8 декабря 2022 г.

# Флаги и дескрипторы свойств

Как мы знаем, объекты могут содержать свойства.

До этого момента мы рассматривали свойство только как пару «ключ-значение». Но на самом деле свойство объекта гораздо мощнее и гибче.

В этой главе мы изучим дополнительные флаги конфигурации для свойств, а в следующей – увидим, как можно незаметно превратить их в специальные функции – геттеры и сеттеры.

## Флаги свойств

Помимо значения `value`, свойства объекта имеют три специальных атрибута (так называемые «флаги»).

- `writable` – если `true`, свойство можно изменить, иначе оно только для чтения.
- `enumerable` – если `true`, свойство перечисляется в циклах, в противном случае циклы его игнорируют.
- `configurable` – если `true`, свойство можно удалить, а эти атрибуты можно изменять, иначе этого делать нельзя.

Мы ещё не встречали эти атрибуты, потому что обычно они скрыты. Когда мы создаём свойство «обычным способом», все они имеют значение `true`. Но мы можем изменить их в любое время.

Сначала посмотрим, как получить их текущие значения.

Метод Object.getOwnPropertyDescriptor позволяет получить *полную* информацию о свойстве.

Его синтаксис:

```
let descriptor = Object.getOwnPropertyDescriptor(obj, propertyName);
```

`obj`

Объект, из которого мы получаем информацию.

`propertyName`

Имя свойства.

Возвращаемое значение – это объект, так называемый «дескриптор свойства»: он содержит значение свойства и все его флаги.

Например:

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

```

```
let descriptor = Object.getOwnPropertyDescriptor(user, 'name');

alert( JSON.stringify(descriptor, null, 2 ) );
/* дескриптор свойства:
{
  "value": "John",
  "writable": true,
  "enumerable": true,
  "configurable": true
}
*/
```

Чтобы изменить флаги, мы можем использовать метод Object.defineProperty.

Его синтаксис:

```
Object.defineProperty(obj, propertyName, descriptor)
```

**`obj`, `propertyName`**

Объект и его свойство, для которого нужно применить дескриптор.

**`descriptor`**

Применяемый дескриптор.

Если свойство существует, `defineProperty` обновит его флаги. В противном случае метод создаёт новое свойство с указанным значением и флагами; если какой-либо флаг не указан явно, ему присваивается значение `false`.

Например, здесь создаётся свойство `name`, все флаги которого имеют значение `false`:

```
let user = {};

Object.defineProperty(user, "name", {
  value: "John"
});

let descriptor = Object.getOwnPropertyDescriptor(user, 'name');

alert( JSON.stringify(descriptor, null, 2 ) );
/*
{
  "value": "John",
  "writable": false,
  "enumerable": false,
  "configurable": false
}
 */
```

Сравните это с предыдущим примером, в котором мы создали свойство `user.name` «обычным способом»: в этот раз все флаги имеют значение `false`. Если это не то, что нам нужно, надо присвоить им значения `true` в параметре `descriptor`.

Теперь давайте рассмотрим на примерах, что нам даёт использование флагов.

## Только для чтения

Сделаем свойство `user.name` доступным только для чтения. Для этого изменим флаг `writable`:

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

Object.defineProperty(user, "name", {
  writable: false
});

user.name = "Pete"; // Ошибка: Невозможно изменить доступное только для чтения
```

Теперь никто не сможет изменить имя пользователя, если только не обновит соответствующий флаг новым вызовом `defineProperty`.

> **Ошибки появляются только в строгом режиме**
>
> В нестрогом режиме, без `use strict`, мы не увидим никаких ошибок при записи в свойства «только для чтения» и т.п. Но эти операции всё равно не будут выполнены успешно. Действия, нарушающие ограничения флагов, в нестрогом режиме просто молча игнорируются.

Вот тот же пример, но свойство создано «с нуля»:

```
let user = { };

Object.defineProperty(user, "name", {
  value: "John",
  // для нового свойства необходимо явно указывать все флаги, для которых знач
  enumerable: true,
  configurable: true
});

alert(user.name); // John
user.name = "Pete"; // Ошибка
```

## Неперечислимое свойство

Теперь добавим собственный метод `toString` к объекту `user`.

Встроенный метод `toString` в объектах – неперечислимый, его не видно в цикле `for..in`. Но если мы напишем свой собственный метод `toString`, цикл `for..in` будет выводить его по умолчанию:

```
let user = {
  name: "John",
  toString() {
    return this.name;
  }
```

```
};

// По умолчанию оба свойства выведутся:
for (let key in user) alert(key); // name, toString
```

Если мы этого не хотим, можно установить для свойства `enumerable:false`. Тогда оно перестанет появляться в цикле `for..in` аналогично встроенному `toString`:

```
let user = {
  name: "John",
  toString() {
    return this.name;
  }
};

Object.defineProperty(user, "toString", {
  enumerable: false
});

// Теперь наше свойство toString пропало из цикла:
for (let key in user) alert(key); // name
```

Неперечислимые свойства также не возвращаются `Object.keys`:

```
alert(Object.keys(user)); // name
```

## Неконфигурируемое свойство

Флаг неконфигурируемого свойства ( `configurable:false` ) иногда предустановлен для некоторых встроенных объектов и свойств.

Неконфигурируемое свойство не может быть удалено, его атрибуты не могут быть изменены.

