нетология

Stream API. Потоки, повторные вызовы, основные методы

Григорий
Вахмистров

Григорий Вахмистров

Backend Developer в Tennisi.bet

План занятия

1. Stream API
2. Промежуточные операции
3. Терминальные операции
4. Итоги
5. Домашнее задание

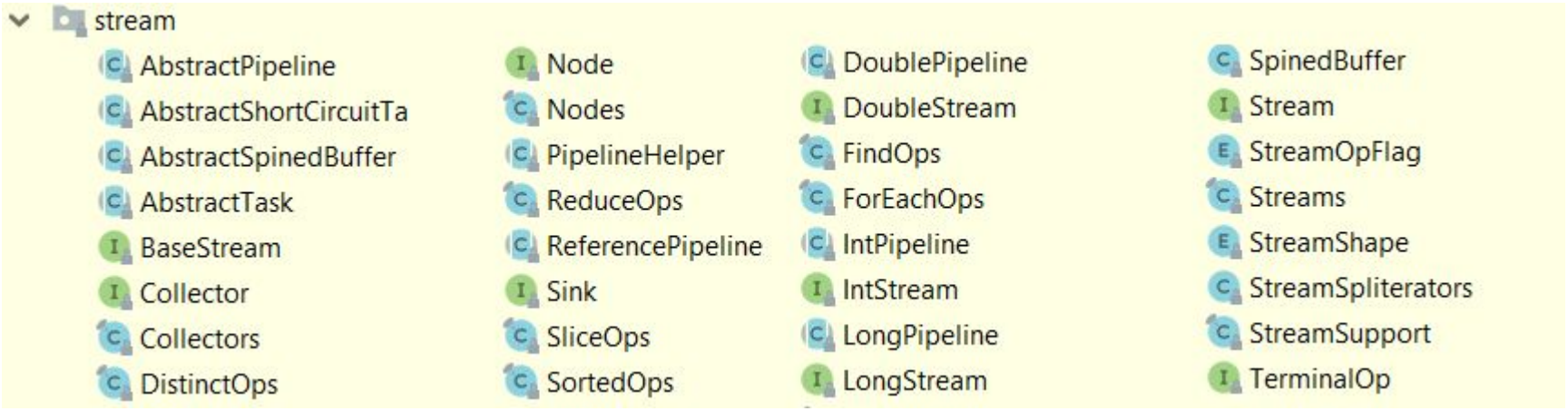
Stream API

Определение

Начиная с JDK 8 в Java появился новый API* - **Stream API**.

Его задача - упростить работу с наборами данных, в частности, упростить операции ***фильтрации***, ***сортировки***, ***модификации*** и другие манипуляции с данными.

Вся основная функциональность данного API сосредоточена в пакете ***java.util.stream***.



*API (Application Programming Interface) - описание способов (набор классов, процедур, функций, структур или констант), которыми одна компьютерная программа может взаимодействовать с другой программой.

Жизненный цикл

Ключевым понятием в Stream API является **стрим** (поток данных).

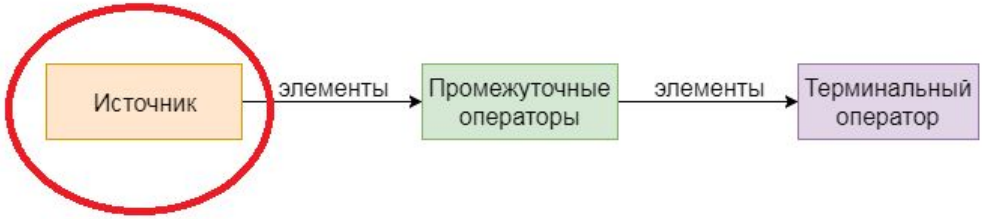
Применительно к Stream API стрим представляет ***канал*** **передачи данных** из источника данных. В качестве источника могут выступать файлы, массивы и коллекции.

Фактически жизненный цикл стрима проходит следующие три стадии:

- создание стрима
- применение к стриму ряда промежуточных операций
- применение к стриму завершающей операции с целью получения результата

Создание стрима

Стримы опираются и берут данные из источника: коллекции, массива или метода.



