JavaScript - продвинутый

arguments;
контекст this;
привязка контекста - apply, call, bind;
стрелочные функции;
функции высшего порядка;
паттерн декоратор;
паттерн фасад;

arguments в функциях

Объект arguments в JavaScript представляет собой массив-подобный объект, который доступен внутри всех функций и содержит все переданные ей аргументы.

arguments выглядит как массив, но это не полноценный массив (Array), а объект с доступом к элементам по индексу, начиная с 0.

arguments используется для доступа к всем переданным функции аргументам, включая те, которые не были объявлены как параметры функции.

Можно использовать arguments.length для получения количества переданных аргументов.

arguments не является массивом (Array), поэтому у него нет методов, доступных у массивов, таких как map, forEach и т. д.

В современных приложениях часто используются rest параметры (например, function example(...args) { ... }), которые предоставляют более удобный способ работы с переменным числом аргументов и являются наследником массива, облегчая их использование и манипуляции.

```
function example() {
  console.log(arguments[0]); // Выведет первый переданный аргумент
  console.log(arguments.length); // Выведет количество переданных аргументов
}

example('one', 'two', 'three');
```

Контекст this в функциях

Контекст this в JavaScript играет ключевую роль, определяя, как функции вызываются и как они имеют доступ к данным внутри себя. Вот основные моменты:

- **Глобальный контекст:** Вне любой функции this ссылается на глобальный объект. В браузере это объект window, в Node.js — global.
- **Как метод объекта:** Если функция является методом объекта (например, obj.method()), то this ссылается на объект, у которого вызван метод (obj).
- **Как функция:** Если функция вызывается как простая функция (func()), то this ссылается на глобальный объект (в строгом режиме undefined).
- **С использованием call, apply или bind:** Эти методы позволяют явно установить this для вызываемой функции.
- **Конструкторы:** Когда функция используется как конструктор (с помощью new), this указывает на новый экземпляр создаваемого объекта.
- **Стрелочные функции:** В стрелочных функциях this берется из контекста, в котором она была определена, а не из контекста вызова. Это делает их особенно полезными внутри методов объектов или при работе с замыканиями.

apply

Метод apply в JavaScript используется для вызова функции с указанием конкретного значения this и передачи аргументов в виде массива (или массивоподобного объекта).

Синтаксис: ***function.apply(thisArg, [argsArray])***

- **thisArg:** Объект, который будет использоваться как значение this внутри вызываемой функции.
- **argsArray:** Массив или массивоподобный объект, содержащий аргументы, которые будут переданы вызываемой функции.

```
function greet(greeting) {
  console.log(`${greeting}, ${this.name}!`);
}

const person = { name: 'Alice' };

// Вызываем функцию greet с контекстом this, установленным в объект person
greet.apply(person, ['Hello']);
```

Использование:

- apply полезен, когда нужно вызвать функцию с определенным контекстом this, который отличается от текущего.
- Это также удобно, когда количество аргументов заранее неизвестно или они должны быть переданы как массив.

call

Метод call в JavaScript, подобно методу apply, используется для вызова функции с указанием конкретного значения this, но в отличие от apply, аргументы передаются в виде списка, а не массива. Вот основные моменты по методу call:

Синтаксис: ***function.call(thisArg, arg1, arg2, ...)***

- **thisArg:** Объект, который будет использоваться как значение this внутри вызываемой функции.
- **arg1, arg2, ...:** Аргументы, которые будут переданы вызываемой функции как отдельные значения.

```
function greet(greeting) {
  console.log(`${greeting}, ${this.name}!`);
}

const person = { name: 'Alice' };

// Вызываем функцию greet с контекстом this, установленным в объект person
greet.call(person, 'Hello');
```

Использование:

- call позволяет вызвать функцию с определенным контекстом this.
- Аргументы передаются как отдельные значения, что может быть удобно, если количество аргументов заранее известно и они известны в момент вызова.

bind

Метод bind в JavaScript используется для создания новой функции, которая имеет определенное значение this, и, при необходимости, предопределенные аргументы. Вот основные аспекты метода bind:

Синтаксис: function.bind(thisArg, arg1, arg2, ...)