Например, свойство `Math.PI` – только для чтения, неперечислимое и неконфигурируемое:

```
let descriptor = Object.getOwnPropertyDescriptor(Math, 'PI');

alert( JSON.stringify(descriptor, null, 2 ) );
/*
{
  "value": 3.141592653589793,
  "writable": false,
  "enumerable": false,
  "configurable": false
}
*/
```

То есть программист не сможет изменить значение `Math.PI` или перезаписать его.

```
Math.PI = 3; // Ошибка, потому что writable: false

// delete Math.PI тоже не сработает
```

Мы также не можем изменить `writable`:

```
// Ошибка, из-за configurable: false
Object.defineProperty(Math, "PI", { writable: true });
```

Мы абсолютно ничего не можем сделать с `Math.PI`.

Определение свойства как неконфигурируемого – это дорога в один конец. Мы не можем изменить его обратно с помощью `defineProperty`.

**Обратите внимание: `configurable: false` не даст изменить флаги свойства, а также не даст его удалить. При этом можно изменить значение свойства.**

В коде ниже свойство `user.name` является неконфигурируемым, но мы все ещё можем изменить его значение (т.к. `writable: true`).

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

Object.defineProperty(user, "name", {
  configurable: false
});

user.name = "Pete"; // работает
delete user.name; // Ошибка
```

А здесь мы делаем `user.name` «навечно запечатанной» константой, как и встроенный `Math.PI`:

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

Object.defineProperty(user, "name", {
  writable: false,
  configurable: false
});

// теперь невозможно изменить user.name или его флаги
// всё это не будет работать:
user.name = "Pete";
delete user.name;
Object.defineProperty(user, "name", { value: "Pete" });
```

**Ошибки отображаются только в строгом режиме**

В нестрогом режиме мы не увидим никаких ошибок при записи в свойства «только для чтения» и т.п. Эти операции всё равно не будут выполнены успешно. Действия, нарушающие ограничения флагов, в нестрогом режиме просто молча игнорируются.

## Метод Object.defineProperties

Существует метод Object.defineProperties(obj, descriptors), который позволяет определять множество свойств сразу.

Его синтаксис:

```
Object.defineProperties(obj, {
  prop1: descriptor1,
  prop2: descriptor2
  // ...
});
```

Например:

```
Object.defineProperties(user, {
  name: { value: "John", writable: false },
  surname: { value: "Smith", writable: false },
  // ...
});
```

Таким образом, мы можем определить множество свойств одной операцией.

## Object.getOwnPropertyDescriptors

Чтобы получить все дескрипторы свойств сразу, можно воспользоваться методом Object.getOwnPropertyDescriptors(obj).

Вместе с `Object.defineProperties` этот метод можно использовать для клонирования объекта вместе с его флагами:

```
let clone = Object.defineProperties({}, Object.getOwnPropertyDescriptors(obj));
```

Обычно при клонировании объекта мы используем присваивание, чтобы скопировать его свойства:

```
for (let key in user) {
  clone[key] = user[key]
}
```

...Но это не копирует флаги. Так что если нам нужен клон «получше», предпочтительнее использовать `Object.defineProperties` .

Другое отличие в том, что `for..in` игнорирует символьные и неперечислимые свойства, а `Object.getOwnPropertyDescriptors` возвращает дескрипторы *всех* свойств.

## Глобальное запечатывание объекта

Дескрипторы свойств работают на уровне конкретных свойств.

Но ещё есть методы, которые ограничивают доступ ко *всему* объекту:

**Object.preventExtensions(obj)**

Запрещает добавлять новые свойства в объект.

**Object.seal(obj)**

Запрещает добавлять/удалять свойства. Устанавливает `configurable: false` для всех существующих свойств.

**Object.freeze(obj)**

Запрещает добавлять/удалять/изменять свойства. Устанавливает `configurable: false, writable: false` для всех существующих свойств.

А также есть методы для их проверки:

**Object.isExtensible(obj)**

Возвращает `false`, если добавление свойств запрещено, иначе `true`.

**Object.isSealed(obj)**

Возвращает `true`, если добавление/удаление свойств запрещено и для всех существующих свойств установлено `configurable: false`.

**Object.isFrozen(obj)**

Возвращает `true`, если добавление/удаление/изменение свойств запрещено, и для всех текущих свойств установлено `configurable: false, writable: false`.

На практике эти методы используются редко.

Предыдущий урок | Следующий урок

Поделиться

Карта учебника

## Комментарии

- Если вам кажется, что в статье что-то не так - вместо комментария напишите на GitHub.
- Для одной строки кода используйте тег `<code>`, для нескольких строк кода — тег `<pre>`, если больше 10 строк — ссылку на песочницу (plnkr, JSBin, codepen...)
- Если что-то непонятно в статье — пишите, что именно и с какого места.

**1.sohr.1**
24 дня назад

Не можешь решать задачки на codewars,Leetcode и много вопросов в процессе обучения
Тогда переходи в Тг чат @communityforprogg

0 1 Ответить

**Юрий**
месяц назад

⚡ **Хочешь понять флаги и дескрипторы свойств?**
Тогда переходи в ТГ-блог «Джун на фронте»!

👨‍💻 Автор - системный администратор, который с декабря 2021 года освоил HTML, CSS, JS, Vue, Nuxt, React Native, MongoDB и Node.js.

Следи за моим путем в мир разработки: от новичка до создателя 🤖 Телеграм-бота для автоматической отправки откликов!