Возможные способы создания стримов:

- Пустой стрим: Stream.empty()
- Стрим из List: list.stream()
- Стрим из Map: map.entrySet().stream()
- Стрим из массива: Arrays.stream(array)
- Стрим из указанных элементов: Stream.of("1", "2", "3")

Рассмотрим следующий пример:

Создать стрим из листа, состоящего из трех элементов.

```
List<String> list = new ArrayList<>();
list.add("One");
list.add("Two");
list.add("Three");
Stream stream = list.stream();
```

Типы стримов

Кроме объектных стримов Stream<T>, существуют специальные стримы для примитивных типов:

- **IntStream** - для **int**,
- **LongStream** - для **long**,
- **DoubleStream** - для **double**.

Для **boolean, byte, short** и **char** специальных стримов нет. Вместо них можно использовать **IntStream**, *а затем приводить к нужному типу*.

Для **float** тоже придётся воспользоваться **DoubleStream**.

Примитивные стримы полезны, так как не нужно тратить время на боксинг/анбоксинг, к тому же у них есть ряд специальных операторов, упрощающих жизнь.

Операции со стримами

Операции со стримами можно разделить на:

- **Промежуточные** — операции, возвращающие трансформированный стрим. К возвращенному стриму также можно применить ряд промежуточных операций.

  К промежуточным операциям относят: фильтрацию, преобразование, сортировку, удаление дубликатов, ограничение и многие другие.

- **Терминальные** — терминальные операции возвращают конкретный результат и являются завершающими.

  К терминальным операциям относят: перебор элементов, поиск, нахождение минимума и максимума, расчет количества и т.д.

Операции со стримами

Рассмотрим следующий пример:

Найти в массиве количество всех чисел больше 0.

```
long count = IntStream.of(-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5)
        .filter(w -> w > 0)
        .count();
System.out.println(count);
```

1) Промежуточный метод **filter** принимает стрим чисел и возвращает преобразованный стрим чисел больше 0.
2) Метод **count** представляет терминальную операцию и возвращает количество чисел в стриме.

Стримы vs Коллекции

Важно понимать *отличие коллекций от стримов*:

- **Стримы не хранят элементы**. Элементы, используемые в стримах, могут храниться в коллекции, либо при необходимости могут быть напрямую сгенерированы.
- **Операции со стримами не изменяют источника данных**, а лишь возвращают новый стрим с результатами этих операций.
- **Для стримов характерно отложенное выполнение**. Выполнение всех операций со стримом происходит лишь тогда, когда выполняется терминальная операция и возвращается конкретный результат, а не новый стрим.

И снова лямбды

Одной из отличительных черт Stream API является применение **лямбда-выражений**, которые позволяют значительно сократить запись выполняемых действий.

Рассмотрим следующий пример:

Отфильтровать из массива значения больше 90, полученные значения увеличить на 10. Вывести на экран результирующий лист.

Пример решения до Java 8:

```
int[] input = {50, 60, 70, 80, 90, 100};
List<Integer> result = new ArrayList<>();
int count = 0;
for (int x : input) {
   if (x >= 90) continue;
   x += 10;
   count++;
   if (count > 3) break;
   result.add(x);
}
System.out.println(result);
```

Тот же код с использованием stream api:

```
int[] input = {50, 60, 70, 80, 90, 100};
List<Integer> result =
Arrays.stream(input)
       .filter(x -> x < 90)
       .map(x -> x + 10)
       .limit(3)
       .boxed()
       .collect(Collectors.toList());
System.out.println(result);
```

Еще больше примеров

Рассмотрим следующий пример:

Отфильтровать лист строк и вывести в консоль заглавными буквами те элементы, длина которых равна 3.

Решение:

```
List<String> list = Arrays.asList("My", "Pen", "Is", "Black");
list.stream()
        .filter(x -> x.length() == 3)
        .map(String::toUpperCase)
        .forEach(System.out::println); // PEN
```