- **thisArg:** Объект, который будет использоваться как значение this внутри новой функции.
- **arg1, arg2, ...:** Аргументы, которые будут фиксированно переданы в новую функцию при ее вызове, вместе с аргументами, переданными при вызове.

```
function greet(greeting) {
  console.log(`${greeting}, ${this.name}!`);
}

const person = { name: 'Alice' };

// Создаем новую функцию с контекстом this, установленным в объект person
const greetAlice = greet.bind(person);

// Вызываем новую функцию, передавая аргумент
greetAlice('Hello');
```

Использование:

- bind создает новую функцию с указанным контекстом this.
- Фиксирует аргументы для этой новой функции, которые будут переданы вместе с любыми аргументами, переданными при вызове новой функции.

Стрелочные функции

Стрелочные функции в JavaScript — это синтаксический сахар для определения функций, который предлагает более краткую и читаемую запись по сравнению с обычными функциями.

Синтаксис:

- Стрелочная функция определяется с использованием стрелки (=>).
- Если функция принимает один аргумент, скобки вокруг аргумента можно опустить.
- Если функция выполняет только одно выражение, фигурные скобки и ключевое слово return также можно опустить.

```
// Базовый синтаксис
const add = (a, b) => {
  return a + b;
};

// Если функция принимает один аргумент
const square = x => x * x;

// Если функция выполняет одно выражение
const greet = name => `Hello, ${name}!`;
```

Стрелочные функции (особенности)

Отсутствие собственного контекста this:

Стрелочные функции не имеют собственного контекста this; вместо этого они заимствуют его из окружающего контекста, в котором они были определены. Это делает их особенно удобными внутри методов объектов или при работе с замыканиями.

Не могут быть использованы как конструкторы:

Стрелочные функции не могут использоваться с оператором new для создания экземпляров объектов. Попытка сделать это вызовет ошибку.

Не имеют своих собственных объектов arguments:

В стрелочных функциях отсутствует объект arguments, который доступен в обычных функциях. Вместо этого можно использовать rest параметры (...args), чтобы получить все переданные аргументы.

Функции высшего порядка

Функции высшего порядка (higher-order functions) — это функции, которые принимают другие функции в качестве аргументов или возвращают функции в качестве результатов. Они являются ключевой концепцией в функциональном программировании и широко используются в JavaScript.

Примеры функций высшего порядка:

- метод массива ***map***
- метод массива ***filter***
- метод массива ***reduce***
- метод массива ***some*** и ***every***

Паттерн декоратор

Декоратор (Decorator) — это паттерн проектирования, который позволяет динамически добавлять новую функциональность к объекту без изменения его структуры. В JavaScript декораторы могут применяться к классам, методам, а также к функциям.

Декоратор функции — это функция, которая принимает другую функцию в качестве аргумента и возвращает новую функцию, расширяющую или изменяющую поведение оригинальной функции.

Декоратор логирования

Декоратор логирования будет оборачивать любую функцию, чтобы при ее вызове логировать входные аргументы и возвращаемое значение.

Использование декоратора логирования позволяет легко добавлять функциональность логирования к любой функции, не изменяя её исходный код. Это делает код более модульным и упрощает его сопровождение.

```
// Декоратор логирования
function logFunctionCall(fn) {
  // Возвращаем новую функцию, которая оборачивает оригинальную функцию
  return function(...args) {
    // Логируем имя функции и её аргументы
    console.log(`Calling function ${fn.name} with arguments:`, args);
    // Вызываем оригинальную функцию с переданными аргументами
    const result = fn.apply(this, args);
    // Логируем результат вызова функции
    console.log(`Function ${fn.name} returned:`, result);
    // Возвращаем результат
    return result;
  };
}

// Оригинальная функция
function add(a, b) {
  return a + b;
}

// Создаем декорированную функцию
const loggedAdd = logFunctionCall(add);

// Используем декорированную функцию
const result = loggedAdd(2, 3);
console.log(`Result: ${result}`);
```

Декоратор кэширования (memoize)

Декоратор memoize используется для кэширования результатов функций, чтобы избежать повторных вычислений при повторных вызовах функции с теми же аргументами. Это может значительно улучшить производительность для функций, которые выполняют дорогие вычисления.