💡 Ежедневно ты найдешь полезные ответы на насущные вопросы, анализ рынка вакансий и советы от человека, прошедшего этот путь.

**Вбивайте «Джун на фронте» и присоединяйтесь к нам!**

0 2 Ответить

**Дмитрий Абрасимовский**
4 месяца назад

Как по мне, при клонировании объекта данным способом:
let clone = Object.defineProperties({}, Object.getOwnPropertyDescriptors(obj));

...пропустили важную особенность, что клонируются только те свойства, в которых флаг enumerable == true

1 1 Ответить

**Maksim Vashkevich** → Дмитрий Абрасимовский
4 месяца назад

только что проверил твое утверждение - это неправда.

0 0 Ответить

**Дмитрий Абрасимовский** → Maksim Vashkevich
3 месяца назад

Возможно ты где-то ошибся, вот попробуй поиграться. В клон попадает только то свойство, где enumerable == true. Попробуй сначала по дефолту, потом явно укажи еще у любого свойства true во флаге enumerable

```
const obj = {};

Object.defineProperties(obj, {
name: {value: 'Дмитрий', enumerable: true},
age: {value: 27},
car: {value: 'Mazda', configurable: true}
});

const clone = Object.defineProperties({}, Object.getOwnPropertyDescriptors(obj));

console.log(clone);
```

1 0 Ответить

**Даниил** → Дмитрий Абрасимовский
3 месяца назад

Интересный пример, который может легко запутать.
При копировании таким образом

```
Object.defineProperties({}, Object.getOwnPropertyDescriptors(obj));
```

Всё же все свойства объекта obj будут клонированы (в том числе и те, которые указаны с флагом enumerable = false).
В твоём примере будет выведено в консоль следующее:

```
{ name: 'Дмитрий' }
```

Что как бы должно наталкивать на мысль, что действительно клонируются свойства только с флагом enumerable = true, однако есть проблема - результат выводится в консоль с помощью console.log.
Это и могло ввести в заблуждение, потому что после клонирования доступ к свойству age и car (которые указаны с флагом enumerable = false) остаётся (они клонируются).

Следующий код подтверждает мои слова:

показать больше

4 0 Ответить

**Rail Batyrshin** → Даниил
месяц назад

Можно сверить все свойства клона с оригинальным объектом и все там верно, сейчас перепроверил с помощью:
`Object.getOwnPropertyDescriptor()`

0 0 Ответить

**SineYlo**

год назад edited

Посмотрел какую-то часть комментариев на тему того, что в статье допущена ошибка в описании к атрибуту `[[Configurable]]`. И по факту это действительно так. Если мы перейдём в спецификацию в раздел 6.1.7.1 Property Attributes (`https://tc39.es/ecma262/#sec-property-attributes`), то там для атрибута `[[Configurable]]`, написано следующее (часть убрана для простоты): «Если значение `false`, то попытки удалить свойство, а также внести изменения в атрибуты (кроме существующего `[[Value]]` и `[[Writable]]` на значение `false`) завершатся неудачей». Надеюсь теперь всё понятно и по этому пункту больше вопросов не будет :).

6 0 Ответить

**Aleksandr S** → SineYlo

4 месяца назад

Спасибо мужик. Сидел пол часа голову себе ломал, почему так происходит.

0 0 Ответить

**ULADZIMIR MARTYNCHYK**

год назад

Сколько не находил ресурсов для обучения по JS этот единственный, который поясняет последовательно и вполне ясно. Не хватает таких ресурсов по HTML & CSS. ***Уважение ребятам!***

6 0 Ответить

**Sergey RJS**

год назад edited

Написал функцию для глубокого клонирования объекта со всеми флагами:

```
function flagsClone(obj) {
if (obj === null || typeof obj === 'function') return obj;

let result = {};
if (Array.isArray(obj)) result = []; // можно клонировать массивы

for (let key in Object.getOwnPropertyDescriptors(obj)) { // Работает даже если enumerable: false
Object.defineProperty(result, key, Object.getOwnPropertyDescriptor(obj, key));
if (typeof obj[key] === 'object') {
result[key] = flagsClone(obj[key]);
}
}

return result;
}
```

0 0 Ответить

**Maxim** → Sergey RJS

год назад edited

Попытка сделать клонирование с учетом флагов похвальная.

Но есть проблемы:
- Вы не учитываете значение null которое является объектом поэтому ваш код не сможет запустить рекурсию и споткнется об него.
- Циклические структуры, это наверное самый главный камень преткновения всех реализаторов перечислений объекта, попытка реализовать перечисление циклической структуры в разных местах приведет к бесконечному циклу. В результате чего у вас будет переполнен стек.
- Экзотические объекты или объекты с нестандартным поведением типа Set или Date копирование таких объектов вернет вам пустые объекты.
- Ну и ко всему прочему у вас не учитывается перечисление прототипа, это конечно необязательно, но как возможность было бы неплохо.

Это хорошая болванка, для мощного итератора свойств. Недавно сам переписывал старые решения таких штук, можете здесь глянуть пример. Там прототипы и экзотика тоже не учитывается, но их можно легко добавить (надо будет найти время и дописать как их так и документацию к этому итератору)

показать больше

3 1 Ответить

**Sergey RJS** → Maxim

год назад edited

flagsClone, оказывается, не работает как надо, если попадается объект с флагом *writable: false* (`TypeError: Cannot assign to read only property...`).