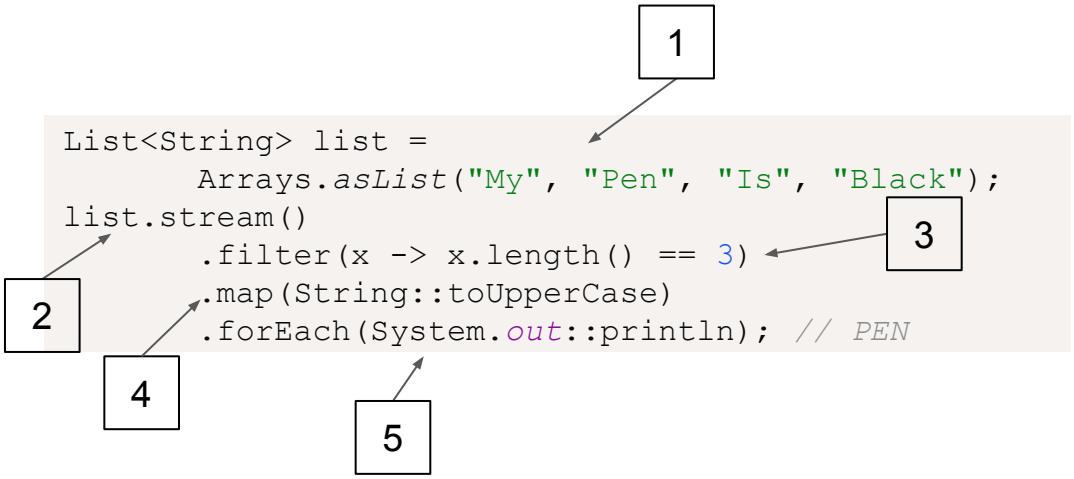
Как это сделать? На следующем слайде рассмотрим пошаговое решение задачи.

Еще больше примеров

Как решаем?

1. Создаём коллекцию **list** и заполняем ее тестовыми данными.

2. Создаём объект **stream**, используя **list** в качестве источника.



3. Метод **filter** — промежуточный оператор, элемент коллекции **x** в котором фильтруется согласно правилу, указанному после символа **->**.

4. Метод **map** – промежуточный оператор, работающий по аналогии с **filter**, но необходимый для преобразования данных в коллекции.

5. Метод **forEach** - терминальный оператор перебора элементов **x**, вызывающий System.out.println(x).

А что там под капотом?

1. Метод **stream()** создает экземпляр класса **Stream**.

2. Каждая промежуточная операция работает с экземпляром стрима и при этом создает новый экземпляр стрима на выходе.

3. Нельзя дважды использовать один и тот же экземпляр стрима. Он одноразовый!

4. Терминальная операция не дает на выходе стрима, но дает какой-либо результат.

```
List<String> list =
       Arrays.asList("My", "Pen", "Is", "Black");
Stream<String> stream_1 = list.stream();
Stream<String> stream_2 = stream_1.filter(x -> x.length() == 3);
Stream<String> stream_3 = stream_2.map(String::toUpperCase);
stream_3.forEach(System.out::println); // PEN
```

Имейте ввиду

- Обработка стримом коллекции не начнётся до тех пор, пока не будет вызван терминальный оператор.

```
list.stream().filter(s -> s > 5);
```

- Экземпляр стрима нельзя использовать более одного раза.

```
Stream<String> stream = list.stream();
stream.filter(x-> x.toString().length() == 3).forEach(System.out::println);
stream.forEach(x -> System.out.println(x));
```

Exception in thread "main" java.lang.IllegalStateException: stream has already been operated upon or closed

- Промежуточных операторов, вызванных на одном стриме, может быть множество, а терминальный оператор - только один.

```
stream.filter(x -> x.toString().length() == 3).
        map(x -> x + " - the length of the letters is three").
        forEach(x -> System.out.println(x));
```

Контрольные вопросы

- Что такое стрим в рамках Stream API?
- Каков жизненный цикл стрима?
- В чем отличие стрима от коллекции?
- Назовите две основные операции со стримами.

Промежуточные операции

Промежуточные операции

Промежуточные (нетерминальные) **операции** Stream API являются операциями, которые **преобразовывают** или **фильтруют** элементы в потоке.

При добавлении нетерминальной операции в стрим, получим новый стрим в качестве результата. Новый стрим будет представлять собой поток элементов, полученных из исходного стрима с применением нетерминальной операции.

Filter

Для фильтрации элементов в стриме применяется метод **filter**. Метод фильтра принимает **Predicate**, который вызывается для каждого элемента в стриме:

- если элемент **должен быть включен** в результирующий стрим, Predicate должен вернуть значение **true**.
- если элемент **не должен быть включен**, Predicate должен вернуть **false**.

Рассмотрим следующий пример:

Отфильтровать лист. Вывести в консоль город с названием длиной более 3 символов и с наличием буквы «M».

```
List<String> list = Arrays.asList("Moscow", "NY", "Tokyo");
list.stream()
        .filter(value -> value.length() >= 3)
        .filter(value -> value.contains("M"))
        .forEach(System.out::println); // Moscow
```

Map

Маппинг **map** задает функцию преобразования одного объекта в другой.