Декоратор memoize полезен для оптимизации производительности функций, выполняющих дорогостоящие вычисления. Он сохраняет результаты вызовов функций в кэше и использует их при последующих вызовах с теми же аргументами, избегая повторных вычислений.

```
// Декоратор кэширования
function memoize(fn) {
  // Объект для хранения кэша
  const cache = new Map();

  // Возвращаем новую функцию, которая оборачивает оригинальную функцию
  return function(...args) {
    // Создаем ключ для кэша на основе аргументов функции
    const key = JSON.stringify(args);

    // Проверяем, есть ли результат в кэше
    if (cache.has(key)) {
      // Если есть, возвращаем кэшированный результат
      return cache.get(key);
    }

    // Если нет, вызываем оригинальную функцию и сохраняем результат в кэше
    const result = fn.apply(this, args);
    cache.set(key, result);

    // Возвращаем результат
    return result;
  };
}

// Оригинальная функция с дорогими вычислениями
function slowFunction(num) {
  for (let i = 0; i < 1e9; i++) {} // Имитация долгих вычислений
  return num * 2;
}

// Создаем декорированную функцию
const memoizedSlowFunction = memoize(slowFunction);

// Используем декорированную функцию
console.log(memoizedSlowFunction(5)); // Медленно, т.к. результат вычисляется впервые
console.log(memoizedSlowFunction(5)); // Быстро, т.к. результат берется из кэша
```

Декоратор debounce

Декоратор debounce используется для управления частотой вызова функции. Он полезен, когда необходимо отложить выполнение функции до тех пор, пока не пройдет определенный интервал времени без последующих вызовов этой функции. Это помогает избежать лишних, частых вызовов функции, особенно в случаях событий, где функция может быть вызвана слишком часто.

Декоратор debounce полезен для управления частотой вызова функции, особенно в случаях, когда функция может быть вызвана слишком часто (например, в ответ на события пользовательского ввода или прокрутки). Он помогает избежать излишнего выполнения функции и улучшает производительность приложения.

```
// Декоратор debounce
function debounce(fn, delay) {
  let timerId;

  // Возвращаем новую функцию, которая оборачивает оригинальную функцию
  return function(...args) {
    // Если таймер уже установлен, очищаем его
    if (timerId) {
      clearTimeout(timerId);
    }

    // Устанавливаем новый таймер для вызова функции через задержку
    timerId = setTimeout(() => {
      fn.apply(this, args);
      timerId = null;
    }, delay);
  };
}

// Пример функции, которую будем декорировать
function saveInput(value) {
  console.log(`Saving value: ${value}`);
}

// Создаем декорированную функцию с задержкой в 500 миллисекунд
const debouncedSaveInput = debounce(saveInput, 500);

// Имитация ввода пользователя, вызывающего функцию слишком часто
debouncedSaveInput('First');
debouncedSaveInput('Second');
debouncedSaveInput('Third');

// После 500 миллисекунд вызовется только последний вызов функции
```

Декоратор throttling

Декоратор троттлинга (throttling) также используется для управления частотой вызова функции, но в отличие от декоратора debounce, который откладывает вызов функции до окончания задержки, декоратор троттлинга гарантирует, что функция будет вызвана не чаще, чем раз в определенный интервал времени. Это полезно, когда необходимо ограничить частоту выполнения функции, например, чтобы уменьшить нагрузку на сервер при множественных вызовах.

Декоратор троттлинга (throttle) полезен для ограничения частоты вызовов функции, особенно при обработке событий, которые могут происходить слишком часто. Это помогает снизить нагрузку на приложение и повысить его производительность.

```
// Декоратор троттлинга
function throttle(fn, delay) {
  let lastCall = 0;

  // Возвращаем новую функцию, которая оборачивает оригинальную функцию
  return function(...args) {
    const now = new Date().getTime();

    // Проверяем, прошло ли достаточно времени с последнего вызова функции
    if (now - lastCall >= delay) {
      fn.apply(this, args);
      lastCall = now;
    }
  };
}

// Пример функции, которую будем декорировать
function reportWindowSize() {
  console.log(`Window size: ${window.innerWidth} x ${window.innerHeight}`);
}

// Создаем декорированную функцию с задержкой в 500 миллисекунд
const throttledReportWindowSize = throttle(reportWindowSize, 500);

// Вызываем декорированную функцию при изменении размеров окна
window.addEventListener('resize', throttledReportWindowSize);
```

Паттерн фасад

Шаблон Facade (фасад) в JavaScript используется для предоставления простого интерфейса к сложной системе классов, функций или модулей. Он помогает скрыть сложность и предоставить более удобный интерфейс для взаимодействия с этой системой.

Пример реализации логики TODO с использованием паттерна фасад - ТЫК

Ресурсы

Декораторы и переадресация вызова, call/apply - ТЫК

Привязка контекста к функции - ТЫК

Повторяем стрелочные функции - ТЫК

Пример реализации TODO с использованием паттерна фасад - ТЫК