Национальный исследовательский университет ИТМО

Факультет ПИиКТ

**ЛАБОРАТОРНАЯ РАБОТА 1**

по дисциплине «Реактивная Java»

Предметная область: «Отзывы на электронные товары»

**Студенты:**

Асташин С. С.

Ершова А. И.

Группы P4114 и P4115

**Преподаватель:**

Гаврилов Антон Валерьевич



Санкт-Петербург, 2024

**Текст задания**

1. Написать для согласованной предметной области как минимум 3 базовых класса и генераторы объектов. Генератор должен уметь создавать указанное количество различных объектов сответствующего класса со случайными (но при этом валидными) характеристиками. Класс, представляющий собой массовый объект должен обязательно содержать поля следющих типов:

- один из примитивов (int, long, double)

- String

- дата/время (LocalDate, LocalTime, ...)

- enum

- record

- массив или коллекция

Остальные поля - произвольные, какие нужны для предметной области.

Два оставшихся класса должны представлять собой дополнительные атрибуты и характиристики массового класса.

2. С помощью генератора создать коллекцию объектов.

3. Написать код, реализующий расчет согласованных агрегированных статических данных тремя способами:

3.1. Итерационным циклом по коллекции

3.2. Конвейером с помощью Stream API на базе коллекторов из стандартной библиотеки

3.3. Конвейером с помощью собственного коллектора.

4. Для каждого варианта измерить время выполнения, зафиксировав моменты начала и окончания расчета для количества элементов в коллекции - 5000, 50000 и 250000. Время измерять с помощью методов класса System или Instant.

**Описание предметной области**

В качестве предметной области был выбран электронный магазин, в котором можно оставлять отзывы на товары. Были созданы следующие базовые классы:

* *Product* – товар (массовый объект)
* *Review* – отзыв на товар
* *Manufacturer* – компания-производитель товара

Класс *Product* является массовым объектом, а классы *Review* и *Manufacturer* – вспомогательные. Ниже представлен исходный код всех базовых классов.

*// Product.java*  
  
@Getter  
@ToString  
@AllArgsConstructor  
public class Product {  
 private int id;  
 private double price; *// примитив* private String name; *// строка* private LocalDate releaseDate; *// дата* private Country country; *// enum* private Manufacturer manufacturer; *// record* private List<Review> reviews; *// коллекция*}

*// Review.java*  
  
@Getter  
@ToString  
@AllArgsConstructor  
public class Review {  
 private int id;  
 private ZonedDateTime dateTime; *// дата и время создания* private String user; *// автор* private int rating; *// оценка от 1 до 5*}

*// Manufacturer.java*

public record Manufacturer(int id, String name, LocalDate foundationDate) {  
}

**Создание генераторов**

Для создания генераторов был использован абстрактный класс *AbstractGenerator*, исходный код которого представлен ниже.

public abstract class AbstractGenerator<T> {  
 protected int id = 1;  
 protected final Faker faker = new Faker();  
  
 protected abstract T generate();  
  
 public List<T> generateList(int count) {  
 List<T> list = new ArrayList<>();  
 for (int i = 0; i < count; i++) {  
 list.add(generate());  
 }  
 return list;  
 }  
}

Все созданные генераторы объектов базовых классов являются наследниками данного класса и переопределяют метод *generate*. Следует отметить, что для генерации значений полей базовых классов была использована библиотека *net.datafaker*.

**Расчет данных**

В качестве расчетного показателя был выбран средний рейтинг товара для каждого производителя. Ниже представлен исходный код класса *Calculation*, в котором реализованы 3 метода расчета данного показателя.

public class Calculation {  
 *// Итерационный цикл* public static Map<Manufacturer, Double> avgRatingWithLoop(List<Product> products, List<Manufacturer> manufacturers) {  
 Map<Manufacturer, Double> averageRatings = new HashMap<>();  
  
 for (Manufacturer manufacturer : manufacturers) {  
 int totalReviews = 0;  
 int totalRating = 0;  
  
 for (Product product : products) {  
 if (product.getManufacturer().equals(manufacturer)) {  
 for (Review review : product.getReviews()) {  
 totalRating += review.getRating();  
 totalReviews++;  
 }  
 }  
 }

double averageRating = totalReviews > 0 ? (double) totalRating / totalReviews : 0.0;  
 averageRatings.put(manufacturer, averageRating);  
 }  
 return averageRatings;  
 }  
  
 *// Конвейер с помощью Stream API на базе коллекторов из стандартной библиотеки* public static Map<Manufacturer, Double> avgRatingWithPipeline(List<Product> products, List<Manufacturer> manufacturers) {  
 return manufacturers.stream()  
 .collect(Collectors.*toMap*(  
 manufacturer -> manufacturer,  
 manufacturer -> {  
 List<Review> reviews = products.stream()  
 .filter(product -> product.getManufacturer().equals(manufacturer))  
 .flatMap(product -> product.getReviews().stream())  
 .toList();  
  
 return reviews.isEmpty() ? 0.0 : reviews.stream()  
 .mapToInt(Review::getRating)  
 .average()  
 .orElse(0.0);  
 }  
 ));  
 }  
  
 *// Собственный коллектор* public static Map<Manufacturer, Double> avgRatingWithCollector(List<Product> products, List<Manufacturer> manufacturers) {  
 return products.stream()  
 .collect(new AverageRatingCollector(manufacturers));  
 }  
  
 public static void printResult(Map<Manufacturer, Double> result) {  
 for (Manufacturer manufacturer : result.keySet()) {  
 System.*out*.printf("For manufacturer \"%s\" average product rating is %f\n", manufacturer.name(), result.get(manufacturer));  
 }  
 }  
}

Ниже представлен исходный код классов *AverageRatingAccumulator* и *AverageRatingCollector*, использованных в методе *avgRatingWithCollector*.

