



→ Язык JavaScript → Свойства объекта, их конфигурация

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:

```
1 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:

```
1 let user = {
2    name: "John"
3 };
4
```



```
let descriptor = Object.getOwnPropertyDescriptor(user, 'name');
5
6
   alert( JSON.stringify(descriptor, null, 2 ) );
7
   /* дескриптор свойства:
8
9
10
      "value": "John",
11
      "writable": true,
12
      "enumerable": true,
13
      "configurable": true
14 }
15 */
```

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

Его синтаксис:

```
1 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:

```
1 let user = {};
2
3 Object.defineProperty(user, "name", {
  value: "John"
4
5
  });
7
   let descriptor = Object.getOwnPropertyDescriptor(user, 'name');
8
9
   alert( JSON.stringify(descriptor, null, 2 ) );
10
   /*
11
12
     "value": "John",
     "writable": false,
13
     "enumerable": false,
14
15
      "configurable": false
16 }
17
    */
```

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 = {
1
2
  name: "John"
3
 };
4
5 Object.defineProperty(user, "name", {
6 writable: false
7
 });
8
9 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»:

```
1 let user = { };
2
3 Object.defineProperty(user, "name", {
4 value: "John",
5
     // для нового свойства необходимо явно указывать все флаги, для которых знач
     enumerable: true,
6
7
     configurable: true
8
  });
9
10 alert(user.name); // John
11 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 = {
1
2
   name: "John",
3
    toString() {
4
      return this.name;
5
    }
```

```
6 };
7
8 // По умолчанию оба свойства выведутся:
9 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:

```
1 let user = {
2
    name: "John",
3
    toString() {
4
      return this.name;
5
     }
6 };
7
8 Object.defineProperty(user, "toString", {
9 enumerable: false
10 });
11
12 // Теперь наше свойство toString пропало из цикла:
13 for (let key in user) alert(key); // name
```

Неперечислимые свойства также не возвращаются Object.keys:

```
1 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');
2
3 alert( JSON.stringify(descriptor, null, 2 ) );
  /*
4
5
6
     "value": 3.141592653589793,
7
     "writable": false,
     "enumerable": false,
8
     "configurable": false
9
10 }
11
   */
```

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

```
1 Math.PI = 3; // Ошибка, потому что writable: false
2
3 // delete Math.PI тоже не сработает
```

Мы также не можем изменить writable:

```
1 // Ошибка, из-за configurable: false
2 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.

Oбратите 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).

```
1 let user = {
2    name: "John"
3 };
4
5 Object.defineProperty(user, "name", {
6    configurable: false
7 });
8
9 user.name = "Pete"; // работает
10 delete user.name; // Ошибка
```

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

```
1 let user = {
2    name: "John"
3 };
4
5 Object.defineProperty(user, "name", {
6    writable: false,
7    configurable: false
8 });
9
10 // теперь невозможно изменить user.name или его флаги
11 // всё это не будет работать:
12 user.name = "Pete";
13 delete user.name;
14 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:

```
1 Object.defineProperties(obj, {
2   prop1: descriptor1,
3   prop2: descriptor2
4   // ...
5 });
```

Например:

```
1 Object.defineProperties(user, {
2   name: { value: "John", writable: false },
3   surname: { value: "Smith", writable: false },
4   // ...
5 });
```

Таким образом, мы можем определить множество свойств одной операцией.

Object.getOwnPropertyDescriptors

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

Bместе с **Object.defineProperties** этот метод можно использовать для клонирования объекта вместе с его флагами:

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

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

```
1 for (let key in user) {
2   clone[key] = user[key]
3 }
```

...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

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.

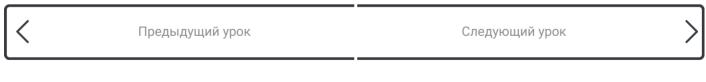
Object.isSealed(obj)

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.



Поделиться









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 , если больше 10 строк — ссылку на песочницу (plnkr, JSBin, codepen...)
- Если что-то непонятно в статье пишите, что именно и с какого места.



Присоединиться к обсуждению... войти с помощью или через disqus ? Имя

О 12 Поделиться
 Лучшие Новые Старые
 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



Хочешь понять флаги и дескрипторы свойств?

Тогда переходи в ТГ-блог «Джун на фронте»!

ABTOP - 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!

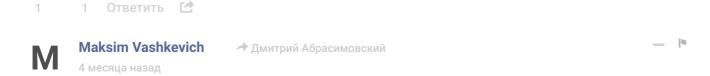
Вбивайте «Джун на фронте» и присоединяйтесь к нам!

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



→ Maksim Vashkevich

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  О 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 саг (которые указаны с флагом enumerable = false) остаётся (они клонируются).

Следующий код подтверждает мои слова:



Можно сверить все свойства клона с оригинальным объектом и все там верно, сейчас перепроверил с помощью:

```
Object.getOwnPropertyDescriptor()
```



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

4 месяца назад

Aleksandr S

4 месяца назад

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

0 Ответить



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)) { // Pa6отает даже если enumerable: false
  Object.defineProperty(result, key, Object.getOwnPropertyDescriptor(obj, key));
  if (typeof obj[key] === 'object') {
    result[key] = flagsClone(obj[key]);
  }
}

return result;
}
```

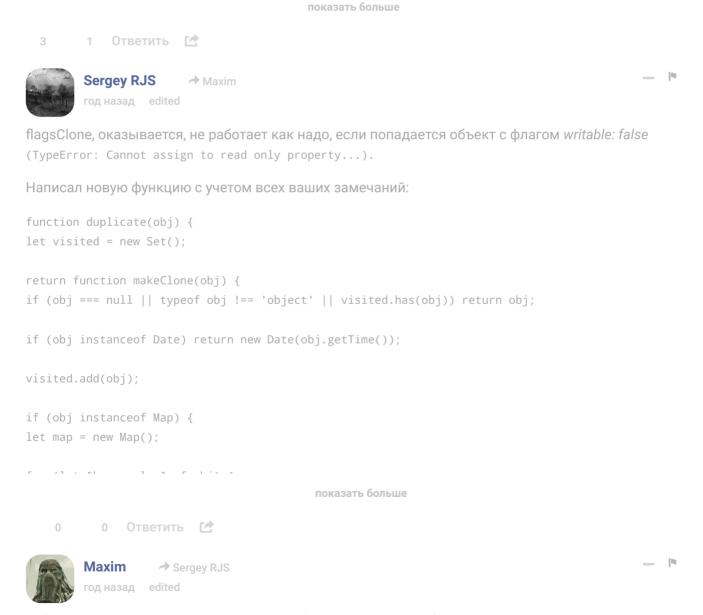


Попытка сделать клонирование с учетом флагов похвальная.

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)



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



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);



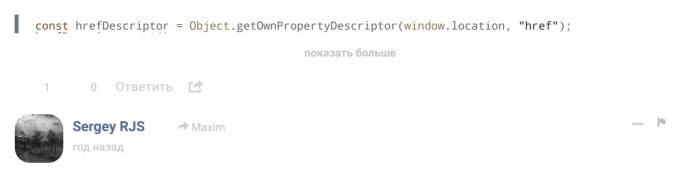
Ну я так и понял что по привычке, бывает :)

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:



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

показать больше

О ОТВЕТИТЬ 

Махіт → 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.



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:)



