20 октября 2023 г.

Декораторы и переадресация вызова, call/apply

JavaScript предоставляет исключительно гибкие возможности по работе с функциями: они могут быть переданы в другие функции, использованы как объекты, и сейчас мы рассмотрим, как *перенаправлять* вызовы между ними и как их декорировать.

Прозрачное кеширование

Представим, что у нас есть функция `slow(x)`, выполняющая ресурсоёмкие вычисления, но возвращающая стабильные результаты. Другими словами, для одного и того же `x` она всегда возвращает один и тот же результат.

Если функция вызывается часто, то, вероятно, мы захотим кешировать (запоминать) возвращаемые ею результаты, чтобы сэкономить время на повторных вычислениях.

Вместо того, чтобы усложнять `slow(x)` дополнительной функциональностью, мы заключим её в функцию-обёртку – «wrapper» (от англ. «wrap» – обёртывать), которая добавит кеширование. Далее мы увидим, что в таком подходе масса преимуществ.

Вот код с объяснениями:

```
function slow(x) {
  // здесь могут быть ресурсоёмкие вычисления
  alert(`Called with ${x}`);
  return x;
}

function cachingDecorator(func) {
  let cache = new Map();

  return function(x) {
    if (cache.has(x)) {    // если кеш содержит такой x,
      return cache.get(x); // читаем из него результат
    }

    let result = func(x); // иначе, вызываем функцию

    cache.set(x, result); // и кешируем (запоминаем) результат
    return result;
  };
}

slow = cachingDecorator(slow);

alert( slow(1) ); // slow(1) кешируем
alert( "Again: " + slow(1) ); // возвращаем из кеша

```

```
alert( slow(2) ); // slow(2) кешируем
alert( "Again: " + slow(2) ); // возвращаем из кеша
```

В коде выше `cachingDecorator` – это *декоратор*, специальная функция, которая принимает другую функцию и изменяет её поведение.

Идея состоит в том, что мы можем вызвать `cachingDecorator` с любой функцией, в результате чего мы получим кеширующую обёртку. Это здорово, т.к. у нас может быть множество функций, использующих такую функциональность, и всё, что нам нужно сделать – это применить к ним `cachingDecorator` .

Отделяя кеширующий код от основного кода, мы также сохраняем чистоту и простоту последнего.

Результат вызова `cachingDecorator(func)` является «обёрткой», т.е. `function(x)` «оборачивает» вызов `func(x)` в кеширующую логику:

```
function                ( ) {
  let       = new    ();

  return function( ) {          обёртка
    if (     .   ( )) {
      return      .get( );
    }

    let result = func(x);       вокруг функции

        .set( ,       );
    return       ;
  };
}
```

С точки зрения внешнего кода, обёрнутая функция `slow` по-прежнему делает то же самое. Обёртка всего лишь добавляет к её поведению аспект кеширования.

Подводя итог, можно выделить несколько преимуществ использования отдельной `cachingDecorator` вместо изменения кода самой `slow` :

- Функцию `cachingDecorator` можно использовать повторно. Мы можем применить её к другой функции.
- Логика кеширования является отдельной, она не увеличивает сложность самой `slow` (если таковая была).
- При необходимости мы можем объединить несколько декораторов (речь об этом пойдёт позже).

Применение «func.call» для передачи контекста.

Упомянутый выше кеширующий декоратор не подходит для работы с методами объектов.

Например, в приведённом ниже коде `worker.slow()` перестаёт работать после применения декоратора:

```
// сделаем worker.slow кеширующим
let worker = {
  someMethod() {
    return 1;
  },

  slow(x) {
    // здесь может быть страшно тяжёлая задача для процессора
    alert("Called with " + x);
    return x * this.someMethod(); // (*)
  }
```

```
};

// тот же код, что и выше
function cachingDecorator(func) {
  let cache = new Map();
  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);
    }
    let result = func(x); // (**)
    cache.set(x, result);
    return result;
  };
}

alert( worker.slow(1) ); // оригинальный метод работает

worker.slow = cachingDecorator(worker.slow); // теперь сделаем его кеширующим

alert( worker.slow(2) ); // Ой! Ошибка: не удаётся прочитать свойство 'someMet
```

Ошибка возникает в строке `(*)`. Функция пытается получить доступ к `this.someMethod` и завершается с ошибкой. Видите почему?

Причина в том, что в строке `(**)` декоратор вызывает оригинальную функцию как `func(x)`, и она в данном случае получает `this = undefined`.

Мы бы наблюдали похожую ситуацию, если бы попытались запустить:

```
let func = worker.slow;
func(2);
```

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

Давайте это исправим.

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

Синтаксис:

```
func.call(context, arg1, arg2, ...)
```

Он запускает функцию `func`, используя первый аргумент как её контекст `this`, а последующие – как её аргументы.

Проще говоря, эти два вызова делают почти то же самое:

```
func(1, 2, 3);
func.call(obj, 1, 2, 3)
```

Они оба вызывают `func` с аргументами `1`, `2` и `3`. Единственное отличие состоит в том, что `func.call` ещё и устанавливает `this` равным `obj`.