Например, в списке строк можно преобразовать элементы в нижний регистр, верхний регистр или в подстроку исходной строки, и т.д..

Рассмотрим следующий пример:

Преобразуем список городов в слова заглавными буквами и добавим строку.

```
List<String> list = Arrays.asList("Moscow", "NY", "Tokyo");
list.stream()
        .map(String::toUpperCase)
        .map(value -> value + " <3")
        .forEach(System.out::println);

// MOSCOW <3
// NY <3
// TOKYO <3
```

FlatMap

Метод **flatMap** отображает один элемент в виде нескольких элементов.

Идея состоит в том, что **flatMap** «сплющивает» каждый элемент из сложной структуры, состоящей из нескольких внутренних элементов, в «плоский» стрим, состоящий только из этих внутренних элементов.

Рассмотрим следующий пример:

Требуется вывести в консоль количество слов в четверостишие.

```
List<String> list = new ArrayList<>();
list.add("И долго буду тем любезен я народу");
list.add("Что чувства добрые я лирой пробуждал");
list.add("Что в мой жестокий век восславил я Свободу");
list.add("И милость к падшим призывал");

long count = list.stream()
       .flatMap(value -> Arrays.stream(value.split(" ")))
       .count();
System.out.println(count); // 26
```

Distinct

Метод **distinct** применяется для удаления дубликатов.

Метод **distinct** является нетерминальной операцией, которая возвращает новый стрим, который будет содержать только уникальные элементы из исходного стрима. Любые дубликаты будут исключены.

Рассмотрим следующий пример:

Требуется удалить дубликаты из листа.

```
List<String> list = Arrays.asList("one", "two", "three", "one", "two");
list = list.stream()
       .distinct()
       .collect(Collectors.toList());
System.out.println(list); // [one, two, three]
```

Limit

Метод **limit** применяется для ограничения количества элементов в стриме.

Метод **limit** возвращает новый стрим, который будет максимально содержать заданное в качестве аргумента количество элементов.

Рассмотрим следующий пример:

Требуется вывести в консоль ограниченное число элементов листа.

```
List<String> list = Arrays.asList("one", "two", "three", "one", "two");
list = list.stream()
       .limit(2)
       .collect(Collectors.toList());
System.out.println(list);  // [one, two]
```

Sorted

Метод **sorted** применяется для сортировки элементов источника стрима.

В зависимости от типа используемого **компаратора**, можно получить различный результат:

```
List<String> list = Arrays.asList("9", "A", "Z", "1", "B", "Y", "4", "a", "c");
```

naturalOrder – сортировка элементов в естественном порядке;

```
list.stream()
     .sorted(Comparator.naturalOrder())
     .forEach(System.out::print); // 149ABYZac
```

reverseOrder – сортировка элементов в обратном порядке;

```
list.stream()
     .sorted(Comparator.reverseOrder())
     .forEach(System.out::print); // caZYBA941
```

Контрольные вопросы

- Какие особенности у промежуточных операций?
- Допускается ли повторный вызов?
- Какая операция применяется для фильтрации?
- Для какой цели применяется оператор map?

Терминальные операции

Терминальные операции

Вызов терминальной операцию в стриме завершает цепочку экземпляров Stream из промежуточных операций и возвращает результат.

Match

Терминальные операции **match** применяются для проверки наличия совпадающего объекта в источнике стрима. В качестве аргумента используется предикат. **Match** запускает внутреннюю итерацию стрима и применяет параметр предиката к каждому элементу.

- Метод **anyMatch** возвращает true, если предикат возвращает true для любого из элементов.
- Метод **allMatch** возвращает true, если предикат возвращает true для всех элементов.
- Метод **noneMath** возвращает true, если предикат возвращает false для всех элементов.

Рассмотрим следующий пример:

Выведем в консоль проверку элементов листа на наличие элемента, начинающего на «o».

```
List<String> list = Arrays.asList("one", "two", "three");

boolean anyMatch = list.stream()
       .anyMatch(value -> value.startsWith("o"));
System.out.println(anyMatch); // true
```

Collect

Метод **collect** является терминальной операцией, которая запускает внутреннюю итерацию элементов и собирает элементы стрима в коллекцию.

