МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ

**Ордена Трудового Красного Знамени**

**Федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Московский технический университет связи и информатики»**

Кафедра «Математическая Кибернетика и Информационные технологии»

Лабораторная работа №7

Выполнил: Студент группы

БВТ2203

Георгян Сергей

Москва

2023

Задание для выполнения лабораторной работы: Вам необходимо разработать приложение, которое считывает данные из исходного источника (например, файл, база данных или сетевой ресурс), применяет к ним различные операции с использованием Stream API, и сохраняет результаты в новый источник данных.

1. Создайте аннотацию @DataProcessor, которая будет использоваться для пометки методов обработки данных.

2. Создайте класс DataManager, который будет отвечать за многопоточную обработку данных. Этот класс должен иметь методы: o registerDataProcessor(Object processor): Регистрирует объект-обработчик данных с аннотацией @DataProcessor. o loadData(String source): Загружает данные из исходного источника. o processData(): Запускает многопоточную обработку данных, применяя методы с аннотацией @DataProcessor с использованием Stream API. o saveData(String destination): Сохраняет обработанные данные в новый источник.

3. Создайте несколько классов, представляющих различные обработчики данных, и пометьте их аннотацией @DataProcessor. Например, можно создать классы для фильтрации, трансформации и агрегации данных.

4. Используйте многопоточность из java.util.concurrent для эффективной обработки данных параллельно.

5. Протестируйте ваше приложение, загрузив данные из исходного источника, применив различные обработчики с помощью Stream API, и сохраните результаты в новый источник.

Ход работы:

import java.lang.annotation.Retention;  
import java.lang.annotation.RetentionPolicy;  
import java.nio.file.Files;  
import java.nio.file.Path;  
import java.nio.file.StandardOpenOption;  
import java.util.ArrayList;  
import java.util.List;  
import java.util.concurrent.ExecutorService;  
import java.util.concurrent.Executors;  
import java.util.stream.Collectors;  
public class Main {  
 public static void main(String[] args)  
 {  
 DataManager dataManager = new DataManager();  
  
 // 3. Создайте и зарегистрируйте обработчики данных  
 dataManager.registerDataProcessor(new DataFilter());  
 dataManager.registerDataProcessor(new DataTransformer());  
 dataManager.registerDataProcessor(new DataAggregator());  
  
 // 5. Загрузите, обработайте и сохраните данные  
 List<String> inputData = dataManager.loadData("source.txt");  
 List<String> processedData = dataManager.processData(inputData);  
 System.*out*.println(processedData);  
 dataManager.saveData(processedData, "data.txt");  
 }  
}  
  
  
  
// 1. Создайте аннотацию @DataProcessor  
@Retention(RetentionPolicy.*RUNTIME*)  
@interface DataProcessor {}  
  
@DataProcessor  
class DataFilter  
{  
 public List<String> filterData(List<String> data)  
 {  
 return data.stream().filter(s -> s.length() > 3).collect(Collectors.*toList*());  
 }  
}  
  
@DataProcessor  
class DataTransformer {  
 public List<String> transformData(List<String> data)  
 {  
 return data.stream().map(String::toUpperCase).collect(Collectors.*toList*());  
 }  
}  
  
@DataProcessor  
class DataAggregator  
{  
 public String aggregateData(List<String> data)  
 {  
 return data.stream().collect(Collectors.*joining*(", "));  
 }  
}  
  
// 2. Создайте класс DataManager  
class DataManager  
{  
 private final List<Object> dataProcessors = new ArrayList<>();  
  
 public void registerDataProcessor(Object processor)  
 {  
 if (processor.getClass().isAnnotationPresent(DataProcessor.class)) {  
 dataProcessors.add(processor);  
 }  
 }  
  
 public List<String> loadData(String source)  
 {  
 Path filePath = Path.*of*(source);  
  
 try  
 {  
 return Files.*readAllLines*(filePath);  
 }  
 catch (Exception e)  
 {  
 throw new RuntimeException("Error loading data from file", e);  
 }  
 }  
  
 // Метод обработки данных с использованием Stream API и многопоточности  
 public List<String> processData(List<String> data)  
 {  
 ExecutorService executorService = Executors.*newFixedThreadPool*(dataProcessors.size());  
  
 try  
 {  
 return dataProcessors.parallelStream()  
 .map(processor -> executorService.submit(() -> processWithAnnotation(processor, data)))  
 .map(future -> {  
 try  
 {  
 return future.get();  
 }  
 catch (Exception e)  
 {  
 throw new RuntimeException(e);  
 }  
 })  
 .flatMap(List::stream)  
 .collect(Collectors.*toList*());  
 }  
 finally  
 {  
 executorService.shutdown();  
 }  
 }  
  
 // Вспомогательный метод для обработки данных с использованием аннотации @DataProcessor  
 private List<String> processWithAnnotation(Object processor, List<String> data)  
 {  
 if (processor instanceof DataFilter) return ((DataFilter) processor).filterData(data);  
 else if (processor instanceof DataTransformer) return ((DataTransformer) processor).transformData(data);  
 else if (processor instanceof DataAggregator) return List.*of*(((DataAggregator) processor).aggregateData(data));  
 return data;  
 }  
  
 // Метод сохранения обработанных данных в новый источник  
 public void saveData(List<String> processedData, String destination)  
 {  
 Path filePath = Path.*of*(destination);  
  
 try  
 {  
 Files.*write*(filePath, processedData, StandardOpenOption.*CREATE*, StandardOpenOption.*WRITE*);  
 System.*out*.println("Data has been saved to: " + filePath);  
 } catch (Exception e) {  
 throw new RuntimeException("Error saving processed data", e);  
 }  
 }  
}

Файл 1 – программа для записи содержимого из одного файла в другой с фильтрацией данных

Вывод: Я научился работать с аннотациями