

# Функции-декораторы, оператор «три точки», `call`, `apply`



Антон  
Степанов



# Антон Степанов

Ведущий фронтэнд разработчик  
StepIntegrator



[Антон Степанов](#)



# План занятия

1. [Повторение](#)
2. [Декоратор-логгер](#)
3. [spread, rest, destructuring](#)
4. [Кеширующий декоратор](#)
5. [Передача контекста: call, apply](#)
6. [Декоратор-шпион](#)
7. [Задерживающий декоратор](#)
8. [Тормозящий декоратор](#)

---

## Повторение

Ранее мы уже говорили о функциях высшего порядка **HOF** и коллбэках. Кто помнит, что это такое?



---

## Повторение

**HOF** — функция, **принимаящая** в качестве аргументов другие функции или **возвращающая** другую функцию в качестве результата.

Принимаемые функции, в таком случае, называются **callback** функциями (функциями обратного вызова).



---

# Функции декораторы

**Функция декоратор** — это обёртка вокруг функции, которая изменяет поведение последней. Основная работа по-прежнему выполняется функцией.

**Декораторы** можно рассматривать как *дополнительные возможности* или *аспекты*, которые можно добавить в функцию.

Мы можем добавить один или несколько декораторов. И всё это **без изменения кода оригинальной функции**.





# Декоратор-логгер

# Logger

Примером простого декоратора можно считать **logger**, который выводит **список** переданных аргументов.

Представим, что у нас есть несколько простых функций:

```
const add = (a, b) => a + b;
```

```
const mult = (a, b, c) => a \* b \* c;
```



# Logger

Тогда декоратор **Logger** мог бы выглядеть следующим образом:

```
function decorator(func) {  
  function wrapper(...args) { // (1)  
    console.log("Аргументы: " + args); // (2)  
    return func(...args); // (3)  
  }  
  return wrapper;  
}  
  
const upgradedAdd = decorator(add);  
const upgradedMult = decorator(mult);  
let res1 = upgradedAdd(10, 20); // "Аргументы: 10,20"  
let res2 = upgradedMult(2, 3, 5); // "Аргументы: 10,20"
```

# Объяснение

Смотрите, что происходит:

- декоратор **принимает** на вход функцию;
- при этом **создаёт новую** функцию `wrapper` (*обёртка*) и возвращает её, но не вызывает.

Это можно сделать и в **одну строчку**:

```
return function () { ... }
```

# Объяснение

Это можно сделать и в **одну строчку**:

```
return function () { ... }
```

- эта функция при исполнении выводит список **аргументов** (*пункт 2*);
- **вызывает** исходную функцию и возвращает результат её работы:

```
function wrapper(...args) {  
    // (1)  
    console.log("Аргументы: " + args); // (2)  
    return func(...args); // (3)  
}
```



**spread, rest, destructuring**

## Объяснение

Обратите внимание на `...args` (пункт 1). Это `rest` параметр — с ним вы уже знакомы. Аналогичного результата можно добиться, используя `arguments`, но в последнем случае это не будет массив.

В строчке (3) `...args` это `spread` оператор. `Rest` параметры и `spread` оператор используются в современной разработке исключительно часто:

```
function wrapper(...args) {  
    // (1)  
    console.log("Аргументы: " + args); // (2)  
    return func(...args); // (3)  
}
```

## *spread* оператор: пример

```
const mid = [3, 4];  
  
const arr1 = [1, 2, mid, 5, 6]; //[1, 2, [3, 4], 5, 6]  
  
const arr2 = [1, 2, ...mid, 5, 6]; //[1, 2, 3, 4, 5, 6]
```

Здесь оператор «три точки» превращает массив *mid* в **последовательный набор аргументов**

# Деструктурирующее присваивание

Также очень полезная и широко применяемая возможность — **деструктурирующее присваивание**.

Пример деструктуризации массива:

```
const arr = ["Vasily", 33, "Moscow", "Junior"];

const [name, age] = arr; // name = Vasily, age = 33

const [name, , city] = arr; // name = Vasily, city = Moscow

const [name, ...rest] = arr; // name = Vasily, rest = [33, 'Moscow', 'Junior ']
```

Здесь `...rest` — `rest` параметр

## Пример деструктуризации объекта

```
const obj = { name: "Vasily", age: 33, city: "Moscow", position: "Junior" };

const { name, age } = obj; // name = Vasily, age = 33

const { name, ...rest };
// name = Vasily, rest = {age:33, city:'Moscow', position: 'Junior'}
```

---

*Подробнее о деструктуризации можно прочитать здесь:*

*<https://learn.javascript.ru/destructuring-assignment>*



## Пример деструктуризации для функции

```
const func = ({name,gender}) => {  
  console.log(name,gender)  
}  
  
let obj = {name:'ivan',gender:'male'}  
func(obj)
```

Если наша функция принимает объект, и мы знаем, что он будет содержать определенные свойства, то можем сразу их деструктурировать.