Написал новую функцию с учетом всех ваших замечаний:

```
function duplicate(obj) {
let visited = new Set();

return function makeClone(obj) {
if (obj === null || typeof obj !== 'object' || visited.has(obj)) return obj;

if (obj instanceof Date) return new Date(obj.getTime());

visited.add(obj);

if (obj instanceof Map) {
let map = new Map();
```

показать больше

0 0 Ответить

**Maxim** → Sergey RJS

год назад edited

Забавно видеть конструкции **__proto__** и **Object.getPrototypeOf**

**Map/Set/Date** и прочая экзотика должна быть независимо расширяема через параметры либо еще как-то удобным способом. И таким образом можно будет расширить реализацию на клонирование любого типа объекта.

В код сильно не вглядывался, но использование **Set** для посещенных узлов - хорошее решение, это теперь **DFS** алгоритм.

Касательно передачи прототипов, я бы не стал этого делать, в клонировании прототипы не нуждаются. Вот если бы задача стояла перечислить свойства прототипов - тогда да, а так я считаю это нецелесообразным.

Не изучал механику **structuredClone**, функция свежая. Но клонировать свойства аксессоры достаточно просто - **const temp1 = Object.getOwnPropertyDescriptor(window.location, "href")**. То есть получаем дескриптор и записываем эти данные в определение свойства другого объекта (клонируемого) так: **Object.defineProperty({}, "href", temp1)**

В копировании свойств не должно быть использования вычисляемых свойств через **[]**, так как с

показать больше

1 1 Ответить

**Sergey RJS** → Maxim

год назад

С прототипами действительно забавно получилось. Хотел уйти от геттера/сеттера __proto__ в пользу новых методов, но по привычке написал по-старому :)

Не понял, как именно должны расширяться экзотические объекты. Речь ведь не о вложенном клонировании для них?

Я решил клонировать все, что только можно, чтобы клон получился максимально приближенным к реальному объекту, поэтому и прототипы клонируются, да и аргументы против я не нашел. Сейчас проверил встроенную реализацию (structuredClone), там прототипы вообще никак не клонируются. Вопрос: перечислять свойства прототипов имеется ввиду - записать их в клонируемый объект без наследования?

С геттерами и сеттерами работает, спасибо. Я почему-то думал, что их можно назначать только при создании объекта.
Теперь можно заменить тело последнего цикла в функции (4 строчки) на:

```
let {writable, enumerable, configurable, get, set} = descriptor;

cloneObj[key] = makeClone(descriptor.value);
```

показать больше

0 0 Ответить

**Maxim** → Sergey RJS

год назад edited

Ну я так и понял что по привычке, бывает :)

Типы расширять не обязательно, можно реализовать наиболее часто используемые типы и забыть об этом. Но можно как я сказал сделать интеграцию через параметры, этим я имел ввиду что внутренняя реализация клонирования имеет доступ к параметрам функции и взаимодействует с ними определенным образом. Я писал как-то реализацию клонирования и там я добавил интеграцию через параметры. Посмотрите если интересно.

Прототипы как я говорил не имеет смысла клонировать, это разрыв связи с цепочкой прототипов

от конструкторов. Клонировать-то можно, но нецелесообразно. Когда я говорил про перечисление, я имел ввиду создание алгоритма, который будет перечислять все свойства объекта - своеобразный итератор свойств, там имело бы смысл перечислить уникальные свойства из прототипов.

**Illegal Constructor** также как и **Illegal Invocation** - внутренние ошибки, конструкторная связана с тесной связью DOM узлов и как следствие не все казалось бы конструкторы могут быть напрямую вызваны, что касается второй ошибки то она практически ничем не отличается и проявляется при потере контекста, например:

```
const hrefDescriptor = Object.getOwnPropertyDescriptor(window.location, "href");
```

показать больше

1 0 Ответить

**Sergey RJS** → Maxim

год назад

Расширение через параметры, как я понял, у вас работает так, что экзотические объекты клонируются только, если это явно указать, передав объект с функциями клонирования в `supplementalTypes`. Эти функции можно "расширить", добавив туда свой код чтобы экзотические объекты делали что-то еще / имели дополнительные свойства.

То есть правильнее было бы записывать свойства из прототипа в клонируемый объект (но только если их нет в исходном объекте).

Насчет того, что нельзя передавать descriptor.value - я имел ввиду изначальную value. Конечно можно что-то такое (деструктуризация как в вашем коде):

```
if (descriptor.get || descriptor.set) {
Object.defineProperty(cloneObj, key, descriptor);
continue;
}

let value = makeClone(descriptor.value);
Object.defineProperty(cloneObj, key, {...descriptor, value});
```

Такая запись читается даже лучше. `value` - уже клонированный объект и его надо добавлять дополнительно перенесенная descriptor.value

показать больше

0 0 Ответить

**Maxim** → Sergey RJS

год назад edited

Не понятно о чем речь.
*То есть правильнее было бы записывать свойства из прототипа в клонируемый объект (но только если их нет в исходном объекте).*

Про ошибки я уже высказался - это внутренняя реализация интерфейсов и нарушение их использования вызывает ошибки, вы не можете вызывать **alert** в контексте другого объекта.

Пример:

```
function ownAlert(arg) {
    if (!(this instanceof Window)) throw TypeError("Illegal Invocation");
    alert(arg);
}
```

Это практически то о чем ты спрашиваешь про ошибки.

