Министерство цифрового развития, связи и массовых коммуникаций

Российской Федерации

Ордена Трудового Красного Знамени

Федерально государственное бюджетное образовательное учреждение

высшего образования

МОСКОВСКИЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ СВЯЗИ И ИНФОРМАТИКИ

Кафедра «Математической кибернетики и информационных технологий»

Лабораторная работа № 8/

По дисциплине

«Информационные технологии и программирование»

Выполнил:

Студент группы БВТ2203

Бородин К.Н.

Москва 2023

ЗАДАНИЯ

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:

* registerDataProcessor(Object processor): Регистрирует объект- обработчик данных с аннотацией @DataProcessor.
* loadData(String source): Загружает данные из исходного источника.
* processData(): Запускает многопоточную обработку данных, применяя методы с аннотацией @DataProcessor с использованием Stream API.
* saveData(String destination): Сохраняет обработанные данные в новый источник.

1. Создайте несколько классов, представляющих различные обработчики данных, и пометьте их аннотацией @DataProcessor. Например, можно создать классы для фильтрации, трансформации и агрегации данных.
2. Используйте многопоточность из java.util.concurrent для эффективной обработки данных параллельно.
3. Протестируйте ваше приложение, загрузив данные из исходного источника, применив различные обработчики с помощью Stream API, и сохраните результаты в новый источник.

ХОД РАБОТЫ.

Часть 1.

Создаём аннотацию DataProcessor. Аннотация будем применяться к методам и использоваться во время выполнения программы.

import java.lang.annotation.ElementType;

import java.lang.annotation.Retention;

import java.lang.annotation.RetentionPolicy;

import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)

//указываем уровень сохранения аннотации. В данном случае

//мы указываем, что она будем использоваться во время

//выполнения программы

@Target(ElementType.METHOD)

//Место применение аннотации, в нашем случае она будем

//применять к методам

public @interface DataProcessor {

}

// с помощью @interface создаются пользовательские

//аннотации в java

Часть 2.

Создаём DataManager, который будет реализовывать много поточную обработку данных. У него будут методы для: добавления обработчик данных, загрузки данных из источника(из файла), многопоточной обработки данных с помощью обработчиков данных, получения аннотированного метода для обработки, применения этого аннотированного метода, и сохранение результатов в отдельные файлы.

import java.io.IOException;

import java.lang.reflect.Method;

import java.nio.file.Files;

import java.nio.file.Paths;

import java.util.ArrayList;

import java.util.List;

import java.util.concurrent.CompletableFuture;

import java.util.concurrent.ExecutorService;

import java.util.concurrent.Executors;

import java.util.concurrent.TimeUnit;

import java.util.stream.Collectors;

import java.util.stream.Stream;

public class DataManager {

    private List<Object> dataProcessors = new ArrayList<>();

    //массив для хранения обработчиков данных

    private List<Integer> inputData = new ArrayList<>();

    //Массив для сохранения входных данных

    private List<List<Integer>> processorResults = new ArrayList<>();

    //Массив для сохранения результатов работы различными обработчиками данных

    public void registerDataProcessor(Object processor) {

        //метод для добавления обработчика данных в список

        dataProcessors.add(processor);

    }

    public void loadData(String source) throws IOException {

        // Чтение данных из файла

        //Paths.get(source) создаёт из строки путь к файлу

        //Files.readAllLines чтение всех строк из файла

        //и возвращение их в виде списк строк

        List<String> lines = Files.readAllLines(Paths.get(source));

        // lines.stream() преобразование списка строк в поток строк

        //.flatMap метод используется для объединения всех потоков

        //слов в один общий поток

        //line -> Stream.of(line.split("\\s+")) для каждой строки line

        //метод split разбивает строку, использую пробле как разделитель

        //Stream.of мы передаем массив строк, полученный после разибения и создаём поток слов

        //.map(Integer::parseInt) к каждому элементу потока мы применяем выражение

        //Integer::parseInt сокращённый синтаксис для передачи метода как параметра

        //мы ссылаемся на статический метод

        //в конце преобразуем в список

        inputData = lines.stream()

                .flatMap(line -> Stream.of(line.split("\\s+")))

                .map(Integer::parseInt)

                .toList();

    }

    public void processData() {

        // Определяем количество потоков, в нашем случае оно соответствует количеству обработчиков данных

        int numberOfThreads = 3;

        //создаём сервис для управления потоками

        ExecutorService executorService = Executors.newFixedThreadPool(numberOfThreads);

        List<CompletableFuture<List<Integer>>> futures = new ArrayList<>();

        //список, в котором находяться задачи. Каждая задача представляет собой результат

        // в виде списка интов.

        for (Object processor : dataProcessors) {

            CompletableFuture<List<Integer>> future = CompletableFuture.supplyAsync(() -> processWithProcessor(processor), executorService);

            futures.add(future);

        }

        //Итерируемся по всем обработчикам данных

        //CompletableFuture.supplyAsync создаёт и возвращает CompletableFuture, который будет выполнен в отдельном поток

        //() -> processWithProcessor(processor) лямбда выражение не принимает аргументы

        // в теле лямбда выражения мы вызывает метод, для работы с данными

        CompletableFuture<Void> combinedFuture = CompletableFuture.allOf(futures.toArray(new CompletableFuture[0]));

        // создаём объект типа CompletableFuture<Void>, когда завершаются все поток.