---

*Подробнее о деструктуризации можно прочитать здесь:*

*<https://learn.javascript.ru/destructuring-assignment>*



# Кеширующий декоратор

# Мемоизация

Часто возникает задача сделать так, чтобы функция **кешировала** результат своего выполнения.

Это можно реализовать, используя **внешние переменные**:

```
let cache = {};  
  
function add(a, b) {  
    const hash = a + "," + b;  
  
    // Хеш нужен для однозначного  
    // сопоставления переменных некоторому ключу  
    if (hash in cache) {  
        console.log("Из кеша: " +  
cache[hash]);  
    } else {  
        let result = a + b;  
        cache[hash] = result;  
        console.log("Вычисляем: " + result);  
    }  
}  
  
add(1, 5); // вычисляем 6  
add(1, 5); // из кеша 6  
add(10, 200); // вычисляем 210  
add(1, 5); // из кеша 6
```

---

# Хеш-функция

Так как мы **помещаем** кэшированные результаты в объект, нам необходимо, чтобы было однозначное **соответствие** между набором параметров и ключом.

В данном случае мы используем **простое преобразование**, которое пару  $(a,b)$  переводит в строку  $a,b$ . В более сложных случаях могут потребоваться другие функции хеширования



# Кеширующий декоратор

Как сделать **декорирование универсальным**? Напишем функцию-декоратор, взяв подход logger за основу:

```
function decorator(func) {  
  let cache = {};  
  
  function wrapper(...args) {  
    const hash = args[0] + "," + args[1];  
  
    if (hash in cache) {  
      console.log("Из кеша: " + cache[hash]);  
    } else {  
      let result = func(...args);  
      cache[hash] = result;  
      console.log("Вычисляем: " + result);  
    }  
  }  
  
  return wrapper;  
}
```

---

# Использование

```
const add = (a, b) => a + b;  
  
const memoizedAdd = decorator(add);  
  
memoizedAdd(10, 51); // Вычисляем 61  
  
memoizedAdd(10, 51); // Из кеша 61
```

# А если функция — метод объекта?

Мы уже знакомились с контекстом выполнения ранее. *This* внутри метода:

```
let computer = {  
  text: "Результат",  
  
  add(a, b) {  
    let c = a + b;  
    return this.text + " " + c; // (1)  
  },  
};
```

Допустим, мы захотим использовать **мемоизацию**:

```
computer.add = decorator(computer.add);  
  
computer.add(1, 2);
```

**Результат:** вычисляем *NaN*

# Объяснение

Ошибка возникает в строке (1):

```
...  
    return this.text + " " + c; // (1)  
...
```

Функция **пытается получить доступ** к `this.text` и завершается с ошибкой. Видите, почему?

Причина в том, что в строке `let result = func(...args)` декоратор вызывает оригинальную функцию как `func(...args)`, и она в данном случае получает `this = window`

*Т.е. декоратор передаёт вызов оригинальному методу, но без контекста. Следовательно, возникает ошибка.*





# Передача контекста

## `func.call()`

Существует специальный встроенный метод `func.call(context, ...args)`, который позволяет вызывать функцию, **явно устанавливая контекст** (`this`).

Он запускает функцию `func`, используя первый аргумент как её **контекст** `this`, а последующие — как её **аргументы**:

```
func(...args);  
  
func.call(obj, 51, 10);  
  
func.call(obj, ...args);
```

`func` вызывается с **аргументами**:

`...args = 51, 10` и `this` равным `obj`

# Исправим декоратор

```
function decorator(func) {
  let cache = {};
  return function wrapper(...args) {
    const hash = "a" + args[0] + "b" + args[1];

    if (hash in cache) {
      console.log("Из кеша: " + cache[hash]);
    } else {
      let result = func.call(this, ...args);
      // (Изменения тут)
      cache[hash] = result;
      console.log("Вычисляем: " + result);
    }
  }
}

computer.add = decorator(computer.add);
computer.add(1, 2); // Вычисляем 3
computer.add(1, 2); // Из кеша 3
```

Теперь всё в порядке.

