ЛАБОРАТОРНАЯ РАБОТА № 4

«Исследование основ реактивного программирования с использованием JavaScript»

1. Цель работы

Изучение особенностей реактивного подхода разработки ПО, получение практических навыков разработки ПО с использованием JavaScript.

2 Краткие теоретические сведения

Реактивное программирование представляет собой парадигму программирования, основанную на обработке асинхронных потоков данных и реакции на изменения в данных или события в реальном времени.

Основные принципы реактивного программирования включают в себя:

Асинхронность. Реактивное программирование предполагает асинхронную обработку данных и событий, что позволяет более эффективно управлять асинхронными операциями, такими как сетевые запросы, события пользовательского взаимодействия и другие асинхронные задачи.

Потоки данных. Реактивное программирование основано на работе с потоками данных, где данные могут быть произведены, преобразованы и потреблены в режиме реального времени. Это позволяет создавать реактивные цепочки операторов для обработки и преобразования данных на лету.

Обработка событий. Реактивное программирование предоставляет механизмы для обработки событий в реальном времени, таких как изменения данных, пользовательские действия или другие события, и реагирования на них с помощью реактивных операторов.

Реактивная модель. Реактивное программирование предполагает создание реактивной модели, где данные и события рассматриваются как потоки, и операторы могут быть использованы для обработки этих потоков данных.

Реактивное программирование предлагает ряд преимуществ:

Отзывчивость. Реактивное программирование позволяет создавать отзывчивые приложения, которые реагируют на изменения данных и событий в реальном времени. Пользователи могут видеть изменения в интерфейсе без задержек, что повышает качество пользовательского опыта.

Масштабируемость. Реактивное программирование позволяет обрабатывать большие объемы данных и событий эффективно, так как операции могут быть выполнены асинхронно и параллельно. Это делает реактивные приложения масштабируемыми и способными справляться с высокими нагрузками.

Гибкость. Реактивное программирование предлагает гибкость в обработке данных. Операторы могут быть комбинированы и переиспользуемы, что позволяет создавать сложные операции обработки данных из простых компонентов. Это делает код более модульным, легко поддерживаемым и расширяемым.

Упрощение сложных асинхронных операций. Реактивные операторы предлагают абстракцию для обработки сложных асинхронных операций, таких как асинхронные вызовы API, обработка событий, работа с потоками данных и других асинхронных задач. Это позволяет разработчикам упростить сложный асинхронный код и избежать проблем, таких как "callback hell", который может возникнуть при работе с асинхронными операциями.

Легкая интеграция. Реактивное программирование может быть легко интегрировано в различные типы приложений и архитектур, такие как клиент-серверные приложения, веб-приложения, мобильные приложения и другие. Реактивные библиотеки и фреймворки доступны на множестве языков программирования, таких как Java, JavaScript, Scala, Kotlin, и других, что делает их универсальными и гибкими для различных технологических стеков.

Реактивные приложения

- Реагируют на события
  - Слабая связность компонентов. Отправитель и получатель могут быть реализованы, не оглядываясь на детали того, как события распространяются в системе
  - Удобство распараллеливания вычислений. Неблокирующее асинхронное взаимодействие позволяет эффективнее использовать ресурсы
- Реагируют на повышение нагрузки. Фокус на масштабируемость, конкурентный доступ к общедоступным ресурсам сводится к минимуму
- Реагируют на сбои. Строятся отказоустойчивые системы с возможностью восстанавливаться на всех уровнях.
- Реагируют на пользователей. Гарантированное время отклика, не зависящее от нагрузки.

Основные архитектурные идеи

- Push-модель взаимодействия. Данные отправляются к своим потребителям, когда становятся доступными, вместо того чтобы впустую тратить ресурсы, постоянно запрашивая или ожидая данные.
- Неблокирующая асинхронная передача сообщений. Поток отправителя не блокируется в ожидании обработки сообщения получателем.
- Минимизация общедоступного изменяемого состояния. Позволяет избегать конкурентного доступа и синхронизации
- Соблюдение этих принципов на всех слоях приложения. «Приложение должно быть реактивным сверху донизу», иначе масштабирование упрётся в слабое звено.