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

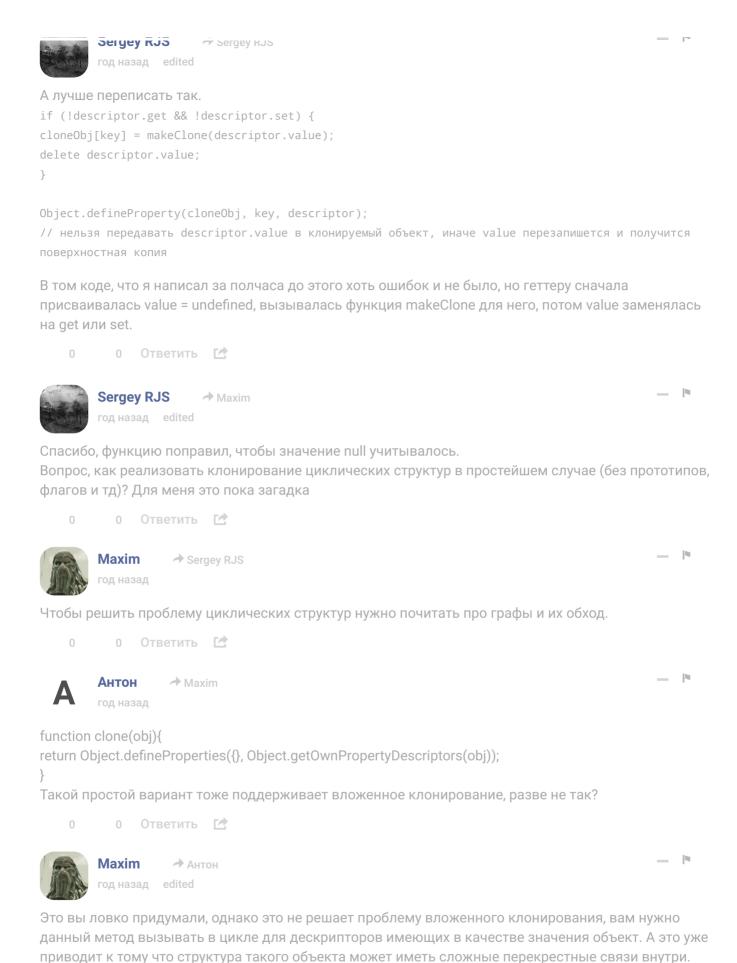
Конечно, ведь нельзя узнать, какие из ловушек были использованы. Только если в консоли свойство [[Handler]] найти. Но это внутреннее свойство.



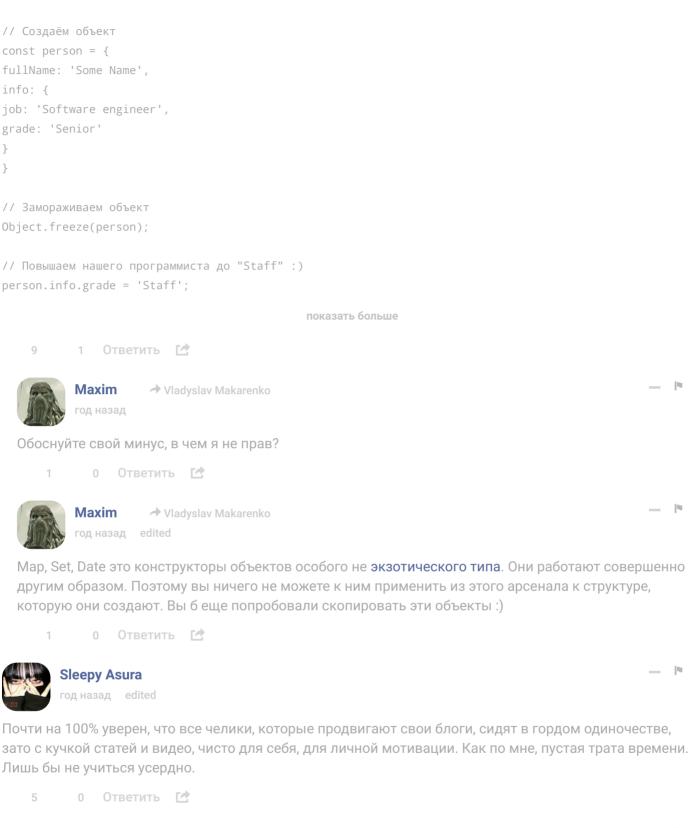
Ну ты опять хочешь клонировать прототипы, но в js это не имеет смысла и выше я уже писал почему. Прототипы это **общие** свойства и методы. И делать суррогатный объект, который будет уникальным прототипом бессмысленно... но это можно сделать, но бессмысленно.

