

JavaScript - объект продвинуто

Конструктор, оператор "new"; "this" для работы с объектами;
Object и его методы; краткое знакомство с массивами

Функция-конструктор

Обычный синтаксис {...} позволяет создать только один объект. Но зачастую нам нужно создать множество похожих, однотипных объектов, таких как пользователи, элементы меню и так далее.

Это можно сделать при помощи функции-конструктора и оператора "new".

Функции-конструкторы технически являются обычными функциями. Но есть два соглашения:

Имя функции-конструктора должно начинаться с большой буквы.

Функция-конструктор должна выполняться только с помощью оператора "new".

This в функциях конструкторах

Когда функция вызывается как `new User(...)`, происходит следующее:

- Создается новый пустой объект, и он присваивается `this`.
- Выполняется тело функции. Обычно оно модифицирует `this`, добавляя туда новые свойства.
- Возвращается значение `this`.

```
function User(name) {  
  this.name = name;  
  this.isAdmin = false;  
}  
  
let user = new User("Jack");  
  
alert(user.name); // Jack  
alert(user.isAdmin); // false
```

```
function User(name) {  
  // this = {}; (неявно)  
  
  // добавляет свойства к this  
  this.name = name;  
  this.isAdmin = false;  
  
  // return this; (неявно)  
}
```

Как “задетектить” был ли использован new

Проверка на вызов в режиме конструктора: `new.target`

В случае обычного вызова функции `new.target` будет `undefined`. Если же она была вызвана при помощи `new`, `new.target` будет равен самой функции.

Синтаксис используется крайне редко

```
function User() {  
  alert(new.target);  
}  
  
// без "new":  
User(); // undefined  
  
// с "new":  
new User(); // function User { ... }
```

```
function User(name) {  
  if (!new.target) { // в случае, если вы вызвали меня без оператора new  
    return new User(name); // ...я добавлю new за вас  
  }  
  
  this.name = name;  
}  
  
let john = User("John"); // переадресовывает вызов на new User  
alert(john.name); // John
```

Пропуск скобок

Кстати, мы можем не ставить круглые скобки после new.

Пропуск скобок считается плохой практикой, но просто чтобы вы знали, такой синтаксис разрешён спецификацией.

```
let user = new User; // <-- без скобок  
// то же, что и  
let user = new User();
```

Возврат значения из конструктора, return

Обычно конструкторы не имеют оператора return. Их задача – записать все необходимое в this, и это автоматически становится результатом.

Но если return всё же есть, то применяется простое правило:

- При вызове return с объектом, вместо this вернётся объект.
- При вызове return с примитивным значением, оно проигнорирует.

Другими словами, return с объектом возвращает этот объект, во всех остальных случаях возвращается this.

```
function BigUser() {  
  
    this.name = "John";  
  
    return { name: "Godzilla" }; // <-- возвращает этот объект  
}  
  
alert( new BigUser().name ); // Godzilla, получили этот объект
```

```
function SmallUser() {  
  
    this.name = "John";  
  
    return; // <-- возвращает this  
}  
  
alert( new SmallUser().name ); // John
```

Создание методов в конструкторе

Использование конструкторов для создания объектов дает большую гибкость. Функции-конструкторы могут иметь параметры, определяющие, как создавать объект и что в него записывать.

Конечно, мы можем добавить к `this` не только свойства, но и методы.

Для создания сложных объектов есть и более продвинутый синтаксис – классы (по сути класс синтаксический сахар)

```
function User(name) {  
  this.name = name;  
  
  this.sayHi = function() {  
    alert( "Меня зовут: " + this.name );  
  };  
}
```

```
let john = new User("John");  
  
john.sayHi(); // Меня зовут: John
```

```
/*  
john = {  
  name: "John",  
  sayHi: function() { ... }  
}  
*/
```

«this» не является фиксированным

В JavaScript ключевое слово «this» ведёт себя иначе, чем в большинстве других языков программирования. Его можно использовать в любой функции, даже если это не метод объекта.

Значение `this` вычисляется во время выполнения кода, в зависимости от контекста.