*// AverageRatingAccumulator.java*  
  
public class AverageRatingAccumulator {  
 private int totalRating = 0;  
 private int reviewCount = 0;  
  
 void addRating(int rating) {  
 totalRating += rating;  
 reviewCount++;  
 }  
  
 AverageRatingAccumulator combine(AverageRatingAccumulator other) {  
 this.totalRating += other.totalRating;  
 this.reviewCount += other.reviewCount;  
 return this;  
 }  
  
 double getAverage() {  
 return reviewCount > 0 ? (double) totalRating / reviewCount : 0.0;  
 }  
}

*// AverageRatingCollector.java*

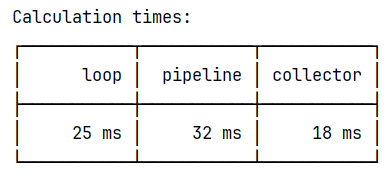
@RequiredArgsConstructor  
public class AverageRatingCollector implements Collector<Product, Map<Manufacturer, AverageRatingAccumulator>, Map<Manufacturer, Double>> {  
 private final List<Manufacturer> manufacturers;  
  
 @Override  
 public Supplier<Map<Manufacturer, AverageRatingAccumulator>> supplier() {  
 return () -> {  
 Map<Manufacturer, AverageRatingAccumulator> map = new HashMap<>();  
 manufacturers.forEach(manufacturer -> map.put(manufacturer, new AverageRatingAccumulator()));  
 return map;  
 };  
 }

@Override  
 public BiConsumer<Map<Manufacturer, AverageRatingAccumulator>, Product> accumulator() {  
 return (map, product) -> {  
 Manufacturer manufacturer = product.getManufacturer();  
 AverageRatingAccumulator accumulator = map.get(manufacturer);  
  
 if (accumulator != null) { *// Убедимся, что производитель существует* product.getReviews().forEach(review -> accumulator.addRating(review.getRating()));  
 }  
 };  
 }  
  
 @Override  
 public BinaryOperator<Map<Manufacturer, AverageRatingAccumulator>> combiner() {  
 return (map1, map2) -> {  
 map2.forEach((manufacturer, accumulator) ->  
 map1.merge(manufacturer, accumulator, AverageRatingAccumulator::combine));  
 return map1;  
 };  
 }  
  
 @Override  
 public Function<Map<Manufacturer, AverageRatingAccumulator>, Map<Manufacturer, Double>> finisher() {  
 return map -> map.entrySet().stream()  
 .collect(Collectors.*toMap*(Map.Entry::getKey, entry -> entry.getValue().getAverage()));  
 }  
  
 @Override  
 public Set<Characteristics> characteristics() {  
 return Set.*of*(Characteristics.*UNORDERED*);  
 }  
}

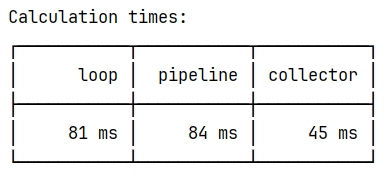
**Измерение времени расчета**

Ниже представлен результат работы программы. Расчет производился при следующих параметрах:

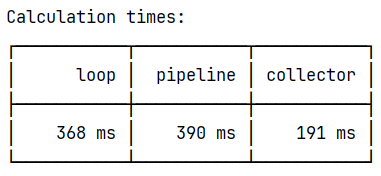
* Количество генерируемых производителей товара – 3.
* Количество генерируемых отзывов о каждом товаре – 100.
* Количество генерируемых товаров – 5000, 50000, 250000.



*Рисунок 1 – Результат работы программы при обработке 5000 товаров*



*Рисунок 2 – Результат работы программы при обработке 50000 товаров*



*Рисунок 3 – Результат работы программы при обработке 250000 товаров*

Как можно заметить, расчет с помощью использования собственного коллектора приблизительно в 2 раза быстрее расчета с использованием других методов.

**Вывод**

В ходе данной лабораторной работы были написаны базовые классы и генераторы объектов для согласованной с преподавателем предметной области «Отзывы на электронные товары». Также, был произведен расчет среднего рейтинг товара для каждого производителя с помощью следующих методов: итеративного цикла, конвейера стандартных коллекторов Stream API и собственного коллектора.

Время расчета при использовании первых двух методов оказалось приблизительно одинаковым, а расчет с использованием собственного коллектора оказался самым быстрым (в 2 раза быстрее других методов), поскольку собственный коллектор позволяет оптимизировать процесс аккумуляции и объединения данных, избегая ненужных промежуточных объектов и затрат на внутреннюю обработку, что обеспечивает повышение скорости. Путем раннего создания структуры для всех производителей он также позволяет управлять комбинацией результатов более эффективно, минимизируя количество обращений к памяти и сохраняя производительность даже на больших наборах данных.