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 🖆

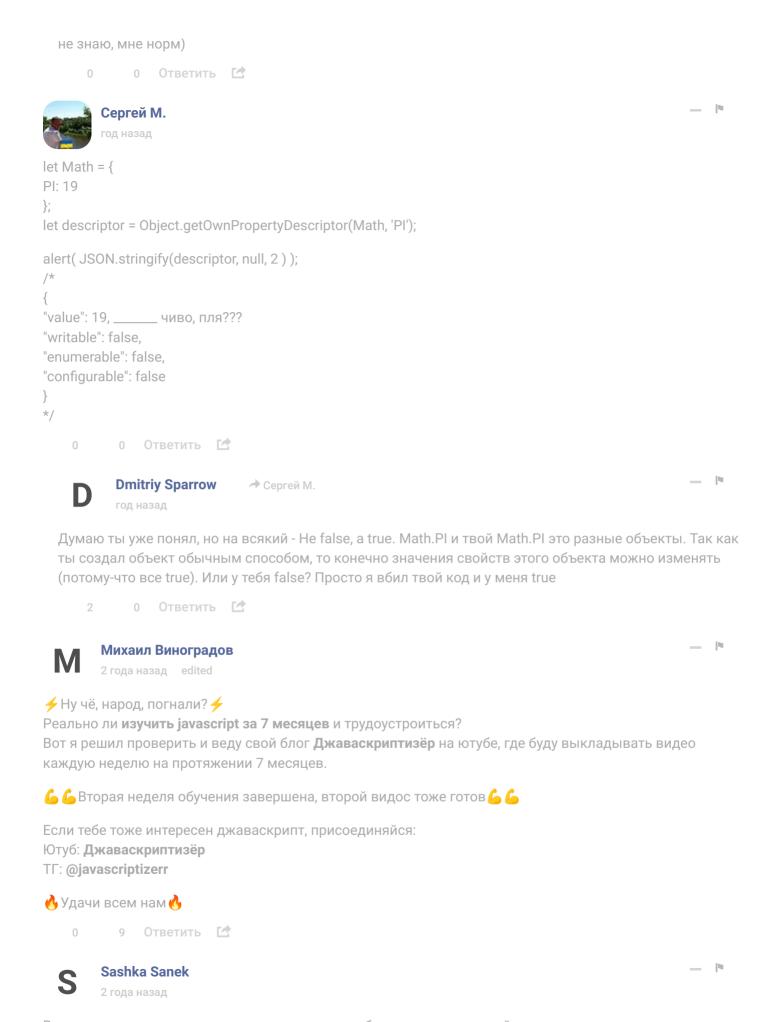


содержимое объектов Мар, Date и Set.



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



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



Это если conigurable было false изначально, иначе один раз можно

0 Ответить 🖆



Ты не так понял. Ошибки в нестрогом режиме не будет, если ты при configurable: false, например удалишь свойство (просто проигнорируется).

A вот если configurable: false, его уже нельзя вернуть на true - тут не зависит от вкл/выкл strict modeпотому что defineProperty не работает с неконфигурируемыми свойствами.



(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 указанным явно флагам присваивается значення false.

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

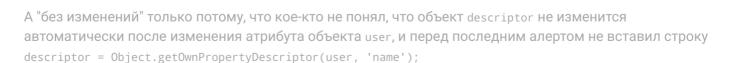
alert( JSON.stringify(descriptor, null, 2 ) );

/*

{
    "value": "John",
    "writable": true,
    """""    f-] --
    показать больше

О ОТВЕТИТЬ 

О ОТВЕТИТЬ
```



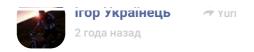
Еще и минусов мне налепил... facepalm.jpg

2 года назад edited

→ Ігор Українець

Yuri

3 0 Ответить 🖆



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"

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

1

0 Ответить



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,
// });
            0 Ответить
            Ігор Українець
                                        → Yuri
            2 года назад
```

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

0 Ответить 🚅

M

Max

2 года назад edited

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 

Сергей Мельников 

→ Мах

2 года назад edited

— I**
```

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)

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 Ответить
```





В случае, когда

configurable: false

И

writable: true

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

Поправьте, если ошибаюсь.



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



Моя реплика призвана исправить неточность авторов:

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

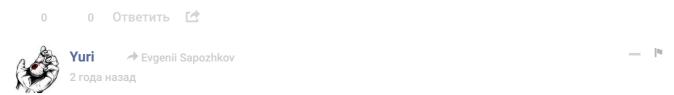
И при изменении атрибута *configurable* (на *false*, других вариантов нет) атрибут *writable* не меняется непосредственно, однако появляются особенности, о которых я и написал. Так что *configurable* и *writable* связаны очень тесно.



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.