0 0 Ответить

**Sergey RJS** → Maxim

год назад

Я имел ввиду следующее:

```
let obj2 = {
prop1: 'value1',
prop2: 'valueNotToClone',
}

let obj1 = {
prop2: 'value2',
__proto__: obj2,
};

let clone = makeClone(obj1);
console.log(clone); // {prop1: 'value1', prop2: 'value2'}
```

Скопированы только свойства прототипа (в данном случае - prop1) , которых нет в объекте obj1.

0 0 Ответить

**Maxim** → Sergey RJS

год назад

А вот про что. Нет я не считаю что данное действие правильно, прототип это прототип, это общие свойства экземпляров конструкторов. Это уже мутация вместо копирования, так как копирование это клонирование структуры как есть. А двигать свойства как захочется это совсем другое.

P.S Копирование proxy объектов невозможно :)

0 0 Ответить

**Sergey RJS** → Maxim

год назад

А как тогда правильно? Можно клонировать свойства прототипа в другой объект, потом добавить ссылку [[Prototype]] на этот объект. Так никто не должен пострадать)

Конечно, ведь нельзя узнать, какие из ловушек были использованы. Только если в консоли свойство [[Handler]] найти. Но это внутреннее свойство.

0 0 Ответить

**Maxim** → Sergey RJS

год назад edited

Ну ты опять хочешь клонировать прототипы, но в js это не имеет смысла и выше я уже писал почему. Прототипы это **общие** свойства и методы. И делать суррогатный объект, который будет уникальным прототипом бессмысленно... но это можно сделать, но бессмысленно.

Не в этом дело, а в том что proxy объект работает совсем иначе, другие внутренние поля в отличии от обычных объектов. Если интересно можно изучить спецификацию по данному вопросу.

1 0 Ответить

**Sergey RJS** Sergey RJS

год назад edited

А лучше переписать так.

```
if (!descriptor.get && !descriptor.set) {
cloneObj[key] = makeClone(descriptor.value);
delete descriptor.value;
}

Object.defineProperty(cloneObj, key, descriptor);
// нельзя передавать descriptor.value в клонируемый объект, иначе value перезапишется и получится поверхностная копия
```

В том коде, что я написал за полчаса до этого хоть ошибок и не было, но геттеру сначала присваивалась value = undefined, вызывалась функция makeClone для него, потом value заменялась на get или set.

0 0 Ответить

**Sergey RJS** Maxim

год назад edited

Спасибо, функцию поправил, чтобы значение null учитывалось.
Вопрос, как реализовать клонирование циклических структур в простейшем случае (без прототипов, флагов и тд)? Для меня это пока загадка

0 0 Ответить

**Maxim** Sergey RJS

год назад

Чтобы решить проблему циклических структур нужно почитать про графы и их обход.

0 0 Ответить

**A** **Антон** Maxim

год назад

function clone(obj){
return Object.defineProperties({}, Object.getOwnPropertyDescriptors(obj));
}
Такой простой вариант тоже поддерживает вложенное клонирование, разве не так?

0 0 Ответить

**Maxim** Антон

год назад edited

Это вы ловко придумали, однако это не решает проблему вложенного клонирования, вам нужно данный метод вызывать в цикле для дескрипторов имеющих в качестве значения объект. А это уже приводит к тому что структура такого объекта может иметь сложные перекрестные связи внутри.

0 0 Ответить

**V** **Vladyslav Makarenko**

год назад

**Note:** методы freeze и seal не работают на вложенные объекты, а также не оказывает эффекта на

Object.freeze() работает на вложенные объекты, а также не оказывает эффекта на содержимое объектов Map, Date и Set.

```
// Создаём объект
const person = {
fullName: 'Some Name',
info: {
job: 'Software engineer',
grade: 'Senior'
}
}

// Замораживаем объект
Object.freeze(person);

// Повышаем нашего программиста до "Staff" :)
person.info.grade = 'Staff';
```

показать больше

9 1 Ответить

**Maxim** Vladyslav Makarenko

год назад

Обоснуйте свой минус, в чем я не прав?

1 0 Ответить

**Maxim** Vladyslav Makarenko

год назад edited

Map, Set, Date это конструкторы объектов особого не экзотического типа. Они работают совершенно другим образом. Поэтому вы ничего не можете к ним применить из этого арсенала к структуре, которую они создают. Вы б еще попробовали скопировать эти объекты :)

1 0 Ответить

**Sleepy Asura**

год назад edited

Почти на 100% уверен, что все челики, которые продвигают свои блоги, сидят в гордом одиночестве, зато с кучкой статей и видео, чисто для себя, для личной мотивации. Как по мне, пустая трата времени. Лишь бы не учиться усердно.

5 0 Ответить

**Н** **Наталья** Sleepy Asura

10 месяцев назад

Пока пишешь блог, сам что-то запомнишь))) лучший способ выучить предмет - его преподавать))

0 0 Ответить

**В** **Влад** Sleepy Asura

год назад

не знаю, мне норм)

0 0 Ответить

**Сергей М.**
год назад

```
let Math = {
PI: 19
};
let descriptor = Object.getOwnPropertyDescriptor(Math, 'PI');

alert( JSON.stringify(descriptor, null, 2 ) );
/*
{
"value": 19, _______ чиво, пля???
"writable": false,
"enumerable": false,
"configurable": false
}
*/
```

0 0 Ответить

**Dmitriy Sparrow** → Сергей М.
год назад

Думаю ты уже понял, но на всякий - Не false, а true. Math.PI и твой Math.PI это разные объекты. Так как ты создал объект обычным способом, то конечно значения свойств этого объекта можно изменять (потому-что все true). Или у тебя false? Просто я вбил твой код и у меня true

2 0 Ответить

**Михаил Виноградов**
2 года назад edited

⚡Ну чё, народ, погнали?⚡
Реально ли **изучить javascript за 7 месяцев** и трудоустроиться?
Вот я решил проверить и веду свой блог **Джаваскриптизёр** на ютубе, где буду выкладывать видео каждую неделю на протяжении 7 месяцев.