Например, в приведённом ниже коде мы вызываем `sayHi` в контексте различных объектов: `sayHi.call(user)` запускает `sayHi`, передавая `this=user`, а следующая строка устанавливает `this=admin`:

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

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

// используем 'call' для передачи различных объектов в качестве 'this'
sayHi.call( user ); // John
sayHi.call( admin ); // Admin
```

Здесь мы используем `call` для вызова `say` с заданным контекстом и фразой:

```
function say(phrase) {
  alert(this.name + ': ' + phrase);
}

let user = { name: "John" };

// 'user' становится 'this', и "Hello" становится первым аргументом
say.call( user, "Hello" ); // John: Hello
```

В нашем случае мы можем использовать `call` в обёртке для передачи контекста в исходную функцию:

```
let worker = {
  someMethod() {
    return 1;
  },

  slow(x) {
    alert("Called with " + x);
    return x * this.someMethod(); // (*)
  }
};

function cachingDecorator(func) {
  let cache = new Map();
  return function(x) {
    if (cache.has(x)) {
      return cache.get(x);
    }
    let result = func.call(this, x); // теперь 'this' передаётся правильно
    cache.set(x, result);
    return result;
  };
```

```
}

worker.slow = cachingDecorator(worker.slow); // теперь сделаем её кеширующей

alert( worker.slow(2) ); // работает
alert( worker.slow(2) ); // работает, не вызывая первоначальную функцию (кешир
```

Теперь всё в порядке.

Чтобы всё было понятно, давайте посмотрим глубже, как передаётся `this`:

1. После *декорации* `worker.slow` становится обёрткой `function (x) { ... }`.
2. Так что при выполнении `worker.slow(2)` обёртка получает `2` в качестве аргумента и `this=worker` (так как это объект перед точкой).
3. Внутри обёртки, если результат ещё не кеширован, `func.call(this, x)` передаёт текущий `this` (`=worker`) и текущий аргумент (`=2`) в оригинальную функцию.

Переходим к нескольким аргументам с «func.apply»

Теперь давайте сделаем `cachingDecorator` ещё более универсальным. До сих пор он работал только с функциями с одним аргументом.

Как же кешировать метод с несколькими аргументами `worker.slow`?

```
let worker = {
  slow(min, max) {
    return min + max; // здесь может быть тяжёлая задача
  }
};

// будет кешировать вызовы с одинаковыми аргументами
worker.slow = cachingDecorator(worker.slow);
```

Здесь у нас есть две задачи для решения.

Во-первых, как использовать оба аргумента `min` и `max` для ключа в коллекции `cache`? Ранее для одного аргумента `x` мы могли просто сохранить результат `cache.set(x, result)` и вызвать `cache.get(x)`, чтобы получить его позже. Но теперь нам нужно запомнить результат для *комбинации аргументов* `(min,max)`. Встроенный `Map` принимает только одно значение как ключ.

Есть много возможных решений:

1. Реализовать новую (или использовать стороннюю) структуру данных для коллекции, которая более универсальна, чем встроенный `Map`, и поддерживает множественные ключи.
2. Использовать вложенные коллекции: `cache.set(min)` будет `Map`, которая хранит пару `(max, result)`. Тогда получить `result` мы сможем, вызвав `cache.get(min).get(max)`.
3. Соединить два значения в одно. В нашем конкретном случае мы можем просто использовать строку `"min,max"` как ключ к `Map`. Для гибкости, мы можем позволить передавать *хеширующую функцию* в декоратор, которая знает, как сделать одно значение из многих.

Для многих практических применений третий вариант достаточно хорош, поэтому мы будем придерживаться его.

Также нам понадобится заменить `func.call(this, x)` на `func.call(this, ...arguments)`, чтобы передавать все аргументы обёрнутой функции, а не только первый.

Вот более мощный `cachingDecorator`:

```
let worker = {
  slow(min, max) {
    alert(`Called with ${min},${max}`);
    return min + max;
  }
};

function cachingDecorator(func, hash) {
  let cache = new Map();
  return function() {
    let key = hash(arguments); // (*)
    if (cache.has(key)) {
      return cache.get(key);
    }

    let result = func.call(this, ...arguments); // (**)

    cache.set(key, result);
    return result;
  };
}

function hash(args) {
  return args[0] + ',' + args[1];
}

worker.slow = cachingDecorator(worker.slow, hash);

alert( worker.slow(3, 5) ); // работает
alert( "Again " + worker.slow(3, 5) ); // аналогично (из кеша)
```

Теперь он работает с любым количеством аргументов.

Есть два изменения:

- В строке `(*)` вызываем `hash` для создания одного ключа из `arguments`. Здесь мы используем простую функцию «объединения», которая превращает аргументы `(3, 5)` в ключ `"3,5"`. В более сложных случаях могут потребоваться другие функции хеширования.
- Затем в строке `(**)` используем `func.call(this, ...arguments)` для передачи как контекста, так и всех аргументов, полученных обёрткой (независимо от их количества), в исходную функцию.

Вместо `func.call(this, ...arguments)` мы могли бы написать `func.apply(this, arguments)`.

Синтаксис встроенного метода func.apply:

```
func.apply(context, args)
```

Он выполняет `func`, устанавливая `this=context` и принимая в качестве списка аргументов псевдомассив `args`.

Единственная разница в синтаксисе между `call` и `apply` состоит в том, что `call` ожидает список аргументов, в то время как `apply` принимает псевдомассив.

Эти два вызова почти эквивалентны:

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

Есть только одна небольшая разница:

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

Так что эти вызовы дополняют друг друга. Для перебираемых объектов сработает `call`, а где мы ожидаем псевдомассив – `apply`.

А если у нас объект, который и то, и другое, например, реальный массив, то технически мы могли бы использовать любой метод, но `apply`, вероятно, будет быстрее, потому что большинство движков JavaScript внутренне оптимизируют его лучше.

Передача всех аргументов вместе с контекстом другой функции называется «перенаправлением вызова» (call forwarding).

Простейший вид такого перенаправления:

```
let wrapper = function() {
  return func.apply(this, arguments);
};
```

При вызове `wrapper` из внешнего кода его не отличить от вызова исходной функции.

Заимствование метода

Теперь давайте сделаем ещё одно небольшое улучшение функции хеширования:

```
function hash(args) {
  return args[0] + ',' + args[1];
}
```

На данный момент она работает только для двух аргументов. Было бы лучше, если бы она могла склеить любое количество `args`.

Естественным решением было бы использовать метод arr.join:

```
function hash(args) {
  return args.join();
}
```

...К сожалению, это не сработает, потому что мы вызываем `hash(arguments)`, а объект `arguments` является перебираемым и псевдомассивом, но не реальным массивом.

Таким образом, вызов `join` для него потерпит неудачу, что мы и можем видеть ниже:

```
function hash() {
  alert( arguments.join() ); // Ошибка: arguments.join не является функцией
}

hash(1, 2);
```

Тем не менее, есть простой способ использовать соединение массива:

```
function hash() {
  alert( [].join.call(arguments) ); // 1,2
}

hash(1, 2);
```

Этот трюк называется *заимствование метода*.

Мы берём (заимствуем) метод `join` из обычного массива `[].join`. И используем `[].join.call`, чтобы выполнить его в контексте `arguments`.

Почему это работает?

Это связано с тем, что внутренний алгоритм встроенного метода `arr.join(glue)` очень прост. Взято из спецификации практически «как есть»:

1. Пускай первым аргументом будет `glue` или, в случае отсутствия аргументов, им будет запятая `","`
2. Пускай `result` будет пустой строкой `""`.
3. Добавить `this[0]` к `result`.
4. Добавить `glue` и `this[1]`.
5. Добавить `glue` и `this[2]`.
6. ...выполнять до тех пор, пока `this.length` элементов не будет склеено.
7. Вернуть `result`.

Таким образом, технически он принимает `this` и объединяет `this[0]`, `this[1]` ... и т.д. вместе. Он намеренно написан так, что допускает любой псевдомассив `this` (не случайно, многие методы следуют этой практике). Вот почему он также работает с `this=arguments`.

Итого

Декоратор – это обёртка вокруг функции, которая изменяет поведение последней. Основная работа по-прежнему выполняется функцией.

Обычно безопасно заменить функцию или метод декорированным, за исключением одной мелочи. Если исходная функция предоставляет свойства, такие как `func.calledCount` или типа того, то декорированная функция их не предоставит. Потому что это обёртка. Так что нужно быть осторожным в их использовании. Некоторые декораторы предоставляют свои собственные свойства.

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!

Для реализации `cachingDecorator` мы изучили методы:

- func.call(context, arg1, arg2...) – вызывает `func` с данным контекстом и аргументами.

- func.apply(context, args) – вызывает `func`, передавая `context` как `this` и псевдомассив `args` как список аргументов.

В основном *переадресация вызова* выполняется с помощью `apply`:

```
let wrapper = function(original, arguments) {
  return original.apply(this, arguments);
};
```

Мы также рассмотрели пример *заимствования метода*, когда мы вызываем метод у объекта в контексте другого объекта. Весьма распространено заимствовать методы массива и применять их к `arguments`. В качестве альтернативы можно использовать объект с остаточными параметрами `...args`, который является реальным массивом.

На практике декораторы используются для самых разных задач. Проверьте, насколько хорошо вы их освоили, решая задачи этой главы.

Задачи

Декоратор-шпион

важность: 5

Создайте декоратор `spy(func)`, который должен возвращать обёртку, которая сохраняет все вызовы функции в своём свойстве `calls`.

Каждый вызов должен сохраняться как массив аргументов.

Например:

```
function work(a, b) {
  alert( a + b ); // произвольная функция или метод
}

work = spy(work);

work(1, 2); // 3
work(4, 5); // 9

for (let args of work.calls) {
  alert( 'call:' + args.join() ); // "call:1,2", "call:4,5"
}
```

P.S.: Этот декоратор иногда полезен для юнит-тестирования. Его расширенная форма – `sinon.spy` – содержится в библиотеке Sinon.JS.

Открыть песочницу с тестами для задачи.

решение

Обертка, возвращаемая `spy(f)`, должна хранить все аргументы, и затем использовать `f.apply` для переадресации вызова.

```
function spy(func) {

  function wrapper(...args) {
    // мы используем ...args вместо arguments для хранения "реального"
    wrapper.calls.push(args);
    return func.apply(this, args);
  }

  wrapper.calls = [];

  return wrapper;
}
```

Открыть решение с тестами в песочнице.

Задерживающий декоратор

важность: 5

Создайте декоратор `delay(f, ms)`, который задерживает каждый вызов `f` на `ms` миллисекунд. Например:

```
function f(x) {
  alert(x);
}

// создаём обёртки
let f1000 = delay(f, 1000);
let f1500 = delay(f, 1500);

f1000("test"); // показывает "test" после 1000 мс
f1500("test"); // показывает "test" после 1500 мс
```

Другими словами, `delay(f, ms)` возвращает вариант `f` с «задержкой на `ms` мс».

В приведённом выше коде `f` – функция с одним аргументом, но ваше решение должно передавать все аргументы и контекст `this`.

Открыть песочницу с тестами для задачи.

решение

Решение:

```
function delay(f, ms) {

```

```
  return function() {
    setTimeout(() => f.apply(this, arguments), ms);
  };

}

let f1000 = delay(alert, 1000);

f1000("test"); // показывает "test" после 1000 мс
```

Обратите внимание, как здесь используется функция-стрелка. Как мы знаем, функция-стрелка не имеет собственных `this` и `arguments`, поэтому `f.apply(this, arguments)` берет `this` и `arguments` из обёртки.

Если мы передадим обычную функцию, `setTimeout` вызовет её без аргументов и с `this=window` (при условии, что код выполняется в браузере).

Мы всё ещё можем передать правильный `this`, используя промежуточную переменную, но это немного громоздко:

```
function delay(f, ms) {

  return function(...args) {
    let savedThis = this; // сохраняем this в промежуточную переменную
    setTimeout(function() {
      f.apply(savedThis, args); // используем её
    }, ms);
  };

}
```

Открыть решение с тестами в песочнице.

Декоратор debounce

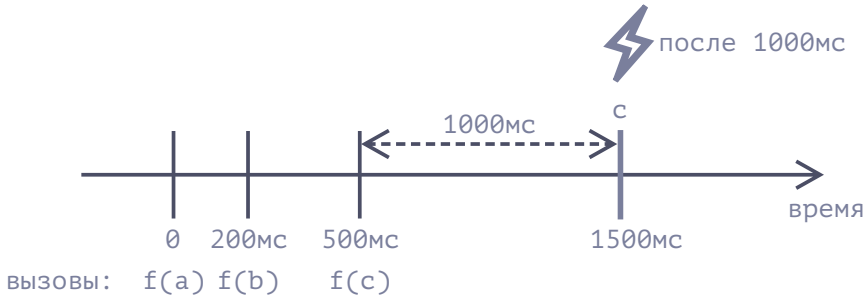
важность: 5

Результат декоратора `debounce(f, ms)` – это обёртка, которая откладывает вызовы `f`, пока не пройдёт `ms` миллисекунд бездействия (без вызовов, «cooldown period»), а затем вызывает `f` один раз с последними аргументами.

Другими словами, `debounce` – это так называемый секретарь, который принимает «телефонные звонки», и ждёт, пока не пройдет `ms` миллисекунд тишины. И только после этого передает «начальнику» информацию о последнем звонке (вызывает непосредственно `f`).

Например, у нас была функция `f` и мы заменили её на `f = debounce(f, 1000)`.

Затем, если обёрнутая функция вызывается в 0, 200 и 500 мс, а потом вызовов нет, то фактическая `f` будет вызвана только один раз, в 1500 мс. То есть: по истечению 1000 мс от последнего вызова.



...И она получит аргументы самого последнего вызова, остальные вызовы игнорируются.

Ниже код этого примера (используется декоратор debounce из библиотеки Lodash):

```
let f = _.debounce(alert, 1000);

f("a");
setTimeout( () => f("b"), 200);
setTimeout( () => f("c"), 500);

// Обёрнутая в debounce функция ждёт 1000 мс после последнего вызова, а затем
```

Теперь практический пример. Предположим, пользователь набирает какой-то текст, и мы хотим отправить запрос на сервер, когда ввод этого текста будет завершён.

Нет смысла отправлять запрос для каждого набранного символа. Вместо этого мы хотели бы подождать, а затем обработать весь результат.

В браузере мы можем настроить обработчик событий – функцию, которая вызывается при каждом изменении поля для ввода. Обычно обработчик событий вызывается очень часто, для каждого набранного символа. Но если мы воспользуемся `debounce` на 1000мс, то он будет вызван только один раз, через 1000мс после последнего ввода символа.

В этом живом примере обработчик помещает результат в поле ниже, попробуйте:

Функция `handler` вызывается на этом поле для ввода:

введите текст

А на этом поле функция `handler` вызывается с применением debounce – `debounce(handler, 1000)`:

введите текст

Функция handler помещает результат сюда

Видите? На втором поле вызывается функция, обёрнутая в `debounce`, поэтому его содержимое обрабатывается через 1000мс с момента последнего ввода.

Таким образом, `debounce` – это отличный способ обработать последовательность событий: будь то последовательность нажатий клавиш, движений мыши или ещё что-либо.

Он ждёт заданное время после последнего вызова, а затем запускает свою функцию, которая может обработать результат.

Задача — реализовать декоратор `debounce`.

Подсказка: это всего лишь несколько строк, если вдуматься :)

Открыть песочницу с тестами для задачи.

решение

```
function debounce(func, ms) {
  let timeout;
  return function() {
    clearTimeout(timeout);
    timeout = setTimeout(() => func.apply(this, arguments), ms);
  };
}
```

Вызов `debounce` возвращает обёртку. При вызове он планирует вызов исходной функции через указанное количество `ms` и отменяет предыдущий такой тайм-аут.

Открыть решение с тестами в песочнице.

Тормозящий (throttling) декоратор

важность: 5

Создайте «тормозящий» декоратор `throttle(f, ms)`, который возвращает обёртку.

При многократном вызове он передает вызов `f` не чаще одного раза в `ms` миллисекунд.

По сравнению с декоратором `debounce` поведение совершенно другое:

- `debounce` запускает функцию один раз после периода «бездействия». Подходит для обработки конечного результата.
- `throttle` запускает функцию не чаще, чем указанное время `ms`. Подходит для регулярных обновлений, которые не должны быть слишком частыми.

Другими словами, `throttle` похож на секретаря, который принимает телефонные звонки, но при этом беспокоит начальника (вызывает непосредственно `f`) не чаще, чем один раз в `ms` миллисекунд.

Давайте рассмотрим реальное применение, чтобы лучше понять это требование и выяснить, откуда оно взято.

Например, мы хотим отслеживать движения мыши.

В браузере мы можем реализовать функцию, которая будет запускаться при каждом перемещении указателя и получать его местоположение. Во время активного использования мыши эта функция запускается очень часто, что-то около 100 раз в секунду (каждые 10 мс). **Мы бы хотели обновлять некоторую информацию на странице при передвижении указателя.**

...Но функция обновления `update()` слишком ресурсоёмкая, чтобы делать это при каждом микродвижении. Да и нет смысла делать обновление чаще, чем один раз в 1000 мс.

Поэтому мы обернём вызов в декоратор: будем использовать `throttle(update, 1000)` как функцию, которая будет запускаться при каждом перемещении указателя вместо оригинальной `update()`. Декоратор будет вызываться часто, но передавать вызов в `update()` максимум раз в 1000 мс.

Визуально это будет выглядеть вот так:

1. Для первого движения указателя декорированный вариант сразу передаёт вызов в `update`. Это важно, т.к. пользователь сразу видит нашу реакцию на его перемещение.
2. Затем, когда указатель продолжает движение, в течение 1000 мс ничего не происходит. Декорированный вариант игнорирует вызовы.
3. По истечению 1000 мс происходит ещё один вызов `update` с последними координатами.
4. Затем, наконец, указатель где-то останавливается. Декорированный вариант ждёт, пока не истечёт 1000 мс, и затем вызывает `update` с последними координатами. В итоге окончательные координаты указателя тоже обработаны.

Пример кода:

```
function f(a) {
  console.log(a)
}

// f1000 передаёт вызовы f максимум раз в 1000 мс
let f1000 = throttle(f, 1000);

f1000(1); // показывает 1
f1000(2); // (ограничение, 1000 мс ещё нет)
f1000(3); // (ограничение, 1000 мс ещё нет)

// когда 1000 мс истекли ...
// ...выводим 3, промежуточное значение 2 было проигнорировано
```

P.S. Аргументы и контекст `this`, переданные в `f1000`, должны быть переданы в оригинальную `f`.

Открыть песочницу с тестами для задачи.

решение

```
function throttle(func, ms) {

  let isThrottled = false,
    savedArgs,
    savedThis;

  function wrapper() {

    if (isThrottled) { // (2)
      savedArgs = arguments;
      savedThis = this;
      return;
    }

    func.apply(this, arguments); // (1)

    isThrottled = true;

    setTimeout(function() {
      isThrottled = false; // (3)
```

```
21        if (savedArgs) {
22          wrapper.apply(savedThis, savedArgs);
23          savedArgs = savedThis = null;
24        }
25      }, ms);
26    }
27
28    return wrapper;
29  }
```

Вызов `throttle(func, ms)` возвращает `wrapper`.

1. Во время первого вызова обёртка просто вызывает `func` и устанавливает состояние задержки (`isThrottled = true`).
2. В этом состоянии все вызовы запоминаются в `saveArgs / saveThis`. Обратите внимание, что контекст и аргументы одинаково важны и должны быть запомнены. Они нам нужны для того, чтобы воспроизвести вызов позднее.
3. Затем по прошествии `ms` миллисекунд срабатывает `setTimeout`. Состояние задержки сбрасывается (`isThrottled = false`). И если мы проигнорировали вызовы, то «обёртка» выполняется с последними запомненными аргументами и контекстом.

На третьем шаге выполняется не `func`, а `wrapper`, потому что нам нужно не только выполнить `func`, но и ещё раз установить состояние задержки и таймаут для его сброса.

Открыть решение с тестами в песочнице.

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.

Юрий

месяц назад

⚡ **Хочешь понять методы call/apply?**
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 0 Ответить

Baikova Maria

3 месяца назад

офигеть, в декораторе throttle как круто передаются savedThis, savedArgs туда, в очередь макрозадач. Это как вообще?? Я думала, так нельзя...

0 0 Ответить

Rail Batyrshin Baikova Maria

2 месяца назад

Тоже немного подвис на этой задаче))

0 0 Ответить

Боб

3 месяца назад edited

Опыт в программировании есть, но всё равно некоторые моменты без опыта использования не усвоишь)
Ещё вопрос по решению задачи 1: если функция видна в environment сразу при запуске программы, почему мы не можем присвоить ей свойство раньше, чем она инициализирована?

0 0 Ответить

hhhscvx
3 месяца назад

В чем разница в третьей задаче с тем, чтобы написать так:

```
function debounce(f, ms) {
return function () {
setTimeout(() => f.apply(this, arguments), ms);
};
}
```

Я в принципе не понял в каком примере это применимо, рад если кто-то объяснит

0 0 Ответить

ToFreeTo → hhhscvx
3 месяца назад

В твоём примере функции вызываются после ms не перебивая старые функции, а в ответе к третьей задаче новые вызовы функции перебивают старые (стирая timer), таким образом она обновляет свой таймер.

Попробуй поиграться с input'ом и посмотри чем оно отличается, так будет намного наглядней

0 0 Ответить

ToFreeTo
3 месяца назад edited

```
function throttle(func, ms) {

    let isStarted = true

    return function () {

        if (isStarted) {

            isStarted = false

            func.apply(this, arguments)

            setTimeout(() => {

                isStarted = true

            }, ms)
        }
    }
}
```

решение для четвёртой задачи. Судя по ответу к 4 задаче мне кажется что я что-то не так сделал, поэтому знающих прошу поправить что в моём решении не так, или мб я не так понял задачу

0 0 Ответить

IT
4 месяца назад

Зачем в 4 задаче в блоке if пишем return? Выполнение же прекратится на втором этапе? Этот момент не особо понятен

0 0 Ответить

Никита Степанов
4 месяца назад

В первой задаче для чего передавать функции контекст *this*? Ведь функция *work* и без *apply* "дотягивается" до массива *calls*

```
return func.apply(this, args); //вот эту строку
return func(...args); //можно заменить на эту
```

И результат выполнения кода останется прежним

1 0 Ответить

Odelschwank Никита Степанов
4 месяца назад

как я понял, apply и call требуют контекста, мб автор просто захотел продеманстрировать такой вариант

0 0 Ответить

Грамматический
5 месяцев назад edited

В задаче «Тормозящий (throttling) декоратор» у меня была идея запоминать время (timestamp) последнего вызова функции, и следующий вызов делать не ранее, чем через заданное количество миллисекунд после предыдущего вызова. Протестировал своё решение в браузере — декоратор работает так, как требуется. Однако это решение не проходит тесты, предложенные авторами учебника. Очень странно. Решение вместе с кодом для отладки прилагаю.

```
function throttle(func, throttleTime) {
    let callThis = null;
    let callArgs = [];
    let lastCallTime = 0;
    let timeoutId = null;

    return function(...args) {
        callThis = this;
        callArgs = args;

        clearTimeout(timeoutId);
        timeoutId = setTimeout(() => {
            lastCallTime = Date.now();
            func.call(callThis, ...callArgs);
        }, Math.max(0, lastCallTime + throttleTime - Date.now()));
    };
}
```

показать больше

0 0 Ответить

Vladimir
5 месяцев назад

2я задача, не могу понять, почему вызов срабатывает сразу
function delay(func, ms) {
function f(...args) {
setTimeout(func.apply(this, args), ms)
};
return f;
}

0 0 Ответить

Грамматический Vladimir

5 месяцев назад edited

Вы передаёте в setTimeout **результат** вызова func.apply. Действительно, в Вашем коде func вызывается сразу, а результат, возвращаемый функцией func, попадает в setTimeout. Если бы func в результате своего вызова возвращала функцию, то эта функция и была бы запланирована с помощью setTimeout.

Исправить решение можно так:

```
function delay(func, ms) {
    function f(...args) {
        setTimeout(() => func.apply(this, args), ms);
    };
    return f;
}
```

То есть мы передаём в setTimeout функцию (в данном случае анонимную =>), которая в дальнейшем вызовется и выполнит func.apply.

0 0 Ответить

М **Максим**

5 месяцев назад edited

Про последнюю задачу. Да и в общем.
1) Например: зачем в последней задаче, они делают это - func.apply(this, arguments); // (1), ведь this = window, смысл передавать глобальный контекст в эту функцию, если данная функция и так будет вызвана в глобальном контексте? Моя догадка: они просто используют данный метод чтобы передать в качестве аргумента функции псевдомассив - arguments. Если это не так, объясните пожалуйста

2) В последней задаче:
Я так понимаю, что при первом вызове, setTimeout заносится в стек и ждет истечения ms. Пока эти ms истекают, начинаются выполняться следующие вызовы функции, с другими аргумента. Во втором и третьем вызове, в условном блоке, мы сохраняем эти аргументы и контекст, опять же, зачем сохранять контекст, ведь он = window? Затем, после выполнения этих вызовов запускается setTimeout из первого вызова, в нем мы рекурсивно вызываем функцию wrapper, где передаем контекст и аргументы, но почему и контекст и аргументы передаются правильно, ведь setTimeout вызывается в самом первом вызове f1000(1) и тогда savedArgs и savedThis = undefined, какие либо значения они получили в следующих вызовах => f1000(2). f1000(3), неужели последующие вызовы, изменили лексическое окружение функции wrapper и для первого вызова?

показать больше

0 0 Ответить

Maxim Максим

5 месяцев назад edited

Проблема в целом понятна и ответить можно достаточно кратко.

Вот смотрите вы спрашиваете зачем this и т.п. но суть в том что ни один ваш пример не пользуется контекстом this, я имею ввиду вы не передаете его явно через операторы доступа к свойству или через вызов call.

Обычный вызов функции как правило не использует this (все будет зависеть от директивы use strict; если есть то this = undefined, если нет тогда this = глобальному объекту и это не всегда window)

Теперь чтобы показать вам как воспользоваться возможностями this тротлирующей функции воспользуйтесь следующим:

```
const data = {
  name: "Максим",
};

function printName() {
  console.log(this.name, arguments);
}
```

Ну и нам нужно теперь это вызывать под контекстом:

показать больше

1 0 Ответить

К **Кирилл Макаров**

6 месяцев назад

Зачем в примере , где
`worker.slow = cachingDecorator(worker.slow, hash);`
вообще используется хэш-функция?
Мало того, ее необязательно передавать аргументом, она и так будет в области видимости, так ведь еще и она сама по себе не нужна, потому что Map хавает всё в качестве ключей. Могли бы объяснить зачем так делать, раз так делаете. Ведь необходимости в этом нет.

0 0 Ответить

К **Кирилл Макаров** Кирилл Макаров

6 месяцев назад

Ага, дальше хоть понятно, что эта хэш-функция была как минимум, чтобы показать другую фичу - заимствование метода. Но тем не менее, есть правило "явное лучше неявного". Почему бы не выражаться определеннее. А то больше похоже не на учебник, а на конспект "для себя". Прокомментировать причины написания необязательных функций. Или сказать, что это необязательно, но мы коснемся некоторых особенностей дальше, поэтому приплели хэш-функцию. А то только задним числом читатель об этом может догадаться. Это не подход учебника.
Подушнил, но возмущение не без причины.

А еще раздражает, что ни с того, ни с сего берете и пишете func или arr, подразумевая под этим абстрактные функции и массивы. Опять приходится догадываться, откуда взялись эти func и arr, хотя вы могли бы указать, что это просто абстрактные функции для примера. Или писать Array.prototype , а не arr, так гораздо понятнее

0 0 Ответить

Алмаз Хусеинов → Кирилл Макаров

5 месяцев назад

в чем проблема не понять, функция hash используется для преобразования нескольких аргументов в ключ для map

0 0 Ответить

Рулер

6 месяцев назад

есть ли разница?

```
let obj = {
name: 'red',
}

function sayHi(...args) {
alert('Hi ' + this.name + args);
}

sayHi.apply(obj, ['! tyloh', ' chtoli', ' petuh'].map(item => alert(item)))
sayHi.call(obj, ['! tyloh', ' chtoli', ' petuh'].map(item => alert(item)))
```

оба возвращают одинаковый результат, несмотря на то что в операторе call мы использовали массив как 2 значение. Так в чем разница между call и apply

0 0 Ответить

Nachang → Рулер

месяц назад edited

Здравствуйте, Рулер.
Вероятно, вы уже сами ответили на свой вопрос, но всё же я оставлю комментарий.

В контексте вашего примера, разница между методами `call` и `apply` не проявилась, однако она всё ещё остаётся в способе передачи аргументов функции.

Разберём на примере:

```
function printArguments(...args) {
        console.log(args);
}

printArguments.apply(null, ["Fry", "Bender", "Hubert"], "Zoidberg");
// Вывод: ["Fry", "Bender", "Hubert"]

printArguments.call(null, ["Fry", "Bender", "Hubert"], "Zoidberg");
// Вывод: [ ["Fry", "Bender", "Hubert"], "Zoidberg" ]
```

Метод `apply` ожидает два аргумента: значение для `this` и массивоподобный объект аргументов функции. Любые дополнительные аргументы, будут проигнорированы.

Метод `call`, в свою очередь, принимает значение для `this` и последовательность аргументов произвольной длины.

Немного изменим пример:

```
function printArguments(...args) {
    console.log(args);
    console.log(args.toString()); // Добавим вывод строкового представления args
}

printArguments.apply(null, ["Fry", "Bender", "Hubert"]); // Изменим параметры
/* Вывод:
    [ "Fry", "Bender", "Hubert" ]
    "Fry,Bender,Hubert"
*/

printArguments.call(null, ["Fry", "Bender"], "Hubert"); // Изменим параметры
/* Вывод:
    [ [ "Fry", "Bender" ], "Hubert" ]
    "Fry,Bender,Hubert"
*/
```

В случае с методом `apply`, массив `args` состоит из трех элементов: `"Fry"`, `"Bender"`, `"Hubert"`.
В случае с методом `call`, массив `args` содержит два элемента: массив `["Fry", "Bender"]` и строку `"Hubert"`.
Несмотря на различную структуру объекта `args`, их строковые представления совпадают.

Это происходит потому, что при приведении массива к примитивному значению, получается строка, где элементы массива перечислены через запятую. Если элементом массива является объект, он также будет приведен к примитивному значению.

Рассмотрим подробнее:

```
// Результатом данного сравнения будет true
["Fry", "Bender", "Hubert"].toString() === [["Fry", "Bender"], "Hubert"].toString();
```

Массив `["Fry", "Bender", "Hubert"]` сразу преобразуется в строку `"Fry,Bender,Hubert"`.

В случае с `[["Fry", "Bender"], "Hubert"]`, первый элемент массива является объектом, поэтому его нужно привести к примитивному значению. После преобразования получаем `["Fry,Bender", "Hubert"]`, где каждый элемент является примитивным значением, и их можно объединить в `"Fry,Bender,Hubert"`.

В итоге сравнение сводится к `"Fry,Bender,Hubert" === "Fry,Bender,Hubert"`.
Это сравнение даёт результат `true`, так как оба строковых представления идентичны.

0 0 Ответить

Ъ

ъ ýˊ
7 месяцев назад edited

эту тему лучше изучить в YT ,например; иначе потеряете время и не только

0 0 Ответить

Neysel
7 месяцев назад

Нахрена так сложно, понятно ровно ни ху я

4 0 Ответить

В

Вячеслав Н.
7 месяцев назад edited