Чтобы всё было понятно,  
давайте посмотрим глубже,  
как передается `this`

## Что происходит

После декорации `computer.add` становится оберткой `function wrapper(...args) { ... }`.

Так что при выполнении `computer.add(1, 2)` обёртка получает 1 и 2 в качестве **аргументов** и `this = computer`, так как это объект перед точкой.

Внутри обёртки, если результат еще **не кеширован**, `func.call(this, x)` передаёт текущий `this = computer` и текущие аргументы в оригинальную функцию

## Ещё пример *call*

```
function getAge() {  
    console.log(this.age);  
}  
  
let user1 = { age: 18 };  
let user2 = { age: 33 };  
  
// используем 'call' для передачи различных объектов в качестве 'this'  
  
getAge.call(user1); // 18  
getAge.call(user2); // 33
```

В приведённом коде мы **вызываем** *getAge* в контексте различных объектов: *getAge.call(user1)* запускает *getAge*, передавая *this = user1*, а следующая строка устанавливает *this = user2*

## Метод *apply*

Ранее мы использовали *func.call(this, ...arguments)*. Вместо этого мы могли бы написать *func.apply(this, arguments)*:

```
func.call(context, ...args); // передаёт массив как список с оператором расширения  
  
func.apply(context, args); // тот же эффект
```

Выполняем *func*, устанавливая *this = context* и принимая в качестве списка аргументов псевдомассив *args*. Есть только одна небольшая разница

---

Синтаксис встроенного метода *func.apply* можете посмотреть здесь:

<https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global%20Objects/Function/apply>

## Метод `apply`

Оператор расширения `...` позволяет **передавать** перебираемый объект `args` в виде списка в `call`, а `apply` принимает только псевдомассив `args`.

А если у нас объект, который и то, и другое, например, **реальный массив**, то технически мы могли бы использовать любой метод.

Псевдомассив - массивоподобный объект со свойством `length` и элементами по индексным ключам.

*Подробнее о перебираемых объектах тут*

*[https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Iteration\\_protocols](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Iteration_protocols)*

---

*Синтаксис встроенного метода `func.apply` можете посмотреть здесь:*

*[https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global\ Objects/Function/apply](https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global%5CObjects/Function/apply)*



# Улучшим хеширующую функцию

А что, если **аргументов много?**

Попробуем сделать хеширующую функцию более **универсальной**

---

*Код на следующей странице*



# Улучшим хеширующую функцию

```
const calculation = (a, b, c, d) => a ** 2 + b ** 2 + c ** 2 + d ** 2;

function decorator(func) {
  let cache = {};
  function wrapper(...args) {
    const hash = args.join(","); // hash = "a,b,c,d"

    if (hash in cache) {
      console.log("Из кеша: " + cache[hash]);
    } else {
      let result = func.call(this, ...args); // (Изменения тут)
      cache[hash] = result;
      console.log("Вычисляем: " + result);
    }
  }
  return wrapper;
}

let cachedCalculation = decorator(calculation);
cachedCalculation(3, 2, 3, 4); // Вычисляем: 38
cachedCalculation(3, 2, 3, 4); // Из кеша: 38;
```



# Декоратор-шпион

# Декоратор-шпион

Мы хотим создать **декоратор**, который будет сохранять список всех вызовов функции (*аргументы*) в свойстве `history` функции обёртки:

```
const add = (a, b) => a + b;

function spyDecorator(func) {

  function wrapper(...args) {
    wrapper.history.push(args);
    return func.call(this, ...args);
  }

  wrapper.history = []; // почему мы можем так сделать?
  return wrapper;
}

const upgradedAdd = spyDecorator(add);
upgradedAdd(100, 200);
upgradedAdd(1, 1);
console.log(upgradedAdd.history); // [100,200] , [1,1]
```



# Задерживающий декоратор

# Задерживающий декоратор

Декоратор создаёт функцию, которая выполняется **с задержкой**:

```
const add = (a, b) => a + b;

function decorator(f, ms) {
  return function (...args) {
    setTimeout(function () {
      f.apply(this, args);
    }, ms);
  };
}

const delayedAdd = decorator(add, 2000);

delayedAdd(51, 10); //61
```

Функция `setTimeout(func, ms)` выполняет переданную внутрь функцию `func` через `ms` миллисекунд. Подробнее об этом будет на лекции «Асинхронность»

# Проблема `setTimeout` и метод объекта

А что будет, если применить декоратор к **методу объекта**?

```
function decorator(f, ms) {  
  return function (...args) {  
    setTimeout(function () {  
      f.apply(this, args); // (1)  
    }, ms);  
  };  
}  
  
let computer = {  
  text: "Результат: ",  
  add(a, b) {  
    console.log(this.text + (a + b)); // (2)  
  },  
};  
  
const delayedAdd = decorator(computer.add, 2000);  
delayedAdd(51, 10);
```

Чему будет равен `this` в строке (1) и соответственно `this.text` в строке (2)?

# Проблема *setTimeout* и метод объекта

А что будет, если применить декоратор к **методу объекта**?

```
function decorator(f, ms) {  
  return function (...args) {  
    setTimeout(function () {  
      f.apply(this, args); // (1)  
    }, ms);  
  };  
}  
  
let computer = {  
  text: "Результат: ",  
  add(a, b) {  
    console.log(this.text + (a + b)); // (2)  
  },  
};  
  
const delayedAdd = decorator(computer.add, 2000);  
delayedAdd(51, 10);
```

**Вывод:** *"undefined 61"*

## Объяснение

Через две секунды `setTimeout` вызовет функцию, которая передана **первым аргументом**:

```
function () { f.apply(this, args); }
```

Как будет вызвана эта функция? Как **метод объекта** или просто как **функция**?

В нашем случае `f = computer.add`. Важно не то, чем является `f`, а как она вызвана.

Тогда чему будет равен `this` внутри вызываемой функции?



## Решение — простой способ сохранить `this`

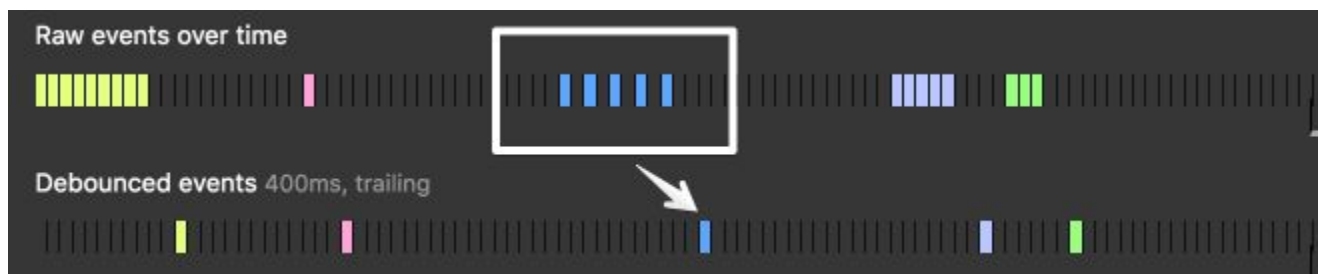
```
function decorator(f, ms) {  
  
    return function (...args) {  
        let savedThis = this; // (2)  
        console.log(savedThis);  
  
        setTimeout(function () {  
            f.apply(savedThis, args); // (3)  
        }, ms);  
    };  
}  
  
computer.add = decorator(computer.add, 2000);  
  
computer.add(51, 10); // (1) // Результат: 61
```

- **вызываем** функцию `add` как метод объекта (*внутри `this = computer`*);
- **сохраняем** `savedThis = computer`;
- **вызываем** `f` (исходная функция) с правильным `this`.

# Декоратор отложенного вызова Debounce

**Декоратор** преобразовывает функцию так, что она будет выполнена только тогда, когда после последней попытки вызова пройдёт определённое время. **Задержка** начинает заново отсчитываться с каждой новой попыткой вызова.

Удобно при обращении к **api**, когда пользователь вбивает текст поиска в *input*. Данные будут отправлены, когда возникнет пауза



На картинке сверху — вызовы функции,  
снизу — работа функции

# Код

```
const showCoords = (x, y) => console.log(`Клик: (${x}, ${y})`);

function decorator(f, ms) {
  let timeout;

  return function (...args) {
    clearTimeout(timeout);

    timeout = setTimeout(() => {
      f.apply(this, args);
      console.timeEnd("time"); // (2)
    }, ms);
  };
}

const delayedFunc = decorator(showCoords, 1000);

console.time("time"); // (1)

setTimeout(() => delayedFunc(10, 5));
setTimeout(() => delayedFunc(20, 10), 980);
setTimeout(() => delayedFunc(30, 30), 980); // "Клик: 30,30" через 2 секунды (примерно)
```

---

## Объяснение

При каждом вызове функция **удаляет старый** таймер и **создает новый**, который сработает, если функция не будет вызвана еще раз.

