10.8. Метод reduce

Метод reduce выполняет терминальные операции сведения, возвращая некоторое значение - результат операции. Он имеет следующие формы:

```
1  Optional<T> reduce(BinaryOperator<T> accumulator)
2  T reduce(T identity, BinaryOperator<T> accumulator)
3  U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner)
```

Первая форма возвращает результат в виде объекта Optional<T>. Например, вычислим произведение набора чисел:

```
1  import java.util.stream.Stream;
2  import java.util.Optional;
3
4  public class Program {
5
6      public static void main(String[] args) {
7
8          Stream<Integer> numbersStream = Stream.of(1,2,3,4,5,6);
9          Optional<Integer> result = numbersStream.reduce((x,y)->x*y);
10         System.out.println(result.get()); // 720
11     }
12 }
```

Объект BinaryOperator<T> представляет функцию, которая принимает два элемента и выполняет над ними некоторую операцию, возвращая результат. При этом метод reduce сохраняет результат и затем опять же применяет к этому результату и следующему элементу в наборе бинарную операцию. Фактически в данном случае мы получим результат, который будет равен: n1 op n2 op n3 op n4 op n5 op n6, где op - это операция (в данном случае умножения), а n1, n2, ... - элементы из потока.

Затем с помощью метода get() мы можем получить собственно результат вычислений: result.get()

Или еще один пример - объединение слов в предложение:

```
1  Stream<String> wordsStream = Stream.of("мама", "мыла", "раму");
2  Optional<String> sentence = wordsStream.reduce((x,y)->x + " " + y);
3  System.out.println(sentence.get());
```

Вторая версия метода reduce() принимает два параметра:

```
1  T reduce(T identity, BinaryOperator<T> accumulator)
```

Первый параметр - T identity - элемент, который предоставляет начальное значение для функции из второго параметра, а также предоставляет значение по умолчанию, если поток не имеет элементов.

Второй параметр - BinaryOperator<T> accumulator, как и первая форма метода reduce, представляет ассоциативную функцию, которая запускается для каждого элемента в потоке и принимает два параметра. Первый параметр представляет промежуточный результат функции, а второй параметр - следующий элемент в потоке. Фактически код этого метода будет равноценен следующей записи:

```
1  T result = identity;
2  for (T element : this stream)
3      result = accumulator.apply(result, element)
4  return result;
```

То есть при первом вызове функция accumulator в качестве первого параметра принимает значение identity, а в качестве второго параметра - первый элемент потока. При втором вызове первым параметром служит результат первого вызова функции accumulator, а вторым параметром - второй элемент в потоке и так далее. Например:

```
1  Stream<Integer> numberStream = Stream.of(-4, 3, -2, 1);
2  int identity = 1;
3  int result = numberStream.reduce(identity, (x,y)->x * y);
4  System.out.println(result); // 24
```

Фактически здесь выполняется следующая цепь операций: identity op n1 op n2 op n3 op n4...

В предыдущих примерах тип возвращаемых объектов совпадал с типом элементов, которые входят в поток. Однако это не всегда удобно. Возможно, мы захотим возвратить результат, тип которого отличается от типа объектов потока. Например, пусть у нас есть следующий класс Phone, представляющий телефон:

```
1  class Phone{
2
3      private String name;
4      private int price;
5
```

```
    public Phone(String name, int price){
        this.name=name;
        this.price=price;
    }

    public String getName() { return name; }

    public int getPrice() { return price; }
}
```

И мы хотим найти сумму цен тех телефонов, у которых цена меньше определенного значения. Для этого используем третью версию метода reduce:

```
Stream<Phone> phoneStream = Stream.of(
        new Phone("iPhone 6 S", 54000), new Phone("Lumia 950", 45000),
        new Phone("Samsung Galaxy S 6", 40000), new Phone("LG G 4", 32000));

int sum = phoneStream.reduce(0,
        (x,y)-> {
            if(y.getPrice()<50000) return x + y.getPrice();
            else return x + 0;
        },
        (x, y)->x+y);
System.out.println(sum); // 117000
```

Опять же здесь в качестве первого параметра идет значение по умолчанию - 0. Второй параметр производит бинарную операцию, которая получает промежуточное значение - суммарную цену текущего и предыдущего телефонов. Третий параметр представляет бинарную операцию, которая суммирует все промежуточные вычисления.

10.9. Тип Optional

Ряд операций сведения, такие как min, max, reduce, возвращают объект Optional<T>. Этот объект фактически обертывает результат операции. После выполнения операции с помощью метода get() объекта Optional мы можем получить его значение:

```
import java.util.Optional;
import java.util.ArrayList;
import java.util.Arrays;
public class Program {

    public static void main(String[] args) {

        ArrayList<Integer> numbers = new ArrayList<Integer>();
        numbers.addAll(Arrays.asList(new Integer[]{1,2,3,4,5,6,7,8,9}));
        Optional<Integer> min = numbers.stream().min(Integer::compare);
        System.out.println(min.get()); // 1
    }
}
```

Но что, если поток не содержит вообще никаких данных:

```
// список numbers пустой
ArrayList<Integer> numbers = new ArrayList<Integer>();
Optional<Integer> min = numbers.stream().min(Integer::compare);
System.out.println(min.get()); // java.util.NoSuchElementException
```

В этом случае программа выдаст исключение java.util.NoSuchElementException. Что мы можем сделать, чтобы избежать выброса исключения? Для этого класс Optional предоставляет ряд методов.

Самой простой способ избежать подобной ситуации - это предварительная проверка наличия значения в Optional с помощью метода isPresent(). Он возвращает true, если значение присутствует в Optional, и false, если значение отсутствует:

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
Optional<Integer> min = numbers.stream().min(Integer::compare);
if(min.isPresent()){ System.out.println(min.get()); }
```

10.9.1. orElse

Метод orElse() позволяет определить альтернативное значение, которое будет возвращаться, если Optional не получит из потока какого-нибудь значения:

```
// пустой список
ArrayList<Integer> numbers = new ArrayList<Integer>();
```

```
3  Optional<Integer> min = numbers.stream().min(Integer::compare);
4  System.out.println(min.orElse(-1)); // -1
5
6  // непустой список
7  numbers.addAll(Arrays.asList(new Integer[]{4,5,6,7,8,9}));
8  min = numbers.stream().min(Integer::compare);
9  System.out.println(min.orElse(-1)); // 4
```

10.9.2. orElseGet

Метод orElseGet() позволяет задать функцию, которая будет возвращать значение по умолчанию:

```
1  import java.util.Optional;
2  import java.util.ArrayList;
3  import java.util.Arrays;
4  import java.util.Random;
5
6  public class Program {
7
8      public static void main(String[] args) {
9
10         ArrayList<Integer> numbers = new ArrayList<Integer>();
11         Optional<Integer> min = numbers.stream().min(Integer::compare);
12         Random rnd = new Random();
13         System.out.println(min.orElseGet(()->rnd.nextInt(100)));
14     }
15 }
```

В данном случае возвращаемое значение генерируется с помощью метода nextInt класса Random, который возвращает случайное число.

10.9.3. orElseThrow

Еще один метод - orElseThrow позволяет сгенерировать исключение, если Optional не содержит значения:

```
1  ArrayList<Integer> numbers = new ArrayList<Integer>();
2  Optional<Integer> min = numbers.stream().min(Integer::compare);
3  // генеррация исключения IllegalStateException
4  System.out.println(min.orElseThrow(IllegalStateException::new));
```

10.9.4. Обработка полученного значения

Метод ifPresent() определяет действия со значением в Optional, если значение имеется:

```
1  ArrayList<Integer> numbers = new ArrayList<Integer>();
2  numbers.addAll(Arrays.asList(new Integer[]{4,5,6,7,8,9}));
3  Optional<Integer> min = numbers.stream().min(Integer::compare);
4  min.ifPresent(v->System.out.println(v)); // 4
```

В метод ifPresent передается функция, которая принимает один параметр - значение из Optional. В данном случае полученное минимальное число выводится на консоль. Но если бы массив numbers был бы пустым, и соответственно Optional не сдержало бы никакого значения (на консоль ничего не выводится), то никакой ошибки бы не было.

Метод ifPresentOrElse() позволяет определить альтернативную логику на случай, если значение в Optional отсутствует:

```
1  ArrayList<Integer> numbers = new ArrayList<Integer>();
2  Optional<Integer> min = numbers.stream().min(Integer::compare);
3  min.ifPresentOrElse(
4      v -> System.out.println(v),
5     () -> System.out.println("Value not found")
6  );
```

В метод ifPresentOrElse передается две функции. Первая обрабатывает значение в Optional, если оно присутствует. Вторая функция представляет действия, которые выполняются, если значение в Optional отсутствует.

10.10. Метод collect

Большинство операций класса Stream, которые модифицируют набор данных, возвращают этот набор в виде потока. Однако бывают ситуации, когда хотелось бы получить данные не в виде потока, а в виде обычной коллекции, например, ArrayList или HashSet. И для этого у класса Stream определен метод collect. Первая версия метода принимает в качестве параметра функцию преобразования к коллекции:

```
1  <R,A> R collect(Collector<? super T,A,R> collector)
```

Параметр R представляет тип результата метода, параметр T - тип элемента в потоке, а параметр A - тип промежуточных накапливаемых данных. В итоге параметр collector представляет функцию преобразования потока в коллекцию.

Эта функция представляет объект Collector, который определен в пакете *java.util.stream*. Мы можем написать свою реализацию функции, однако Java уже предоставляет ряд встроенных функций, определенных в классе Collectors:

- toList(): преобразование к типу List
- toSet(): преобразование к типу Set
- toMap(): преобразование к типу Map

Например, преобразуем набор в потоке в список:

```
1   import java.util.ArrayList;
2   import java.util.Collections;
3   import java.util.List;
4   import java.util.stream.Collectors;
5
6   public class Program {
7
8       public static void main(String[] args) {
9
10          List<String> phones = new ArrayList<String>();
11          Collections.addAll(phones, "iPhone 8", "HTC U12", "Huawei Nexus 6P",
12                  "Samsung Galaxy S9", "LG G6", "Xiaomi MI6", "ASUS Zenfone 2",
13                  "Sony Xperia Z5", "Meizu Pro 6", "Lenovo S850");
14
15          List<String> filteredPhones = phones.stream()
16                  .filter(s->s.length()<10)
17                  .collect(Collectors.toList());
18
19          for(String s : filteredPhones){ System.out.println(s); }
20      }
21  }
```

Использование метода toSet() аналогично.

```
1   Set<String> filteredPhones = phones.stream()
2               .filter(s->s.length()<10)
3               .collect(Collectors.toSet());
```

Для применения метода toMap() надо задать ключ и значение. Например, пусть у нас есть следующая модель:

```
1   class Phone{
2       private String name;
3       private int price;
4
5       public Phone(String name, int price){
6           this.name=name;
7           this.price=price;
8       }
9
10      public String getName() { return name; }
11
12      public int getPrice() { return price; }
13  }
```

Теперь применим метод toMap():

```
1   import java.util.Map;
2   import java.util.stream.Collectors;
3   import java.util.stream.Stream;
4
5   public class Program {
6       public static void main(String[] args) {
7
8           Stream<Phone> phoneStream = Stream.of(
9               new Phone("iPhone 8", 54000), new Phone("Nokia 9", 45000),
10              new Phone("Samsung Galaxy S9", 40000), new Phone("LG G6", 32000));
```

```
11
12          Map<String, Integer> phones = phoneStream
13              .collect(Collectors.toMap(p->p.getName(), t->t.getPrice()));
14
15          phones.forEach((k,v)->System.out.println(k + " " + v));
16      }
17  }
18  class Phone{
19      private String name;
20      private int price;
21
22      public Phone(String name, int price){
23          this.name=name;
24          this.price=price;
25      }
26
27      public String getName() { return name; }
28      public int getPrice() { return price; }
29  }
```

Лямбда-выражение p->p.getName() получает значение для ключа элемента, а t->t.getPrice() - извлекает значение элемента.

Если нам надо создать какой-то определенный тип коллекции, например, HashSet, то мы можем использовать специальные функции, которые определены в классах-коллекций. Например, получим объект HashSet:

```
1   import java.util.HashSet;
2   import java.util.stream.Collectors;
3   import java.util.stream.Stream;
4
5   public class Program {
6
7       public static void main(String[] args) {
8
9           Stream<String> phones = Stream.of("iPhone 8", "HTC U12", "Huawei Nexus 6P",
10              "Samsung Galaxy S9", "LG G6", "Xiaomi MI6", "ASUS Zenfone 2",
11              "Sony Xperia Z5", "Meizu Pro 6", "Lenovo S850");
12
13          HashSet<String> filteredPhones = phones.filter(s->s.length()<12).
14                              collect(Collectors.toCollection(HashSet::new));
15
16          filteredPhones.forEach(s->System.out.println(s));
17      }
18  }
```

Выражение HashSet::new представляет функцию создания коллекции. Аналогичным образом можно получать другие коллекции, например, ArrayList:

```
1   ArrayList<String> result = phones.collect(Collectors.toCollection(ArrayList::new));
```

Вторая форма метода collect имеет три параметра:

```
1   <R> R collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator, BiConsumer<R,R> combiner)
```

- supplier: создает объект коллекции
- accumulator: добавляет элемент в коллекцию
- combiner: бинарная функция, которая объединяет два объекта

Применим эту версию метода collect:

```
1   import java.util.ArrayList;
2   import java.util.stream.Collectors;
3   import java.util.stream.Stream;
4
5   public class Program {
6
7       public static void main(String[] args) {
8
9           Stream<String> phones = Stream.of("iPhone 8", "HTC U12", "Huawei Nexus 6P",
```

```
                "Samsung Galaxy S9", "LG G6", "Xiaomi MI6", "ASUS Zenfone 2",
                "Sony Xperia Z5", "Meizu Pro 6", "Lenovo S850");

        ArrayList<String> filteredPhones = phones.filter(s->s.length()<12)
            .collect(
                ()->new ArrayList<String>(), // создаем ArrayList
                (list, item)->list.add(item), // добавляем в список элемент
                (list1, list2)-> list1.addAll(list2)); // добавляем в список другой список

        filteredPhones.forEach(s->System.out.println(s));
    }
}
```

10.11. Группировка

Чтобы сгруппировать данные по какому-нибудь признаку, нам надо использовать в связке метод collect() объекта Stream и метод Collectors.groupingBy(). Допустим, у нас есть следующий класс:

```
class Phone{
    private String name;
    private String company;
    private int price;

    public Phone(String name, String comp, int price){
        this.name=name;
        this.company=comp;
        this.price = price;
    }

    public String getName() { return name; }
    public int getPrice() { return price; }
    public String getCompany() { return company; }
}
```

И, к примеру, у нас есть набор объектов Phone, которые мы хотим сгруппировать по компании:

```
import java.util.List;
import java.util.Map;
import java.util.stream.Stream;
import java.util.stream.Collectors;

public class Program {

    public static void main(String[] args) {

        Stream<Phone> phoneStream = Stream.of(new Phone("iPhone X", "Apple", 600),
            new Phone("Pixel 2", "Google", 500),
            new Phone("iPhone 8", "Apple",450),
            new Phone("Galaxy S9", "Samsung", 440),
            new Phone("Galaxy S8", "Samsung", 340));

        Map<String, List<Phone>> phonesByCompany = phoneStream.collect(
                Collectors.groupingBy(Phone::getCompany));

        for(Map.Entry<String, List<Phone>> item : phonesByCompany.entrySet()){
            System.out.println(item.getKey());

            for(Phone phone : item.getValue()){ System.out.println(phone.getName()); }
            System.out.println();
        }
    }
}
```

Консольный вывод:

```
Google
Pixel 2

Apple
```

```
iPhone X
iPhone 8

Samsung
Galaxy S9
Galaxy S8
```

Итак, для создания групп в метод phoneStream.collect() передается вызов функции Collectors.groupingBy(), которая с помощью выражения Phone::getCompany группирует объекты по компании. В итоге будет создан объект Map, в котором ключами являются названия компаний, а значениями - список связанных с компаниями телефонов.

10.11.1. Метод Collectors.partitioningBy

Метод Collectors.partitioningBy() имеет похожее действие, только он делит элементы на группы по принципу, соответствует ли элемент определенному условию. Например:

```
1  Map<Boolean, List<Phone>> phonesByCompany = phoneStream.collect(
2            Collectors.partitioningBy(p->p.getCompany()=="Apple"));
3
4  for(Map.Entry<Boolean, List<Phone>> item : phonesByCompany.entrySet()){
5     System.out.println(item.getKey());
6
7     for(Phone phone : item.getValue()){ System.out.println(phone.getName()); }
8     System.out.println();
9  }
```

В данном случае с помощью условия p->p.getCompany()=="Apple" мы смотрим, принадлежит ли телефон компании Apple. Если телефон принадлежит этой компании, то он попадает в одну группу, если нет, то в другую.

10.11.2. Метод Coollectors.counting

Метод Collectors.counting применяется в Collectors.groupingBy() для вычисления количества элементов в каждой группе:

```
1  Map<String, Long> phonesByCompany = phoneStream.collect(
2        Collectors.groupingBy(Phone::getCompany, Collectors.counting()));
3
4  for(Map.Entry<String, Long> item : phonesByCompany.entrySet()){
5
6     System.out.println(item.getKey() + " - " + item.getValue());
7  }
```

Консольный вывод:

```
Google -1
Apple - 2
Samsung - 2
```

10.11.3. Метод Collectors.summing

Метод Collectors.summing применяется для подсчета суммы. В зависимости от типа данных, к которым применяется метод, он имеет следующие формы: summingInt(), summingLong(), summingDouble(). Применим этот метод для подсчета стоимости всех смартфонов по компаниям:

```
1  Map<String, Integer> phonesByCompany = phoneStream.collect(
2        Collectors.groupingBy(Phone::getCompany, Collectors.summingInt(Phone::getPrice)));
3
4  for(Map.Entry<String, Integer> item : phonesByCompany.entrySet()){
5
6     System.out.println(item.getKey() + " - " + item.getValue());
7  }
```

С помощью выражения Collectors.summingInt(Phone::getPrice)) мы указываем, что для каждой компании будет вычислять совокупная цена всех ее смартфонов. И поскольку вычисляется результат - сумма для значений типа int, то в качестве типа возвращаемой коллекции используется тип Map<String, Integer>

Консольный вывод:

```
Google - 500
Apple - 1050
Samsung - 780
```

10.11.4. Методы maxBy и minBy

Методы maxBy и minBy применяются для подсчета минимального и максимального значения в каждой группе. В качестве параметра эти методы принимают функцию компаратора, которая нужна для сравнения значений. Например, найдем для каждой компании телефон с минимальной ценой:

```
1   Map<String, Optional<Phone>> phonesByCompany = phoneStream.collect(
2         Collectors.groupingBy(Phone::getCompany,
3               Collectors.minBy(Comparator.comparing(Phone::getPrice))));
4
5   for(Map.Entry<String, Optional<Phone>> item : phonesByCompany.entrySet()){
6
7      System.out.println(item.getKey() + " - " + item.getValue().get().getName());
8   }
```

Консольный вывод:

```
Google - Pixel 2
Apple - iPhone 8
Samsung - Galaxy S8
```

В качестве возвращаемого значения операции группировки используется объект Map<String, Optional<Phone>>. Опять же поскольку группируем по компаниям, то ключом будет выступать строка, а значением - объект Optional<Phone>.

10.11.5. Метод summarizing

Методы summarizingInt() / summarizingLong() / summarizingDouble() позволяют объединить в набор значения соответствующих типов:

```
1   import java.util.IntSummaryStatistics;
2   //.....................................
3
4    Map<String, IntSummaryStatistics> priceSummary = phoneStream.collect(
5      Collectors.groupingBy(Phone::getCompany,
6          Collectors.summarizingInt(Phone::getPrice)));
7
8   for(Map.Entry<String, IntSummaryStatistics> item : priceSummary.entrySet()){
9
10     System.out.println(item.getKey() + " - " + item.getValue().getAverage());
11  }
```

Метод Collectors.summarizingInt(Phone::getPrice)) создает набор, в который помещаются цены для всех телефонов каждой из групп. Данный набор инкапсулируется в объекте IntSummaryStatistics. Соответственно если бы мы применяли методы summarizingLong() или summarizingDouble(), то соответственно бы получали объекты LongSummaryStatistics или DoubleSummaryStatistics.

У этих объектов есть ряд методов, который позволяют выполнить различные атомарные операции над набором:

- getAverage(): возвращает среднее значение
- getCount(): возвращает количество элементов в наборе
- getMax(): возвращает максимальное значение
- getMin(): возвращает минимальное значение
- getSum(): возвращает сумму элементов
- accept(): добавляет в набор новый элемент

В данном случае мы получаем среднюю цену смартфонов для каждой группы.

Консольный вывод:

```
Google - 500.0
Apple - 525.0
Samsung - 390.0
```

10.11.6. Метод mapping

Метод mapping позволяет дополнительно обработать данные и задать функцию отображения объектов из потока на какой-нибудь другой тип данных. Например:

```
1   Map<String, List<String>> phonesByCompany = phoneStream.collect(
2      Collectors.groupingBy(Phone::getCompany,
3          Collectors.mapping(Phone::getName, Collectors.toList())));
4
5   for(Map.Entry<String, List<String>> item : phonesByCompany.entrySet()){
6
7      System.out.println(item.getKey());
8      for(String name : item.getValue()){
9          System.out.println(name);
10     }
11  }
```

Выражение Collectors.mapping(Phone::getName, Collectors.toList()) указывает, что в группу будут выделятся названия смартфонов, причем группа будет представлять объект List.

10.12. Параллельные потоки

Кроме последовательных потоков Stream API поддерживает параллельные потоки. Распараллеливание потоков позволяет задействовать несколько ядер процессора (если целевая машина многоядерная) и тем самым может повысить производительность и ускорить вычисления. В то же время говорить, что применение параллельных потоков на многоядерных машинах однозначно повысит производительность - не совсем корректно. В каждом конкретном случае надо проверять и тестировать.

Чтобы сделать обычный последовательный поток параллельным, надо вызвать у объекта Stream метод parallel. Кроме того, можно также использовать метод parallelStream() интерфейса Collection для создания параллельного потока из коллекции.

В то же время если рабочая машина не является многоядерной, то поток будет выполняться как последовательный.

Применение параллельных потоков во многих случаях будет аналогично. Например:

```
1   import java.util.Optional;
2   import java.util.stream.Stream;
3
4   public class Program {
5
6       public static void main(String[] args) {
7
8           Stream<Integer> numbersStream = Stream.of(1, 2, 3, 4, 5, 6);
9           Optional<Integer> result = numbersStream.parallel().reduce((x,y)-> x*y);
10          System.out.println(result.get()); // 720
11      }
12  }
```

Однако не все функции можно без ущерба для точности вычисления перенести с последовательных потоков на параллельные. Прежде всего такие функции должны быть без сохранения состояния и ассоциативными, то есть при выполнении слева направо давать тот же результат, что и при выполнении справа налево, как в случае с произведением чисел. Например:

```
1   Stream<String> wordsStream = Stream.of("мама", "мыла", "раму");
2   String sentence = wordsStream.parallel().reduce("Результат:", (x,y)->x + " " + y);
3   System.out.println(sentence);
```

Результатом этой функции будет консольный вывод:

```
Результат: мама Результат: мыла Результат: раму
```

Данный вывод не является правильным. Если же мы не уверены, что на каком-то этапе работы с параллельным потоком он адекватно сможет выполнить какую-нибудь операцию, то мы можем преобразовать этот поток в последовательный посредством вызова метода sequential():

```
1   Stream<String> wordsStream = Stream.of("мама", "мыла", "раму", "hello world");
2   String sentence = wordsStream.parallel()
3           .filter(s->s.length()<10) // фильтрация над параллельным потоком
4           .sequential()
5           .reduce("Результат:", (x,y)->x + " " + y); // операция над последовательным потоком
6   System.out.println(sentence);
```

И возьмем другой пример:

```
1   Stream<Integer> numbersStream = Stream.of(1, 2, 3, 4, 5, 6);
2   Integer result = numbersStream.parallel().reduce(1, (x,y)->x * y);
3   System.out.println(result);
```

Фактически здесь происходит перемножение чисел. При этом нет разницы между 1 * 2 * 3 * 4 * (5 * 6) или 5 * 6 * 1 * (2 * 3) * 4. Мы можем расставить скобки любым образом, разместить последовательность чисел в любом порядке, и все равно мы получим один и тот же результат. То есть данная операция является ассоциативной и поэтому может быть распараллелена.

10.12.1. Вопросы производительности в параллельных операциях

Фактически применение параллельных потоков сводится к тому, что данные в потоке будут разделены на части, каждая часть обрабатывается на отдельном ядре процессора, и в конце эти части соединяются, и над ними выполняются финальные операции. Рассмотрим некоторые критерии, которые могут повлиять на производительность в параллельных потоках:

- Размер данных. Чем больше данных, тем сложнее сначала разделять данные, а потом их соединять.
- Количество ядер процессора. Теоретически, чем больше ядер в компьютере, тем быстрее программа будет работать. Если на машине одно ядро, нет смысла применять параллельные потоки.
- Чем проще структура данных, с которой работает поток, тем быстрее будут происходить операции. Например, данные из ArrayList легко использовать, так как структура данной коллекции предполагает последовательность несвязанных данных.

А вот коллекция типа LinkedList - не лучший вариант, так как в последовательном списке все элементы связаны с предыдущими/последующими. И такие данные трудно распараллелить.

- Над данными примитивных типов операции будут производиться быстрее, чем над объектами классов.

10.12.2. Упорядоченность в параллельных потоках

Как правило, элементы передаются в поток в том же порядке, в котором они определены в источнике данных. При работе с параллельными потоками система сохраняет порядок следования элементов. Исключение составляет метод forEach(), который может выводить элементы в произвольном порядке. И чтобы сохранить порядок следования, необходимо применять метод forEachOrdered:

```
1  phones.parallelStream()
2      .sorted()
3      .forEachOrdered(s->System.out.println(s));
```

Сохранение порядка в параллельных потоках увеличивает издержки при выполнении. Но если нам порядок не важен, то мы можем отключить его сохранение и тем самым увеличить производительность, использовав метод unordered:

```
1  phones.parallelStream()
2      .sorted()
3      .unordered()
4      .forEach(s->System.out.println(s));
```

10.13. Параллельные операции над массивами

В JDK 8 к классу Arrays было добавлено ряд методов, которые позволяют в параллельном режиме совершать обработку элементов массива. И хотя данные методы формально не входят в Stream API, но реализуют схожую функциональность, что и параллельные потоки:

- parallelPrefix(): вычисляет некоторое значение для элементов массива (например, сумму элементов)
- parallelSetAll(): устанавливает элементы массива с помощью лямбда-выражения
- parallelSort(): сортирует массив

Используем метод parallelSetAll() для установки элементов массива:

```
1  import java.util.Arrays;
2  public class Program {
3
4      public static void main(String[] args) {
5
6          int[] numbers = initializeArray(6);
7          for(int i: numbers) System.out.println(i);
8
9      }
10     public static int[] initializeArray(int size) {
11         int[] values = new int[size];
12         Arrays.parallelSetAll(values, i -> i*10);
13         return values;
14     }
15 }
```

В метод Arrays.parallelSetAll передается два параметра: изменяемый массив и функция, которая устанавливает элементы массива. Эта функция перебирает все элементы и в качестве параметра получает индекс текущего перебираемого элемента. Выражение i -> i*10 означает, что по каждому индексу в массиве будет хранится число, равное i*10. В итоге мы получим следующий вывод:

```
0
10
20
30
40
50
```

Рассмотрим более сложный пример. Пусть у нас есть следующий класс Phone:

```
1  class Phone{
2      private String name;
3      private int price;
4
5      public Phone(String name, int price){
6          this.name=name;
```

```
        this.price = price;
    }

    public String getName() { return name; }
    public void setName(String val) { this.name=val; }
    public int getPrice() { return price; }
    public void setPrice(int val) { this.price=val; }
}
```

Теперь произведем манипуляции с массивом объектов Phone:

```
Phone[] phones = new Phone[]{new Phone("iPhone 8", 54000),
    new Phone("Pixel 2", 45000),
    new Phone("Samsung Galaxy S9", 40000),
    new Phone("Nokia 9", 32000)};

Arrays.parallelSetAll(phones, i -> {
    phones[i].setPrice(phones[i].getPrice()-10000);
    return phones[i];
});

for(Phone p: phones) System.out.printf("%s - %d \n", p.getName(), p.getPrice());
```

Теперь лямбда-выражение в методе Arrays.parallelSetAll представляет блок кода. И так как лямбда-выражение должно возвращать объект, то нам надо явным образом использовать оператор return. В этом лямбда-выражении опять же функция получает индексы перебираемых элементов, и по этим индексам мы можем обратиться к элементам массива и их изменить. Конкретно в данном случае происходит уменьшение цены смартфонов на 10000 единиц. В итоге мы получим следующий консольный вывод:

```
iPhone 8 - 44000
Pixel 2 - 35000
Samsung Galaxy S9 - 30000
Nokia 9 - 22000
```

10.13.1. Сортировка

Отсортируем массив чисел в параллельном режиме:

```
int[] nums = {30, -4, 5, 29, 7, -8};
Arrays.parallelSort(nums);
for(int i: nums) System.out.println(i);
```

Метод Arrays.parallelSort() в качестве параметра принимает массив и сортирует его по возрастанию:

```
-8
-4
5
7
29
30
```

Если же нам надо как-то по-другому отсортировать объекты, например, по модулю числа, или у нас более сложные объекты, то мы можем создать свой компаратор и передать его в качестве второго параметра в Arrays.parallelSort(). Например, возьмем выше определенный класс Phone и создадим для него компаратор:

```
import java.util.Arrays;
import java.util.Comparator;
public class Program {

    public static void main(String[] args) {

        Phone[] phones = new Phone[]{new Phone("iPhone 8", 54000),
        new Phone("Pixel 2", 45000),
        new Phone("Samsung Galaxy S9", 40000),
        new Phone("Nokia 9", 32000)};

        Arrays.parallelSort(phones,new PhoneComparator());

         for(Phone p: phones) System.out.println(p.getName());
    }
}
class PhoneComparator implements Comparator<Phone>{
```

```
18
19      public int compare(Phone a, Phone b){
20          return a.getName().toUpperCase().compareTo(b.getName().toUpperCase());
21      }
22  }
```

10.13.2. Метод parallelPrefix

Метод parallelPrefix() походит для тех случаев, когда надо получить элемент массива или объект того же типа, что и элементы массива, который обладает некоторыми признаками. Например, в массиве чисел это может быть максимальное, минимальное значения и т.д. Например, найдем произведение чисел:

```
1  int[] numbers = {1, 2, 3, 4, 5, 6};
2  Arrays.parallelPrefix(numbers, (x, y) -> x * y);
3
4  for(int i: numbers) System.out.println(i);
```

Мы получим следующий результат:

```
1
2
6
24
120
720
```

То есть, как мы видим из консольного вывода, лямбда-выражение из Arrays.parallelPrefix, которое представляет бинарную функцию, получает два элемента и выполняет над ними операцию. Результат операции сохраняется и передается в следующий вызов бинарной функции.