💪💪Вторая неделя обучения завершена, второй видос тоже готов💪💪

Если тебе тоже интересен джаваскрипт, присоединяйся:
Ютуб: **Джаваскриптизёр**
ТГ: **@javascriptizerr**

🔥Удачи всем нам🔥

0 9 Ответить

**Sashka Sanek**
2 года назад

В нестрогом режиме мы не увидим никаких ошибок при записи в свойства «только для чтения» и т.п

Попытка переопределить configurable в не строгом режиме все таки выдает ошибку

0 0 Ответить

**Сергей М.** → Sashka Sanek

год назад

Это если conigurable было false изначально, иначе один раз можно

0 0 Ответить

**Иван** → Sashka Sanek

2 года назад edited

Ты не так понял. Ошибки в нестрогом режиме не будет, если ты при configurable: false, например удалишь свойство (просто проигнорируется).
А вот если configurable: false, его уже нельзя вернуть на true - тут не зависит от вкл/выкл strict mode*потому что defineProperty не работает с неконфигурируемыми свойствами.*

0 0 Ответить

**Ігор Українець**

2 года назад edited

(Прошляпил здесь обновление descriptor. Спасибо Yuri за раз'ьяснение и потраченное время)

Все однозначно. Флаг configurable: false запрещает удаление свойства и изменение флагов

```
let user = {};

Object.defineProperty(user, "name", {
    value: "John",
    writable: true
});                                    // не указанньім явно флагам присваивается значення false.

let descriptor = Object.getOwnPropertyDescriptor(user, 'name');

alert( JSON.stringify(descriptor, null, 2 ) );
/*
{
    "value": "John",
    "writable": true,
    "enumerable": false
```

показать больше

0 0 Ответить

**Yuri** → Ігор Українець

2 года назад edited

А "без изменений" только потому, что кое-кто не понял, что объект `descriptor` не изменится автоматически после изменения атрибута объекта `user`, и перед последним алертом не вставил строку `descriptor = Object.getOwnPropertyDescriptor(user, 'name');`

Еще и минусов мне налепил...
facepalm.jpg

3 0 Ответить

**Ігор Українець** → Yuri

2 года назад

Каюсь!)) Ну Вы ведь могли написать яснее:
"Замачен типа один баг. При установленном configurable: false допускается лишь единоразовое изменение writable, причем лишь в направлении true => false"

Минусы отлепил, спасибо за прояснение ситуации - теперь єту особенность запомню надолго)

0 0 Ответить

**Yuri** → Ігор Українець

2 года назад

Как вы уже заколебали...

```
'use strict';

const toConsole = obj => console.log(JSON.stringify(Object.getOwnPropertyDescriptors(obj), null, 4)
let user = {};

Object.defineProperty(user, 'name', {
        value: 'Joe',
        writable: true,
        enumerable: true,
        configurable: false,
});
toConsole(user);

console.log('Меняем атрибут writable при configurable: false\n');

Object.defineProperty(user, 'name', {
        value: 'Boris',
        writable: false, /* !!! */
        enumerable: true,
        configurable: false,
});
toConsole(user);

console.log('Дальше все, невозможно изменить ни один атрибут');
console.log(`Ошибки нет, если значения "изменяются" на такие же
(но только с помощью Object.defineProperty)\n`);

Object.defineProperty(user, 'name', {
        value: 'Boris',
        writable: false,
        enumerable: true,
        configurable: false,
});
toConsole(user);

// user.name = 'Jens'; /* error */

// user.name = 'Boris'; /* error */

// Object.defineProperty(user, 'name', {
//      value: 'Boris',
//      writable: true, /* error */
//      enumerable: true,
//      configurable: false,
// });
```

2 0 Ответить

**Ігор Українець** → Yuri

2 года назад

Благодарю! Вы правы! А я был невнимателен. Извините за потраченное время.

0 0 Ответить

**M** **Max**

2 года назад edited

Коллеги, возник вопрос по цитате из урока:

> Определение свойства как неконфигурируемого – это дорога в один конец. Мы не сможем отменить это действие, потому что **defineProperty не работает с неконфигурируемыми свойствами**.

На практике, defineProperty еще как работает с неконфигурируемыми свойствами.
Создаем объект и делаем его свойство неконфигурированным (переводим флаг configurable в положение false) :

```
'use strict'

let obj = {
    name: 'John',
}
Object.defineProperty(obj, 'name', { configurable: false });
let descriptor = Object.getOwnPropertyDescriptor(obj, 'name');
console.log(descriptor);
/*
{
  value: 'John',
  writable: true,
  enumerable: true,
  configurable: false
}
*/
```

Теперь попробуем взаимодействовать со свойством name снова через defineProperty:

```
'use strict'

Object.defineProperty(obj, 'name', { value: 'Pete', writable: false})
descriptor = Object.getOwnPropertyDescriptor(obj, 'name');
console.log(descriptor);
/*
{
  value: 'Pete',
  writable: false,
  enumerable: true,
  configurable: false
}
*/
```

Как видно из примера, даже если свойства неконфигурируемые, через defineProperty можно спокойно взаимодействовать с флагом writable и изменять их значение value.

