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». 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. 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. 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.

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`,
`[[OwnPropertyKeys]]`	`ownKeys`	Object.getOwnPropertyNames, Object.getOwnPropertyS

Значение по умолчанию с ловушкой «get»

Чтобы перехватить операцию чтения, handler должен иметь метод

get(target, property, receiver)

- **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).

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

```
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)
```

Прокси должен заменить собой оригинальный объект повсюду (numbers = new Proxy...)! 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**).

```
let numbers = [];
numbers = new Proxy(numbers, { // (*)
  set(target, prop, val) { // для перехвата записи свойства
    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 прокси вернула false)
alert("Интерпретатор никогда не доходит до этой строки (из-за ошибки в строке выше)");
```

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

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**
- 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.startsWith('_'));
  }
});
// ownKeys исключил _password
for(let key in user) alert(key); // name, затем: age
```

```
// аналогичный эффект для этих методов:
alert( Object.keys(user) ); // name,age
alert( Object.values(user) ); // Вася,30
```

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.

```
let user = { };
user = new Proxy(user, {
 ownKeys(target) { // вызывается 1 раз для получения списка свойств
  return ['a', 'b', 'c'];
 },
 getOwnPropertyDescriptor(target, prop) { // вызывается для каждого свойства
  return {
   enumerable: true,
   configurable: true
   /* ...другие флаги, возможно, "value: ..." */
  };
 }
});
alert( Object.keys(user) ); // a, b, c
```

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. 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:

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

user = new Proxy(user, {
 get(target, prop) {
  if (prop.startsWith('_')) {
   throw new Error("Отказано в доступе");
  } else {
   let value = target[prop];
   return (typeof value === 'function') ? value.bind(target) : value; // (*)
  }
 },
 set(target, prop, val) { // перехватываем запись свойства
  if (prop.startsWith('_')) {
   throw new Error("Отказано в доступе");
  } else {
   target[prop] = val;
   return true;
  }
 },
 deleteProperty(target, prop) { // перехватываем удаление свойства
  if (prop.startsWith('_')) {
   throw new Error("Отказано в доступе");
  } else {
   delete target[prop];
   return true;
  }
 },
 ownKeys(target) { // перехватываем попытку итерации
  return Object.keys(target).filter(key => !key.startsWith('_'));
 }
```

```
});

// "get" не позволяет прочитать _password
try {
  alert(user._password); // Error: Отказано в доступе
} catch(e) { alert(e.message); }

// "set" не позволяет записать _password
try {
  user._password = "test"; // Error: Отказано в доступе
} catch(e) { alert(e.message); }

// "deleteProperty" не позволяет удалить _password
try {
  delete user._password; // Error: Отказано в доступе
} catch(e) { alert(e.message); }

// "ownKeys" исключает _password из списка видимых для итерации свойств
for(let key in user) alert(key); // name
```

Метод [[Delete]] должен возвращать **true**, если значение было успешно удалено, иначе **false**.

«В диапазоне» с ловушкой «has»

Предположим, у нас есть объект range, описывающий диапазон:

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

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

```
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
```

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, delay(f, ms) возвращает функцию, которая передает вызовы f после ms миллисекунд. Вот предыдущая реализация, на основе функции:

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

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

// после обёртки вызовы sayHi будут срабатывать с задержкой в 3 секунды
sayHi = delay(sayHi, 3000);
sayHi("Вася"); // Привет, Вася! (через 3 секунды)
```

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.

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 чтение свойства length на исходную функцию

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

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]]`
...	...	...

Например:

```
let user = {};
Reflect.set(user, 'name', 'Вася');
alert(user.name); // Вася
```

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); // (2)
  }
```

```
});

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

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. Например, могут быть нюансы с наследованием (см. учебник). Здесь поотребуется третий аргумент **receiver** в get

Ограничения прокси:

- Встроенные объекты (**Map, Set, Date, Promise** и некоторые др.) используют так называемые «**внутренние слоты**», доступ к которым нельзя проксировать. Способ, как обойти ограничение - см. учебник. Объект **Array** не использует внутренние слоты.
- 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.
- 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.
- 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.

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 объекта...
alert(proxy.data); // Важные данные
// позже в коде
revoke();
// прокси больше не работает (отключён)
alert(proxy.data); // Ошибка
```