name;
    allUsers.add(this);
  }
}

let user = new User("Вася");

alert(allUsers.has(user)); // true

user = new Proxy(user, {});

alert(allUsers.has(user)); // false
```

Как мы видим, после проксирования не получается найти объект `user` внутри множества `allUsers`, потому что прокси – это другой объект.

Прокси не перехватывают проверку на строгое равенство `===`

Прокси способны перехватывать много операторов, например `new` (ловушка `construct`), `in` (ловушка `has`), `delete` (ловушка `deleteProperty`) и так далее.

Но нет способа перехватить проверку на строгое равенство. Объект строго равен только самому себе, и никаким другим значениям.

Так что все операции и встроенные классы, которые используют строгую проверку объектов на равенство, отличат прокси от изначального объекта. Прозрачной замены в данном случае не произойдёт.

Отключаемые прокси

Отключаемый (revocable) прокси – это прокси, который может быть отключён вызовом специальной функции.

Допустим, у нас есть какой-то ресурс, и мы бы хотели иметь возможность закрыть к нему доступ в любой момент.

Для того, чтобы решить поставленную задачу, мы можем использовать отключаемый прокси, без ловушек. Такой прокси будет передавать все операции на проксируемый объект, и у нас будет возможность в любой момент отключить это.

Синтаксис:

```
let {proxy, revoke} = Proxy.revocable(target, handler)
```

Вызов возвращает объект с `proxy` и функцией `revoke`, которая отключает его.

Вот пример:

```
let object = {
  data: "Важные данные"
};

let {proxy, revoke} = Proxy.revocable(object, {});

// передаём прокси куда-нибудь вместо оригинального объ
alert(proxy.data); // Важные данные

// позже в коде
revoke();

// прокси больше не работает (отключён)
alert(proxy.data); // Ошибка
```

Вызов `revoke()` удаляет все внутренние ссылки на оригинальный объект из прокси, так что между ними больше нет связи, и оригинальный объект теперь может быть очищен сборщиком мусора.

Мы можем хранить функцию `revoke` в `WeakMap`, чтобы легко найти её по объекту прокси:

```
let revokes = new WeakMap();

let object = {
  data: "Важные данные"
};

let {proxy, revoke} = Proxy.revocable(object, {});

revokes.set(proxy, revoke);

// ..позже в коде..
revoke = revokes.get(proxy);
revoke();

alert(proxy.data); // Ошибка (прокси отключён)
```

Преимущество такого подхода в том, что мы не должны таскать функцию `revoke` повсюду. Мы получаем её при необходимости из `revokes` по объекту прокси.

Мы использовали `WeakMap` вместо `Map`, чтобы не блокировать сборку мусора. Если прокси объект становится недостижимым (то есть на него больше нет ссылок), то `WeakMap` позволяет сборщику мусора удалить его из памяти вместе с соответствующей функцией `revoke`, которая в этом случае больше не нужна.

Ссылки

- Спецификация: Proxy, Reflect.

- MDN: Proxy, Reflect.

Итого

Прокси – это обёртка вокруг объекта, которая «по умолчанию» перенаправляет операции над ней на объект, но имеет возможность перехватывать их.

Проксировать можно любой объект, включая классы и функции.

Синтаксис:

```
let proxy = new Proxy(target, {
  /* ловушки */
});
```

...Затем обычно используют прокси везде вместо оригинального объекта `target`. Прокси не имеет собственных свойств или методов. Он просто перехватывает операцию, если имеется соответствующая ловушка, а иначе перенаправляет её сразу на объект `target`.

Мы можем перехватывать:

- Чтение (`get`), запись (`set`), удаление (`deleteProperty`) свойства (даже несуществующего).
- Вызов функции (`apply`).
- Оператор `new` (ловушка `construct`).
- И многие другие операции (полный список приведён в начале статьи, а также в документации).

Это позволяет нам создавать «виртуальные» свойства и методы, реализовывать значения по умолчанию, наблюдаемые объекты, функции-декораторы и многое другое.

Мы также можем оборачивать один и тот же объект много раз в разные прокси, добавляя ему различные аспекты функциональности.

Reflect API создано как дополнение к Proxy. Для любой ловушки из `Proxy` существует метод в `Reflect` с теми же аргументами. Нам следует использовать его, если нужно перенаправить вызов на оригинальный объект.

Прокси имеют некоторые ограничения:

- Встроенные объекты используют так называемые «внутренние слоты», доступ к которым нельзя проксировать. Однако, ранее в этой главе был показан один способ, как обойти это ограничение.
- То же самое можно сказать и о приватных полях классов, так как они реализованы на основе слотов. То есть вызовы проксированных методов должны иметь оригинальный объект в качестве `this`, чтобы получить к ним доступ.
- Проверка объектов на строгое равенство `===` не может быть перехвачена.
- Производительность: конкретные показатели зависят от интерпретатора, но в целом получение свойства с помощью простейшего прокси занимает в несколько раз больше времени. В реальности это имеет значение только для некоторых «особо нагруженных» объектов.

Задачи

Ошибка при чтении несуществующего свойства

Обычно при чтении несуществующего свойства из объекта возвращается `undefined`.

Создайте прокси, который генерирует ошибку при попытке прочитать несуществующее свойство.

Это может помочь обнаружить программные ошибки пораньше.

Напишите функцию `wrap(target)`, которая берёт объект `target` и возвращает прокси, добавляющий в него этот аспект функциональности.

Вот как это должно работать:

```
let user = {
  name: "John"
};

function wrap(target) {
  return new Proxy(target, {
      /* ваш код */
  });
}

user = wrap(user);

alert(user.name); // John
alert(user.age); // Ошибка: такого свойства не существу
```

решение

Получение элемента массива с отрицательной позиции

В некоторых языках программирования возможно получать элементы массива, используя отрицательные индексы, отсчитываемые с конца.

Вот так:

```
let array = [1, 2, 3];

array[-1]; // 3, последний элемент
array[-2]; // 2, предпоследний элемент
array[-3]; // 1, за два элемента до последнего
```

Другими словами, `array[-N]` – это то же, что и `array[array.length - N]`.

Создайте прокси, который реализовывал бы такое поведение.

Вот как это должно работать:

```
let array = [1, 2, 3];

array = new Proxy(array, {
  /* ваш код */
});

alert( array[-1] ); // 3
alert( array[-2] ); // 2

// вся остальная функциональность массивов должна остат
```

решение

Observable

Создайте функцию `makeObservable(target)`, которая делает объект «наблюдаемым», возвращая прокси.

Вот как это должно работать:

```
function makeObservable(target) {
  /* ваш код */
}
```

Поделиться

Редактировать на GitHub

```
4
5  let user = {};
6  user = makeObservable(user);
7
8  user.observe((key, value) => {
9    alert(`SET ${key}=${value}`);
10 });
11
12 user.name = "John"; // выводит: SET name=John
```

Другими словами, возвращаемый `makeObservable` объект аналогичен исходному, но также имеет метод `observe(handler)`, который позволяет запускать `handler` при любом изменении свойств.

При изменении любого свойства вызывается `handler(key, value)` с именем и значением свойства.

P.S. В этой задаче ограничьтесь, пожалуйста, только записью свойства. Остальные операции могут быть реализованы похожим образом.

решение

Комментарии

перед тем как писать...