> Эта страница была переведена с английского языка силами сообщества. Вы тоже можете внести свой вклад,

Мета-программирование

С приходом ECMAScript 2015, в JavaScript введены объекты `Proxy` и `Reflect`, позволяющие перехватить и переопределить поведение фундаментальных процессов языка (таких как поиск свойств, присвоение, итерирование, вызов функций и так далее). С помощью этих двух объектов вы можете программировать на мета уровне JavaScript.

Объекты Proxy

Введённый в ECMAScript 6, объект `Proxy` позволяет перехватить и определить пользовательское поведение для определённых операций. Например, получение свойства объекта:

```
JS
var handler = {
  get: function (target, name) {
    return name in target ? target[name] : 42;
  },
};
var p = new Proxy({}, handler);
p.a = 1;
console.log(p.a, p.b); // 1, 42
```

Объект `Proxy` определяет `target` (в данном случае новый пустой объект) и `handler` - объект в котором реализована особая *функция-ловушка* `get`. "Проксированный" таким образом объект, при доступе к его несуществующему свойству вернёт не `undefined`, а числовое значение 42.

Дополнительные примеры доступны в справочнике `Proxy`.

Терминология

В разговоре о функциях объекта `Proxy` применимы следующие термины:

`handler (en-US)` (обработчик)

Объект - обёртка, содержащий в себе функции-ловушки.

ловушки (traps)

Методы, реализующие доступ к свойствам. В своей концепции они аналогичны методам перехвата(hooking) в операционных системах.

цель (target)

Объект, который оборачивается в Proxy. Часто используется лишь как внутреннее хранилище для Proxy объекта. Проверка на нарушение ограничений (invariants), связанных с нерасширяемостью объекта или неконфигурируемыми

свойствами объекта производится для конкретной цели.

неизменяемые ограничения (дословно Invariants - те что остаются неизменными)

Некоторые особенности поведения объекта, которые должны быть сохранены при реализации пользовательского поведения названы invariants. Если в обработчике нарушены такие ограничения, будет выброшена ошибка `TypeError`.

Обработчики и ловушки

В следующей таблице перечислены ловушки, доступные для использования в объекте `Proxy`. Смотрите подробные объяснения и примеры в документации (en-US).

Обработчик / ловушка	Перехватываемые методы	Неизменяемые ограничения
handler.getPrototypeOf() (en-US)	Object.getPrototypeOf() Reflect.getPrototypeOf() (en-US) __proto__ Object.prototype.isPrototypeOf() instanceof	• метод getPrototypeOf должен вернуть object или null. • если целевой объект target нерасширяем, метод Object.getPrototypeOf(proxy) должен возвращать тот же результат что и Object.getPrototypeOf(target).
handler.setPrototypeOf() (en-US)	Object.setPrototypeOf() Reflect.setPrototypeOf() (en-US)	если целевой объект target нерасширяем, значение параметра prototype должно быть равным значению возвращаемому методом Object.getPrototypeOf(target).
handler.isExtensible() (en-US)	Object.isExtensible() Reflect.isExtensible() (en-US)	Object.isExtensible(proxy) должно возвращать тоже значение, что и Object.isExtensible(target).
handler.preventExtensions() (en-US)	Object.preventExtensions() Reflect.preventExtensions() (en-US)	Object.preventExtensions(proxy) возвращает true только в том случае, если Object.isExtensible(proxy) равно false.
handler.getOwnPropertyDescriptor() (en-US)	Object.getOwnPropertyDescriptor() Reflect.getOwnPropertyDescriptor() (en-US)	• метод getOwnPropertyDescriptor должен возвращать object или undefined. • Свойство не может быть описано как несуществующее, если оно существует и является некофигурируемым, собственным свойством целевого объекта target. • Свойство не может быть описано как несуществующее, если оно существует как собственное свойство целевого объекта target и target не расширяем. • Свойство не может быть описано как существующее, если оно не существует как собственное свойство целевого объекта target и target не расширяем. • Свойство не может быть описано как неизменяемое, если оно не существует как

Обработчик / ловушка	Перехватываемые методы	Неизменяемые ограничения
		собственное свойство целевого объекта `target` или если оно существует и является изменяемым, собственным свойством целевого объекта `target`. • Значение возвращённое методом `Object.getOwnPropertyDescriptor(target)` может быть применено к целевому объекту через метод `Object.defineProperty` и это не вызовет ошибки.
`handler.defineProperty()` (en-US)	`Object.defineProperty()` `Reflect.defineProperty()`	• Новое свойство не может быть добавлено, если целевой объект не расширяем. • Нельзя добавить новое конфигурируемое свойство, или преобразовать существующее свойство в конфигурируемое, если оно не существует как собственное свойство целевого объекта или не является конфигурируемым. • Свойство не может быть неконфигурируемым, если целевой объект имеет соответствующее собственное, конфигурируемое свойство. • Если объект имеет свойство соответствующее создаваемому свойству, то `Object.defineProperty(target, prop, descriptor)` не вызовет ошибки. • В строгом режиме (`"use strict";`), если обработчик `defineProperty` вернёт `false`, это вызовет ошибку `TypeError`.
`handler.has()` (en-US)	Property query: `foo in proxy` Inherited property query: `foo in Object.create(proxy)` `Reflect.has()` (en-US)	• Свойство не может быть описано как несуществующее, если оно существует как собственное неконфигурируемое свойство целевого объекта. • Свойство не может быть описано как несуществующее, если оно существует как собственное свойство целевого объекта, и целевой объект является нерасширяемым.
`handler.get()` (en-US)	Property access: `proxy[foo]` and `proxy.bar` Inherited property access: `Object.create(proxy)[foo]` `Reflect.get()`	• Значение, возвращаемое для свойства, должно равняться значению соответствующего свойства целевого объекта, если это свойство является