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

```
function sayHi() {  
  alert( this.name );  
}
```

```
// используем одну и ту же функцию в двух объектах  
user.f = sayHi;  
admin.f = sayHi;
```

```
// эти вызовы имеют разное значение this  
// "this" внутри функции - это объект "перед точкой"  
user.f(); // John (this == user)  
admin.f(); // Admin (this == admin)
```

У стрелочных функций нет «this»

Стрелочные функции особенные: у них нет своего «собственного» this. Если мы ссылаемся на this внутри такой функции, то оно берется из внешней «нормальной» функции.

Это особенность стрелочных функций. Она полезна, когда мы на самом деле не хотим иметь отдельное this, а скорее хотим взять его из внешнего контекста.

```
let user = {  
  firstName: "Ilya",  
  sayHi() {  
    let arrow = () => alert(this.firstName);  
    arrow();  
  }  
};  
  
user.sayHi(); // Ilya
```

Перебираемые объекты

for...of представляет собой удобный и элегантный способ для работы с итерируемыми объектами в JavaScript, упрощая перебор и обработку их элементов.

Не поддерживает обычные объекты: Необходимо использовать цикл for...in для итерации по свойствам обычных объектов.

```
let colors = ['red', 'green', 'blue'];

for (let color of colors) {
  console.log(color);
}

// Вывод:
// red
// green
// blue
```

```
let greeting = 'Hello';

for (let char of greeting) {
  console.log(char);
}

// Вывод:
// H
// e
// l
// l
// o
```

Массив (знакомство)

В JavaScript массив (Array) является специальным типом объекта, который используется для хранения упорядоченной коллекции элементов. В отличие от обычного объекта, где ключи могут быть строками или символами, массивы в JavaScript используют целочисленные индексы для доступа к их элементам.

Временно:

Массив это объект у которого ключи это индекс свойства

```
let arr = new Array();  
let arr = [];
```

```
// Создание массива с несколькими элементами
```

```
let fruits = ['яблоко', 'апельсин', 'банан'];
```

```
// Доступ к элементам массива по индексу
```

```
console.log(fruits[0]); // выводит 'яблоко'
```

```
// Изменение элемента массива
```

```
fruits[1] = 'груша';
```

Object методы

`new Object()` - конструктор базового пустого объекта

Также у `Object` есть встроенные методы для работы с объектами

- статические методы
- методы экземпляра

Статические методы вызываются через конструктор `Object` и существуют сами по себе как абстрактные функции описанные в самом конструкторе `Object` (экземпляры не наследуют их)

Методы экземпляра наследуются каждым объектом, так как происходит наследование - функции которыми владеет каждый объект по факту своей принадлежности к конструкции объекта!

Встроенные методы:

Статические методы (на уровне пользователя):

- `Object.assign()`
- `Object.keys()`
- `Object.values()`
- `Object.entries()`
- `Object.fromEntries()`
- `Object.freeze()`
- `Object.isFrozen()`

Методы экземпляра (на уровне пользователя)

- `Object.prototype.hasOwnProperty()`

Object.assign()

Метод `Object.assign()` используется для копирования значений всех собственных перечисляемых свойств из одного или более исходных объектов в целевой объект. После копирования он возвращает целевой объект.

Object.assign(target, ...sources)

target - Целевой объект.

sources - Исходные объекты.

Возвращается получившийся целевой объект.

```
const target = { a: 1, b: 2 };
const source = { b: 4, c: 5 };

const returnedTarget = Object.assign(target, source);

console.log(target);
// Expected output: Object { a: 1, b: 4, c: 5 }

console.log(returnedTarget === target);
// Expected output: true
```

Object.keys()

Метод **Object.keys()** возвращает массив из собственных перечисляемых свойств переданного объекта, в том же порядке, в котором они бы обходились циклом `for...in` (разница между циклом и методом в том, что цикл перечисляет свойства и из цепочки прототипов).