        //массив futures преобразуется в массив типа CompletableFuture, указано в скобках. [0] пустой массив, в нашем случае используем как шаблон

        //метод AllOf ожидает завершения всех потоков. он создает новые CompletableFuture, передавая ему все остальные после завершения работы CompletableFuture

        try {

            combinedFuture.get();

            //метод блокирует текущий поток, ожидая завершенеия всех асинхронных задач в этой переменно

        } catch (Exception e) {

            e.printStackTrace();

        } finally {

            executorService.shutdown();

            //сервис больше не принимает новые задачи и завершает предыдщуеи

        }

        // Получаем результаты из CompletableFuture

        processorResults = futures.stream() //преобразует список в поток объектов

                .map(CompletableFuture::join)

                //преобразуем каждый элемент при помощи метода .join, т.е. преобразует в поток результатов каждой задачи

                .collect(Collectors.toList());

                //сбор результатов из поток в список

    }

    private List<Integer> processWithProcessor(Object processor) {

        List<Integer> result = new ArrayList<>();

        //создаём объект, в котором будем хранить результат обработки данных

        for (Method method : processor.getClass().getDeclaredMethods()) {

            //итерируемся по методам в процессоре

            if (method.isAnnotationPresent(DataProcessor.class)) {

                //проверка, явялется ли текущий метод аннотированным согласна DataProccesor

                processWithMethod(processor, method, inputData, result);

                //вызов метода для обработки данных

            }

        }

        return result;

    }

    private void processWithMethod(Object processor, Method method, List<Integer> input, List<Integer> result) {

        try {

            method.invoke(processor, input, result);

            //вызов метода обработчика данных

        } catch (Exception e) {

            e.printStackTrace();

        }

    }

    public void saveResults(String destination) {

        for (int i = 0; i < processorResults.size(); i++) {

            //итерация по объектам массива

            List<Integer> result = processorResults.get(i);

            //получаем результат обработки данных для текущего процесса

            try {

                Files.write(Paths.get(destination + "\_result" + i + ".txt"), result.stream().map(Object::toString).toList());

                //запись в файл.

            } catch (IOException e) {

                e.printStackTrace();

            }

        }

    }

}

Часть 3.

Обработчик данных, который агрегирует весь массив и возвращает количество элементов:

import java.util.List;

public class CountDataProccessor {

    @DataProcessor

    public static void countNumbers(List<Integer> input, List<Integer> output) {

        output.add(input.size());

    }

}

Обработчик данных, который фильтрует данные и оставляет только те, что подходят по шаблону:

import java.util.List;

public class FilterDataProccesor {

    @DataProcessor

    public static void filterNumbers(List<Integer> input, List<Integer> output) {

        output.addAll(input.stream().filter(num -> num > 5).toList());

        //возвращем число, если оно больше 5

    }

}

Обработчик данных для трансформации данных:

import java.util.List;

public class SquareDataProcessor {

    @DataProcessor

    public static void squareNumbers(List<Integer> input, List<Integer> output) {

        output.addAll(input.stream().map(num -> num \* num).toList());

        //возводим в квадрат.

    }

}

Часть 4.

Программа для тестирования приложения. Мы создаём менеджер обработчиков данных, добавляем в него обработчики, открываем файл, выполняем обработку и сохраняем результаты.

import java.io.IOException;

public class Main {

    public static void main(String[] args) {

        DataManager dataManager = new DataManager();

        // добавляем обработчики данных

        dataManager.registerDataProcessor(new FilterDataProccesor());

        dataManager.registerDataProcessor(new SquareDataProcessor());

        dataManager.registerDataProcessor(new CountDataProccessor());

        // Загружаем данные из файла

        try {

            dataManager.loadData("input.txt");

        } catch (IOException e) {

            e.printStackTrace();

            // Дополнительная обработка исключения по вашему усмотрению

        }

        // Обрабатываем данные

        dataManager.processData();

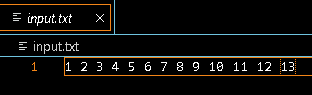
        // Сохраняем обработанные данные в новый файл

        dataManager.saveResults("output.txt");

    }

}

Пример входных данных:



Выходные файлы:

