Национальный исследовательский университет ИТМО

Факультет ПИиКТ

**ЛАБОРАТОРНАЯ РАБОТА 2**

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. В один из методов, использовавшийся для сбора статистики, добавить возможность задать задержку, имитирующую задержку получения результата, например из базы данных. К примеру, был метод getName(), в который нужно добавить параметр getName(long delay).

2. Заменить последовательный стрим, собирающий статистику из лабораторной №1, на параллельный. Поменять итоговую коллекцию, где собирается результат, на соответствующую потокобезопасную. Измерить производительность для разного количества элементов с дополнительной задержкой и без задержки. Для случаев с задержкой и без найти количество элементов, при котором сбор статистики последовательным и параллельным стримами дают одинаковую скорость выполнения.

3. Оптимизировать параллельный сбор статистики, реализовав собственный ForkJoinPool или Spliterator. Измерить производительность своего варианта.

4. Измерения производительности выполнять с помощью фреймворка JMH.

**Добавление задержки**

Для задержки было принято решение использовать метод *getReviews* класса *Product*. Итоговый исходный код этого класса представлен ниже.

@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* public List<Review> getReviewsWithDelay(long delay) {  
 try {  
 Thread.*sleep*(delay);  
 return getReviews();  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 }  
}

**Создание сплитератора**

Для оптимизации работы с параллельным сбором статистики было принято решение создать собственный сплитератор, код которого представлен ниже.

@RequiredArgsConstructor  
public class ProductSpliterator implements Spliterator<Product> {  
 private final List<Product> products;  
 private int current = 0;  
  
 @Override  
 public boolean tryAdvance(Consumer<? super Product> action) {  
 if (current < products.size()) {  
 action.accept(products.get(current++));  
 return true;  
 }  
 return false;  
 }  
  
 @Override  
 public Spliterator<Product> trySplit() {  
 int currentSize = products.size() - current;  
 if (currentSize < 10) { *// Минимальный размер для разделения* return null;  
 }  
 int splitPos = current + currentSize / 2;  
 List<Product> splitList = products.subList(current, splitPos);  
 current = splitPos;  
 return new ProductSpliterator(splitList);  
 }  
  
 @Override  
 public long estimateSize() {  
 return products.size() - current;  
 }  
  
 @Override  
 public int characteristics() {  
 return *ORDERED* | *SIZED* | *SUBSIZED* | *NONNULL* | *IMMUTABLE*;  
 }  
}

**Использование JMH**

Для использования фреймворка JMH был создан класс *BenchmarkCalculation,* содержащий методы, запускающие расчет при различных параметрах. Ниже представлена исходный код данного класса, в котором опущена часть методов.

@Fork(1)  
@Threads(8)  
@State(Scope.*Benchmark*)  
@Warmup(iterations = 0)  
@Measurement(iterations = 3, time = 1)  
@BenchmarkMode(Mode.*AverageTime*)  
@OutputTimeUnit(TimeUnit.*MILLISECONDS*)  
public class BenchmarkCalculation {  
 @Param({"5000", "25000", "50000"})  
 private int productCount;  
 private List<Manufacturer> manufacturers;  
 private List<Product> products;  
  
 @Setup(Level.*Trial*)public void generate() {  
 int manufacturerCount = 3;  
 int reviewCount = 10;  
  
 ManufacturerGenerator manufacturerGenerator = new ManufacturerGenerator();  
 manufacturers = manufacturerGenerator.generateList(manufacturerCount);  
 ProductGenerator productGenerator = new ProductGenerator(manufacturers, reviewCount);  
 products = productGenerator.generateList(productCount);  
 }@Benchmark  
 @Group("\_1\_SequentialStreamNoDelay")  
 public Map<Manufacturer, Double> seqCollectorNoDelay() {  
 return Calculation.*avgRatingWithSeqCollector*(products, manufacturers, 0);  
 }  
  
 @Benchmark()  
 @Group("\_1\_SequentialStreamNoDelay")  
 public Map<Manufacturer, Double> seqPipelineNoDelay() {  
 return Calculation.*avgRatingWithSeqPipeline*(products, 0);  
 }

*// ...*  
}

**Измерение времени расчета**

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

* Количество генерируемых производителей товара – 3.
* Количество генерируемых отзывов о каждом товаре – 10.
* Количество генерируемых товаров – 5000, 25000, 50000.

Изображение выглядит как текст, документ, меню, снимок экрана

Автоматически созданное описание

*Рисунок 1 – Результат работы программы (часть 1)*

Изображение выглядит как текст, снимок экрана, документ, число

Автоматически созданное описание

*Рисунок 2 – Результат работы программы (часть 2)*

Ниже представлены эти же данные в виде таблиц, поскольку понять что-либо по данным изображениям проблематично.

*Таблица 1 – Результат работы программы без задержки*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Метод расчета | Кол-во элементов | Sequential | Parallel | Parallel + Spliterator |
| Pipeline | 5000 | 3,011 | 6,929 | 7,488 |
| 25000 | 16,375 | 39,293 | 39,885 |
| 50000 | 34,054 | 72,068 | 83,290 |
| Collector | 5000 | 2,886 | 3,917 | 3,639 |
| 25000 | 16,352 | 22,346 | 20,245 |
| 50000 | 33,049 | 44,356 | 42,113 |

*Таблица 2 – Результат работы программы с задержкой 1 мс*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Метод расчета | Кол-во элементов | Sequential | Parallel | Parallel + Spliterator |
| Pipeline | 5000 | 10005,546 | 7544,790 | 6343,481 |
| 25000 | 49958,792 | 37096,192 | 37034,187 |
| 50000 | 98661,043 | 60447,431 | 81151,520 |
| Collector | 5000 | 9987,961 | 5470,454 | 5226,254 |
| 25000 | 49494,733 | 27577,672 | 28185,166 |
| 50000 | 98188,627 | 48321,585 | 56387,845 |

**Вывод**

В ходе данной лабораторной работы была добавлена возможность добавить задержку при получении списка отзывов из объекта. Были произведены замеры времени расчета метрики из лабораторной работы 1 с использованием:

* последовательных стримов;
* параллельных стримов;
* параллельных стримов с собственным сплитератором.

При отсутствии задержки параллельные стримы (и со сплитератором, и без) показали время примерно в 2 раза больше, чем последовательные, что может быть обусловлено использованием большего количества операций “под капотом”, чтобы обеспечить параллельную обработку.

При добавлении задержки в 1 мс наблюдается обратная ситуация. Время расчета сокращается, поскольку ожидание происходит параллельно, а не последовательно. параллельные.