Функции `console.time` и `console.timeEnd` позволяют **логировать время**

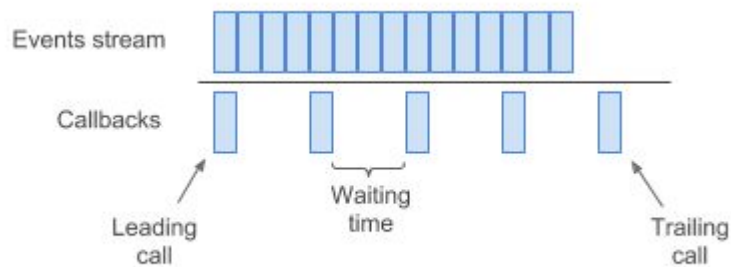




# Тормозящий декоратор

# Тормозящий (*throttle*) декоратор

Пусть у нас есть некоторое действие, например, перемещение персонажа в онлайн-игре, и мы хотим **отправлять информацию** об этом действии на сервер, *но не слишком часто, чтобы не спамить*.



Взято с сайта: [xandeadx.ru](http://xandeadx.ru)

Несмотря на то, что персонаж двигается часто, мы отправляем его координаты не чаще, чем раз в сколько-то *ms*

---

## Тормозящий (*throttle*) декоратор

Нам нужно **сохранять аргументы каждого вызова** до момента отправки, так как мы не знаем, какой будет последним.

Также хотелось бы **сохранить контекст**, если мы будем применять декоратор к методу объекта.

Вот наиболее **комплексное решение**, которое можно использовать как шаблон для остальных

# Тормозящий (*throttle*) декоратор

```
function decorator(func, ms) {  
  
    let isThrottled = false, // (1)  
        savedArgs,  
        savedThis;  
  
    return function (...args) {  
        savedArgs = args; // (2)  
        savedThis = this;  
        if (isThrottled) {  
            return; // (3)  
        }  
  
        func.apply(this, savedArgs); // (4)  
        isThrottled = true;  
  
        setTimeout(() => {  
            isThrottled = false; // (5)  
            func.apply(savedThis, savedArgs);  
        }, ms);  
    };  
}
```



---

## Объяснение

- **создаём переменные** в лексическом окружении, куда будем сохранять; последние аргументы, контекст и состояние: *задержка активна или нет*;
- на каждом вызове **обновляем контекст и аргументы**, *чтобы взять последние*;
- если функция на задержке — просто **выходим**;
- иначе сначала **выполняем** функцию и взводим флаг;
- затем **ставим таймер**, чтобы очистить флаг через **ms** и выполнить функцию в конце с последними сохраненными аргументами и контекстом.

## Пример

```
function func(a) {
  console.log(a);
  console.timeLog();
}

let throttled = decorator(func, 1000);

console.time();
throttled(1); // (задержки нет) 1
throttled(2); // (задержка, 1000 мс ещё не прошло)
throttled(3); // (задержка)

setTimeout(() => throttled(5), 900); // (задержка)

// выведет 5 (последний аргумент) примерно через 1000мс

setTimeout(() => throttled(6), 1100); // (задержки нет) 6

setTimeout(() => throttled(7), 2000); // (задержка)

// выведет 7 (последний аргумент) примерно через
//1000мс с момента запуска throttled(6)
```

Будем **тормозить** очень простую функцию, которая выводит в консоль переданный аргумент и время для отладки



# Итоги



## Чему мы научились?

- **Познакомились** с концепцией декораторов и научились их писать
- **Узнали** о деструктуризации, `spread` операторе и параметрах `rest`
- **Научились** деструктурировать массивы и объекты
- **Научились** использовать параметр `rest` для получения всех аргументов функции
- **Научились** задавать контекст вызова с помощью `apply` и `call`

# Домашнее задание

Давайте посмотрим ваше [домашнее задание](#):

- **вопросы** по домашней работе задаем в группе Slack;
- задачи можно сдавать **по частям**
- зачёт по домашней работе проставляется после того, как приняты **все обязательные задачи**

**Задавайте вопросы и  
пишите отзыв о лекции!**

**Антон Степанов**