2 0 Ответить

**С** **Сергей Мельников** Max
2 года назад edited

поэкспериментировал, вроде бы получается так, что с configurable=true любое из двух других можно ужесточить, заменив true на false, но нельзя ослабить, заменив false на true
*UPD посмотрел ниже коммент Юрия, там всё указано (enumerable оказывается нельзя менять даже с true на false)

1 0 Ответить

**sobanja** Max
2 года назад

Думаю, что тут речь не про изменение других флагов или свойства, а про изменение самого config. Ты не сможешь изменить именно его значение. Попробуй задать в defineProperty configurable false, а потом заменить его на true. Тебе выдаст ошибку.

0 0 Ответить

**Yuri**

В случае, когда

```
configurable: false
```

И

```
writable: true
```

,
менять МОЖНО value (сколько угодно раз, пока writable остается true) и writable (один раз - на false).

Поправьте, если ошибаюсь.

8 0 Ответить

**Evgenii Sapozhkov** → Yuri
2 года назад

configurable влияет только на настройку enumerable и самого configurable свойств и на удаление самого свойства. Writable свойство никак не затрагивается при изменении configurable

0 0 Ответить

**Yuri** → Evgenii Sapozhkov
2 года назад

Моя реплика призвана исправить неточность авторов:

> configurable – если true, свойство можно удалить, а эти атрибуты можно изменять, иначе этого делать нельзя

И при изменении атрибута *configurable* (на *false*, других вариантов нет) атрибут *writable* не меняется непосредственно, однако появляются особенности, о которых я и написал. Так что *configurable* и *writable* связаны очень тесно.

1 0 Ответить

**Evgenii Sapozhkov** → Yuri
2 года назад

Не появляется никаких особенностей, про которые ты говорил. В статье не указано, что configurable никак не аффектит writable, поэтому writable ведет себя абсолютно независимо от cofigurable, это касается как изменения флага, так и самого значения свойств.

> configurable – если true, свойство можно удалить, а эти атрибуты можно изменять, иначе этого делать нельзя

А здесь допущена ошибка, что configurable отвечает только за самого себя и enumerable и за возможность удаления свойства.

0 0 Ответить

**Yuri** → Evgenii Sapozhkov
2 года назад

Перечитай этот абзац статьи (о *configurable*) еще раз. Первый мой комментарий к нему.

Комментарий **Denis** чуть ниже, наконец. Я устал, честно...

0 0 Ответить

**R** **Rafa** → Yuri
2 года назад

если я правильно понял, теперь объект с данным дескриптором будет навечено writable true, и мы не сможем изменить его свойства, ни удалить его

0 0 Ответить

 **Yuri** → Rafa
2 года назад

Когда *configurable: false*, то *writable* "навечно" **можно** установить только в состояние *false*. А можно и не устанавливать.

2 0 Ответить

**D** **Denis** → Yuri
2 года назад

Да, заметил такую же особенность. При этом даже если enumerable изначально стоит в true, то его нельзя поменять на false (в случае если configurable = false). Однако writable можно поставить в false вне зависимости от флага configurable

5 0 Ответить

**M** **Margarita Kisil**
2 года назад edited

не понимаю фразу : "Если свойство существует, defineProperty обновит его флаги"

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

Object.defineProperty(user, "name", {});

let descriptor = Object.getOwnPropertyDescriptor(user, 'name');

alert( JSON.stringify(descriptor, null, 2 ) );
```

свойство существует, но флаги не обновляются (все - тру). В чем тут дело?

1 0 Ответить

 **Алексей Обухов** → Margarita Kisil
2 года назад

проставляется false только при создании, при изменении свойств значения по умолчанию не проставляется. допустим, вы хотите сделать writable: false. вы же не хотите, чтобы configurable тоже стал false.

2 0 Ответить

**R** **Ragima Bagirova**
2 года назад

Метод Object.assign(obj) смог создать клон объекта со всеми соответствующими значениями флагов. Так что еще один легкий способ ))

0 1 Ответить

**К** **Кирилл** Ragima Bagirova
2 года назад edited

Нет, метод Object.assign(obj) не создает клон объекта с соответствующими дескрипторами-свойств.

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

Object.defineProperty(user, "name", {writable: false});
let clone = Object.assign({}, user);

alert( JSON.stringify(Object.getOwnPropertyDescriptors(clone), null, 2) );
```

```
"{
  'name': {
    'value': 'John',
    'writable': true,
    'enumerable': true,
    'configurable': true
  }
}"
```

5 0 Ответить

**Р** **Радислав Ялилов**
2 года назад

Почему-то если сначала применить к объекту seal, а затем freeze, то значение writable изменится, хотя configurable уже false

1 0 Ответить

 **EXO** Радислав Ялилов
2 года назад

configurable защищает от изменения используя defineProperty(ies)

1 0 Ответить

**E** **Evgenii Sapozhkov** EXO
2 года назад

не только, он так же защищает от применение оператора delete на свойство

0 0 Ответить

 **Сергей**
2 года назад

Неплохо было бы добавить примечание, что "распечатать" объект обратно уже не получится. Не обратных свойств для freeze и seal

2 0 Ответить

**Pavel Kurchavov** → Сергей
2 года назад

Так это понятно из лекции, выше говорилось, что флаг configurable в true обратно не установить, а эти методы ставят его в false

6 0 Ответить

**Бухгалтер** → Pavel Kurchavov
2 года назад

это никак не следует, что это должно касаться новых свойств объекта и самого объекта, иначе сам объект имеет свойство configurable, а про это не говорится здесь.