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

комментарии репіз чуть ниже, наконец. л устал, честно...

0 0 Ответить



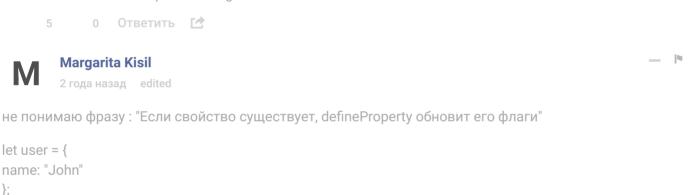
если я правильно понял, теперь объект с данным дескриптором будет навечено writable true, и мы не сможем изменить его свойства, ни удалить его



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



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



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

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

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

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



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



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

0 1 Ответить

Кирилл → Ragima Bagirova
2 года назад edited

Het, метод Object.assign(obj) не создает клон объекта с соответствующими дескриптороми-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
}
}"
```

Радислав Ялилов

0 Ответить 🖾

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



configurable защищает от изменения используя defineProperty(ies)



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



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



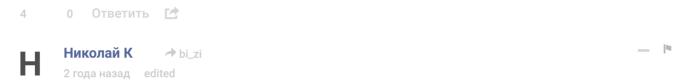
Так это понятно из лекции, выше говорилось, что флаг configurable в true обратно не установить, а эти методы ставят его в false



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.



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?



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.



Со всем уважением, но я бы не делал таких громких умозаключений что на чистом јз 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 јз что могу утверждать с уверенностью обратное. У вас недостоверная информация.

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
          0 Ответить
          alphakappa86
                           → Валодька
          2 года назад
  приватные должны быть enumerable: false
           0 Ответить
          Sergey Nadegnyy
                              → Валодька
  Пародия на инкапсуляцию - замыкания. Это скорее свойства для чтения или свойства-константы.
           0 Ответить
          Andrew Sokolovsky
                              → Sergey Nadegnyy
  Ну я так понимаю, что эта тема нужна, что врубиться потом в геттеры и сеттеры, наверное?
      0 0 Ответить ГА
```

// Вот коммент от @Денис Бахматов

Евгений Шевцов

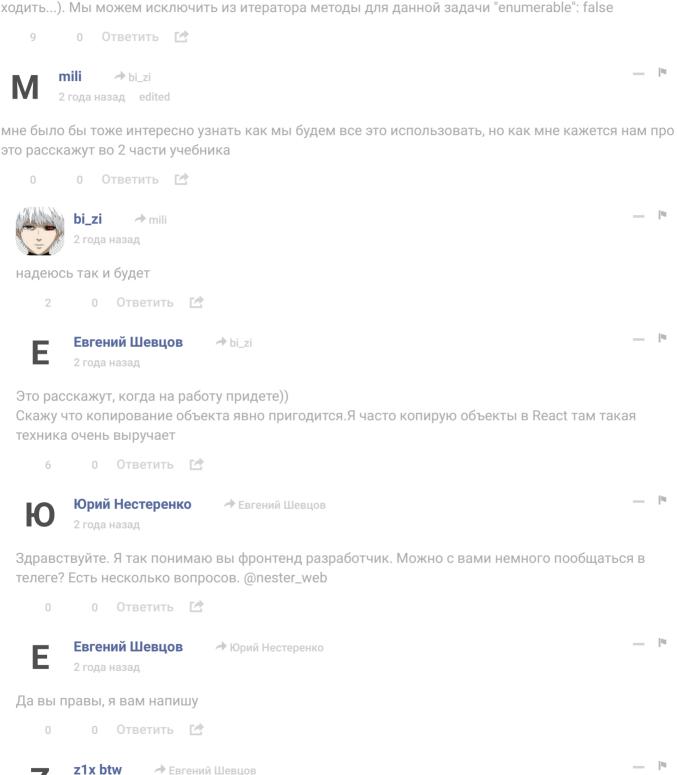
Вы имели в виду зачем изменять спец свойства типа "writable"?

→ bi_zi

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 лолжен получить этот лоступ 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



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

0 Ответить 🗠

2 года назад edited

```
Lev Levitiii
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

Загрузить ещё комментарии

Подписаться

О защите персональных данных

Не продавайте мои данные