Обработчик / ловушка	Перехватываемые методы	Неизменяемые ограничения
		доступным только для чтения, неконфигурируемым. • Значение, возвращаемое для свойства, должно равняться undefined, если соответствующее свойство целевого объекта является неконфигурируемым и обёрнуто в геттер и сеттер, где сеттер равен undefined.
handler.set()	Property assignment: proxy[foo] = bar and proxy.foo = bar Inherited property assignment: Object.create(proxy)[foo] = bar Reflect.set() (en-US)	• Нельзя изменить значение свойства на значение, отличное от значения соответствующего свойства целевого объекта, если это свойство целевого объекта доступно только для чтения, и является неконфигурируемым. • Нельзя установить значение свойства, если соответствующее свойство целевого объекта является неконфигурируемым, и обёрнуто в геттер и сеттер, где сеттер равен undefined. • В строгом режиме, возвращение false из обработчика set вызовет ошибку TypeError.
handler.deleteProperty()	Property deletion: delete proxy[foo] and delete proxy.foo Reflect.deleteProperty()	Свойство не может быть удалено, если оно существует в целевом объекте как собственное, неконфигурируемое свойство.
handler.enumerate()	Property enumeration / for...in: for (var name in proxy) {...} Reflect.enumerate()	Метод enumerate должен возвращать объект.
handler.ownKeys() (en-US)	Object.getOwnPropertyNames() Object.getOwnPropertySymbols() Object.keys() Reflect.ownKeys()	• Метод ownKeys должен возвращать список. • Типом каждого элемента в возвращаемом списке должен быть String или Symbol. • Возвращаемый список должен содержать ключи для всех неконфигурируемых, собственных свойств целевого объекта. • Если целевой объект является нерасширяемым, возвращаемый список должен содержать все ключи для собственных полей целевого объекта и больше никаких других значений.

Обработчик / ловушка	Перехватываемые методы	Неизменяемые ограничения
handler.apply() (en-US)	proxy(..args) Function.prototype.apply() and Function.prototype.call() Reflect.apply()	Ограничений нет.
handler.construct() (en-US)	new proxy(...args) Reflect.construct()	Обработчик должен возвращать Object.

Отзываемый Proxy

Метод Proxy.revocable() создаёт отзываемый объект Proxy. Такой прокси объект может быть отозван функцией revoke, которая отключает все ловушки-обработчики. После этого любые операции над прокси объектом вызовут ошибку TypeError.

JS

```
var revocable = Proxy.revocable(
  {},
  {
    get: function (target, name) {
      return "[[" + name + "]]";
    },
  },
);
var proxy = revocable.proxy;
console.log(proxy.foo); // "[[foo]]"

revocable.revoke();

console.log(proxy.foo); // ошибка TypeError
proxy.foo = 1; // снова ошибка TypeError
delete proxy.foo; // опять TypeError
typeof proxy; // "object", для метода typeof нет ловушек
```

Рефлексия

Reflect это встроенный объект, предоставляющий методы для перехватываемых операций JavaScript. Это те же самые методы, что имеются в обработчиках Proxy (en-US). Объект Reflect не является функцией.

Reflect помогает при пересылке стандартных операций из обработчика к целевому объекту.

Например, метод Reflect.has() (en-US) это тот же оператор in но в виде функции:

JS

```
Reflect.has(Object, "assign"); // true
```

Улучшенная функция apply

В ES5 обычно используется метод Function.prototype.apply() для вызова функции в определённом контексте (с определённым this) и с параметрами, заданными в виде массива (или массива-подобного объекта).

JS

```
Function.prototype.apply.call(Math.floor, undefined, [1.75]);
```

С методом `Reflect.apply` эта операция менее громоздка и более понятна:

```
JS
```

```
Reflect.apply(Math.floor, undefined, [1.75]);
// 1;

Reflect.apply(String.fromCharCode, undefined, [104, 101, 108, 108, 111]);
// "hello"

Reflect.apply(RegExp.prototype.exec, /ab/, ["confabulation"]).index;
// 4

Reflect.apply("".charAt, "ponies", [3]);
// "i"
```

Проверка успешности определения нового свойства

Метод `Object.defineProperty`, в случае успеха операции, возвращает объект, а при неудаче вызывает ошибку `TypeError`. Из-за этого определение свойств требует обработки блоком `try...catch` для перехвата возможных ошибок. Метод `Reflect.defineProperty`, в свою очередь, возвращает успешность операции в виде булева значения, благодаря чему возможно использование простого `if...else` условия:

```
JS
```

```
if (Reflect.defineProperty(target, property, attributes)) {
  // успех
} else {
  // что-то пошло не так
}
```

This page was last modified on 7 авг. 2023 г. by MDN contributors.