```
function cachingDecorator(func) {
let cache = new Map();

return function(x) {
if (cache.has(x)) { // если кеш содержит такой x,
return cache.get(x); // читаем из него результат
}

let result = func(x); // иначе, вызываем функцию

cache.set(x, result); // и кешируем (запоминаем) результат
return result;
};
}
```

Объясните пожалуйста внутри функции cachingDecorator есть безымянная функция в которую попадает параметр x (function x). Что попадает в этот параметр? Не могу понять

0 0 Ответить

Алмаз Хусеинов Вячеслав Н.
5 месяцев назад

ты создаешь функцию и возвращаешь ее, и в объекте в свойстве slow будет новый метод, и таким кодом worker.slow(5) ты как раз вызываешь этот возвращенный метод и туда передавать 5, x = 5

0 0 Ответить

Rewwoken
8 месяцев назад edited

Моя попытка решить 1 задачу. По сути ответ нужный, но некоторые нарушенные условия увидел уже в Планкере, а сам Планкер почему-то только после открытия ответа, лучше было бы его сразу дать, дабы была возможность проверить код.

показать больше

1 0 Ответить

5array
8 месяцев назад

Обьясните, пожалуйста, зачем мне нужен сохраненный контекст в throttle? разве он не будет в случае обычных фукций всегда равен window?

0 0 Ответить

Sergey RJS → 5array

7 месяцев назад edited

Контекст функции определяется в момент ее вызова. Если функция вызывается как метод объекта, `obj.function()`, то значение *this* будет равно объекту перед точкой. Если же функция вызывается без объекта, `function()`, то значение *this* будет равно `undefined` в строгом и `Window` в нестрогом режиме.

По условию задачи в **throttle** необходимо отложить вызов функции, поэтому контекст (аргументы функции и значение *this*) сохраняются в переменные, чтобы в дальнейшем вызвать функцию в нужном контексте.

0 0 Ответить

Татьяна Гончарова (thepupsa) → 5array

7 месяцев назад

согласна. я по-другому делала. результат одинаковый получился

```
function throttle(func, ms) {
let allowed = true;
let timer;
return function(...args) {
if (allowed) {
allowed = false;
func.apply(this, args);
timer = setTimeout(() => {
allowed = true;
clearTimeout(timer);
}, 1000);
}
}
}
```

0 0 Ответить

Sergey RJS → Татьяна Гончарова (thepupsa)

7 месяцев назад

Советую проверить ваш код с помощью тестов, которые приложены к задаче

0 0 Ответить

David

8 месяцев назад edited

На мой взгляд, слишком трудно усваивается данная подача, внимание теряется от сути, запоминая параметры и названия функций. Перетрудился ты с примерами, чувак

2 0 Ответить

Алмаз Хусеинов → David

5 месяцев назад

да примеры не сложные, просто в прошлых темах надо чуть чуть разобраться и все норм будет

0 0 Ответить

С **Сергей**
9 месяцев назад

Объясните пожалуйста, почему тут дебаунс реализован почти как тротл, это правильно? В большинстве источников, на которые наткнулся, дебаунс работает по другому принципу: декоратор передает вызов только по истечении задержки с последнего вызова функции, например при вводе текста в инпут он передаст вызов только по истечении времени после того как пользователь закончит вводить текст.

0 0 Ответить

Д **Денис Некрасов**
9 месяцев назад

Помогите с пониманием первого примера, про создание прозрачного кэширования. Я понимаю принцип работы, но вот пытаюсь разобраться в реализации. В функции cachindDecorator мы создаём ассоциативный массив cache, а потом возвращаем новую функции, внутри которой переменная cache видна по замыканию. Но после переопределения функции у неё же не будет доступа в переменной cache, она же создавалась локально внутри cachingDecorator, так почему же код всё равно работает. Что я упускаю из виду?

0 0 Ответить

Sergey RJS → Денис Некрасов
9 месяцев назад

Вам следует повторить главу про замыкания. При переопределении функции, как вы выразились, возвращается как раз внутренняя функция, которая имеет доступ к переменным *cachingDecorator*.

P.S. Я сам, когда начинал, декораторы осилил только со 2 раза, повторив все, что было до этого.

1 0 Ответить

Д **Денис Некрасов** → Sergey RJS
9 месяцев назад

Сергей, спасибо за наставление , обязательно повторю

1 0 Ответить

А **Армен Черноморских**
9 месяцев назад edited

Декоратор debounce

0 0 Ответить

Хунотик

9 месяцев назад

Гении, гигачады и просто те кто шарит, не могли бы вы объяснить одному дурачку (МНЕ), как вообще ВЫ мыслили при решении 4-й задачки. Хочу понять как умные люди (НЕ Я) думают и рассуждают при решении

0 0 Ответить

A **Anatolii Tarasov (Tarasov.Fron**

9 месяцев назад edited

Зафигачил дебоунс с сохранением аргументов вызовов и обращением к ним после истечения времени задержки! Критика приветствуется!

```
const debounce = (func, ms) => {
  let isReady = true;
  const argsArr = [];

  return function debouncedFunc() {
    if (!isReady) {
      if (arguments.length) {
        argsArr.push(Array.from(arguments));
      }
      setTimeout(() => debouncedFunc(), ms);
      return;
    }

    if (!argsArr.length && arguments.length) {
      func(...arguments);
```

показать больше

0 0 Ответить

R **RyTel**

9 месяцев назад

Какой Аналду это писал?! Особенно раздражают задачи и их решения!

3 0 Ответить

A **Anatolii Tarasov (Tarasov.Fron** RyTel

9 месяцев назад edited

Понятно сложно, но нужно. Например, в реакте под это целых два специальных хука написано: useTransition и useDeferredValue. Они используются в высоконагруженных приложениях.

0 0 Ответить

Загрузить ещё комментарии