0 0 Ответить

**bi_zi**
2 года назад

Может кто ни будь привести примеры как это использовать и для чего вообще это может понадобится начинающему программисту или это уже более высокий уровень кода?

4 0 Ответить

**Николай К** → bi_zi
2 года назад edited

Это может понадобится если будете писать на нативном JS, на котором почти никто не пишет. И то в редких случаях т.к. обычно на приватные свойства просто делают геттер через замыкания.
В реальности это всё будет писаться с помощью typeScript где реализованы приватные переменные, например:

```
class TaskStore {
constructor() {
//.... какой-то код
}

private _tasks = [];

get tasks() {
return this._tasks
}
```

И в редких случаях на собесе могут спросить.

0 0 Ответить

**sudo mkfs -t vfat /dev/sdb1** → Николай К
2 года назад

Со всем уважением, но я бы не делал таких громких умозаключений что на чистом js как вы выразились никто не пишет, являясь частью огромного коммерческого проекта на ноде с тонной app engine/k8 сервисов, клауд функций, кьюшек, клауд тасок, крон джоб и прочего на GCP и AWS в одночасье (не только легаси старья, но и свежего кода) на чистом js что могу утверждать с уверенностью обратное. У вас недостоверная информация.

0 0 Ответить

**В** **Валодька** → bi_zi
2 года назад edited

Ну, к примеру, чтобы имитировать приватные свойства. В некотором роде пародия на инкапсуляцию.

```
class User {

constructor(name, age){
Object.defineProperties(this, {
name: { value: name, enumerable: true, writable: false, configurable: true },
age: { value: age, enumerable: true, writable: false, configurable: true}
});
}

setName(new_name){
Object.defineProperty(this, "name", {writable: true});
this.name = new_name;
Object.defineProperty(this, "name", {writable: false});
}

setAge(new_age){
```

показать больше

1 0 Ответить

**alphakappa86** → Валодька
2 года назад

приватные должны быть `enumerable: false`

1 0 Ответить

**S** **Sergey Nadegnyy** → Валодька
2 года назад

Пародия на инкапсуляцию - замыкания. Это скорее свойства для чтения или свойства-константы.

1 0 Ответить

**A** **Andrew Sokolovsky** → Sergey Nadegnyy
2 года назад

Ну я так понимаю, что эта тема нужна, что врубиться потом в геттеры и сеттеры, наверное?

0 0 Ответить

**E** **Евгений Шевцов** → bi_zi
2 года назад

// Вот коммент от @Денис Бахматов
Вы имели в виду зачем изменять спец свойства типа "writable"?

Например, чтобы защитить данные от перезаписи объекта извне и иметь гарантию содержимого (у человека дата рождения не может быть изменена, персонал с уровнем доступа строго определен, никто не должен получить этот доступ еще);

Другой пример: для оптимизации кода. Реактивность в некоторых фреймворках реализуется за счет обертки свойств объекта через геттеры и сеттеры. Если объект "заморожен" от изменений, то его свойства не будут обернуты в геттеры и сеттеры и, соответственно, затрат на его "реактивность" не будет.

Другой пример: у нас есть объект со свойствами и методами. Например, человек. Мы хотим получить полную характеристику человека. Используя цикл for..in мы получим также и его методы (дышать, ходить...). Мы можем исключить из итератора методы для данной задачи "enumerable": false

9 0 Ответить

**M** **mili** bi_zi
2 года назад edited

мне было бы тоже интересно узнать как мы будем все это использовать, но как мне кажется нам про это расскажут во 2 части учебника

0 0 Ответить

 **bi_zi** mili
2 года назад

надеюсь так и будет

2 0 Ответить

**E** **Евгений Шевцов** bi_zi
2 года назад

Это расскажут, когда на работу придете))
Скажу что копирование объекта явно пригодится.Я часто копирую объекты в React там такая техника очень выручает

6 0 Ответить

**Ю** **Юрий Нестеренко** Евгений Шевцов
2 года назад

Здравствуйте. Я так понимаю вы фронтенд разработчик. Можно с вами немного пообщаться в телеге? Есть несколько вопросов. @nester_web

0 0 Ответить

**E** **Евгений Шевцов** Юрий Нестеренко
2 года назад

Да вы правы, я вам напишу

0 0 Ответить

**Z** **z1x btw** Евгений Шевцов
2 года назад edited

хотелось бы задать пару вопросов вам, если не заняты, буквально отниму у вас пару минут @z1xbtw

0 0 Ответить

L Lev Levitin
3 года назад

"Если свойство существует, defineProperty обновит его флаги. В противном случае метод СОЗДАЁТ НОВОЕ СВОЙСТВО с указанным значением и флагами."
Подскажите, почему код ниже выдаёт пустой список?

```
let user = {};

Object.defineProperty(user, "name", {
value: "John"
});

alert( JSON.stringify(user) );
```

0 0 Ответить