Пример реактивного подхода разработки в JavaScript:

Рассмотрим задачу поиска дублей элементов побочной диагонали матрицы и вынесения их в отдельный массив.

```
import { from } from 'rxjs';
import { map, filter, toArray } from 'rxjs/operators';
// Исходная матрица
const matrix = [
  [6, 2, 3, 8],
  [5, 6, 0, 8],
  [9, 8, 7, 6],
  [0, 4, 3, 2]
];
// Создаем поток данных из элементов побочной диагонали матрицы
const diagonalStream = from(matrix).pipe(
   map((row, rowIndex) => row[matrix.length - 1 - rowIndex]) // Извлечение элементов побочной диагонали
);
// Создаем поток данных из дублирующихся элементов побочной диагонали
const duplicatesStream = diagonalStream.pipe(
  // Используем операторы RxJS для поиска дублирующихся элементов
  toArray(), // Преобразуем поток в массив
  map(arr => arr.reduce((acc, cur, idx, src) => {
    if (src.indexOf(cur) !== idx && acc.indexOf(cur) === -1) {
      acc.push(cur);
    }
    return acc;
  }, [])) // Поиск дублирующихся элементов
);
// Собираем дублирующиеся элементы побочной диагонали в отдельный массив
duplicatesStream.subscribe(duplicates => {
  console.log('Дублирующиеся элементы побочной диагонали:', duplicates);
});
```

Этот код использует RxJS для создания реактивного потока данных из элементов побочной диагонали матрицы. Затем применяется оператор фильтрации для выявления дублирующихся элементов. В конце, дублирующиеся элементы собираются в отдельный массив с помощью оператора toArray(), и выводятся в консоль.

4 Порядок выполнения

3.1 Изучить основные средства языка JavaScript для разработки программ с использованием реактивного подхода.

3.2 Выполнить две задачи из варианта на языке JavaScript согласно своему варианту.

3.3 Разработать тестовые примеры.

3.4 Выполнить отладку программ.

3.5 Сформулировать выводы, проанализировав разницу разработки программ с использованием декларативных и императивных парадигм.

3.6 Оформить отчет по проделанной работе.

3 Варианты заданий

Для каждого варианта требуется создать одноместный и двухмерный массивы. Для однокамерного массива нужно добавить минимум 10 элементов, для матрицы 16 элементов.

Вариант 1

Найти сумму всех элементов в массиве.

Удалить все строки матрицы, в которых есть отрицательные элементы.

Вариант 2

Найти наибольший элемент в массиве.

Проверить, содержит ли матрица магический квадрат (сумма элементов всех строк, столбцов и диагоналей одинакова).

Вариант 3

Найти индекс первого вхождения определенного элемента в массив.

Найти среднее арифметическое элементов в каждой строке матрицы.

Вариант 4

Посчитать количество четных элементов в массиве.

Поменять местами две заданные строки матрицы.

Вариант 5

Отсортировать массив по возрастанию.

Найти сумму всех элементов в двумерном массиве.

Вариант 6

Изменить порядок элементов массива на обратный.

Отсортировать строки матрицы по возрастанию суммы их элементов.

Вариант 7

Удалить все дубликаты из массива.

Посчитать сумму элементов каждого столбца и сохранить результаты в одномерный массив.

Вариант 8

Найти среднее арифметическое всех элементов массива.

Посчитать количество строк матрицы, в которых есть хотя бы один отрицательный элемент.

Вариант 9

Проверить, является ли массив палиндромом.

Найти наименьший элемент в двумерном массиве.

Вариант 10

Найти сумму элементов на нечетных позициях массива.

Найти сумму элементов главной диагонали матрицы.