Object.keys(obj)

obj - Объект, чьи собственные перечисляемые свойства будут возвращены.

```
// Массивоподобный объект
var obj = { 0: "a", 1: "b", 2: "c" };
console.log(Object.keys(obj)); // консоль: ['0', '1', '2']

// Массивоподобный объект со случайным порядком ключей
var an_obj = { 100: "a", 2: "b", 7: "c" };
console.log(Object.keys(an_obj)); // консоль: ['2', '7', '100']
```

Object.values()

Метод **Object.values()** возвращает массив значений перечисляемых свойств объекта в том же порядке что и цикл `for...in`. Разница между циклом и методом в том, что цикл перечисляет свойства и из цепочки прототипов.

Object.values(obj)

obj - Объект, чьи значения перечисляемых свойств будут возвращены.

```
const object1 = {  
  a: 'somestring',  
  b: 42,  
  c: false,  
};  
  
console.log(Object.values(object1));  
// Expected output: Array ["somestring", 42, false]
```

Object.entries()

Object.entries() метод возвращает массив собственных перечисляемых свойств указанного объекта в формате [key, value], в том же порядке, что и в цикле for...in (разница в том, что for-in перечисляет свойства из цепочки прототипов). Порядок элементов в массиве который возвращается Object.entries() не зависит от того как объект объявлен. Если существует необходимость в определенном порядке, то массив должен быть отсортирован до вызова метода, например Object.entries(obj).sort((a, b) => a[0] - b[0]);.

Object.entries(obj)

obj - Объект, чьи перечислимые свойства будут возвращены в виде массива [key, value].

```
const object1 = {  
  a: 'somestring',  
  b: 42,  
};  
  
for (const [key, value] of Object.entries(object1)) {  
  console.log(`${key}: ${value}`);  
}  
  
// Expected output:  
// "a: somestring"  
// "b: 42"
```

Object.fromEntries()

Метод `Object.fromEntries()` преобразует список пар ключ-значение в объект.

Метод `Object.fromEntries()` принимает список пар ключ-значение и возвращает новый объект, свойства которого задаются этими записями. Ожидается, что аргумент `iterable` будет объектом, который реализует метод `@@iterator`, который возвращает объект итератора, который создаёт двухэлементный массивоподобный объект, первый элемент которого является значением, которое будет использоваться в качестве ключа свойства, а второй элемент — значением связанного с этим ключом свойства.

Object.fromEntries(iterable);
iterable - Итерируемый объект
(массив),

`Object.fromEntries()` выполняет
процедуру, обратную `Object.entries()`.

```
const entries = new Map([
  ['foo', 'bar'],
  ['baz', 42],
]);

const obj = Object.fromEntries(entries);

console.log(obj);
// Expected output: Object { foo: "bar", baz: 42 }
```

Object.freeze()

Метод **Object.freeze()** замораживает объект: это значит, что он предотвращает добавление новых свойств к объекту, удаление старых свойств из объекта и изменение существующих свойств или значения их атрибутов, настраиваемости и записываемости. В сущности, объект становится эффективно неизменным. Метод возвращает замороженный объект.

Когда объект заморожен с помощью Object.freeze(), его уже невозможно "разморозить" в прямом смысле этого слова.

Object.freeze(obj)

obj - Объект для заморозки.

Object.isFrozen()

Метод **Object.isFrozen()** определяет, был ли объект заморожен.

Объект является замороженным только в том случае, если он не расширяем, все его свойства являются не настраиваемыми и все его свойства данных (то есть такие, которые не являются свойствами доступа с функциями сеттера или геттера) являются не записываемыми.

Object.isFrozen(obj)

obj - Проверяемый объект.

Object.prototype.hasOwnProperty()

Метод **hasOwnProperty()** возвращает логическое значение, указывающее, содержит ли объект указанное свойство.

Каждый объект, произошедший от Object, наследует метод hasOwnProperty. Этот метод может использоваться для определения того, содержит ли объект указанное свойство в качестве собственного свойства объекта; в отличие от оператора in, этот метод не проверяет существование свойств в цепочке прототипов объекта.

obj.hasOwnProperty(prop)

prop - Имя проверяемого свойства.

Ресурсы:

Конструктор, оператор “new” - [ТЫК](#)

this - [ТЫК](#)

MDN документация объекта - [ТЫК](#)