Рассмотрим следующий пример:

Преобразуем элементы листа и занесем их в отдельный лист.

```
List<String> list = Arrays.asList("one", "two", "three");

List<String> uppercaseList = list.stream()
       .map(String::toUpperCase)
       .collect(Collectors.toList());
System.out.println(uppercaseList);
```

Count

Метод **count** является терминальной операцией, которая запускает внутреннюю итерацию элементов и определяет количество элементов.

Рассмотрим следующий пример:

Определим количество слов в четверостишье.

```
List<String> list = new ArrayList<>();
list.add("И долго буду тем любезен я народу");
list.add("Что чувства добрые я лирой пробуждал");
list.add("Что в мой жестокий век восславил я Свободу");
list.add("И милость к падшим призывал");

long count = list.stream()
       .flatMap(value -> Arrays.stream(value.split(" ")))
       .count();
System.out.println(count); // 26
```

Find

Метод **find** является терминальной операцией, которая производит поиск элементов в стриме.

- метод **findAny** может найти один элемент из стрима. Найденный элемент может быть из любой точки стрима.
- метод **findFirst** вернет первый элемент, если таковой существует.

Рассмотрим следующий пример:

Выведем в консоль первый элемент листа строчек. Проверим, существует ли результат.

```
List<String> strings
       = Arrays.asList("Java Script", "Java 8", "Java 11", "Android",
"Spring");

Optional<String> result = strings.parallelStream()
       .filter(s -> s.contains( "Java"))
       .findFirst();

result.ifPresentOrElse(
       System.out::println,
       () -> System.out.println("There is no Java :(")
); // Java Script
```

Обратите внимание, что методы **find** возвращают тип **Optional**. Если стрим будет пустой, то метод не вернет никакого результата. С помощью метода **ifPresent** проверим был ли найден результат в **Optional**.

ForEach

Метод **forEach** является терминальной операцией, которая запускает внутреннюю итерацию элементов в стриме и применяет **Consumer** к каждому элементу в стриме.

Сам метод **forEach** возвращает **void**.

```
Stream<String> stream = stringList.stream();
stream.forEach(System.out::print);
```

Min & Max

Методы **min** и **max** это терминальные операции, которые возвращают наименьший и наибольший элемент стрима.

Определение наименьшего и наибольшего элемента происходит с помощью передачи определенной имплементации компаратора в методы min и max.

Рассмотрим следующий пример:

Найдем наименьший элемент листа целых чисел.

```
List<Integer> intList = Arrays.asList(1, 2, 5, 10);
Optional<Integer> min = intList.stream().min(Integer::compareTo);
if (min.isPresent()) {
    Integer minString = min.get();
    System.out.println(minString);
}
```

> Обратите внимание, что метод min и max возвращает тип Optional, который может содержать или не содержать результат. Если стрим пустой, метод get пробросит исключение NoSuchElementException.

Reduce

Метод **reduce** это терминальная операция, которая может свести все элементы в стриме к одному элементу.

Reduce возвращает тип Optional. Этот необязательный параметр содержит значение (если оно есть), возвращаемое лямбда-выражением, переданным методу reduce. Получить значение можно с помощью get.

Рассмотрим следующий пример:

Объединим элементы листа в одну строчку через «+».

```
List<String> stringList = new ArrayList<String>();
stringList.add("one");
stringList.add("two");
stringList.add("three");

Optional<String> reduced = stringList.stream()
     .reduce((value, combinedValue) -> combinedValue + " + " + value);
reduced.ifPresent(System.out::println); // one + three + two
```

ToArray

Метод **toArray** это терминальная операция, которая запускает внутреннюю итерацию элементов в стриме и возвращает массив Object, содержащий все элементы.

Рассмотрим следующий пример:

Преобразуем лист строчек в массив объектов.

```
List<String> stringList = new ArrayList<String>();
stringList.add("One");
stringList.add("Two");
stringList.add("Thee");

Stream<String> stream = stringList.stream();
Object[] objects = stream.toArray();
```

Контрольные вопросы

- Какие особенности у терминальных операций?
- Допускается ли повторный вызов?
- Какая операция применяется для поиска?
- Для какой цели применяется оператор forEach?

Итоги

Итоги

- Стримы предназначены для упрощения работы с наборами данных, в частности, упростить операции фильтрации, сортировки, модификации и другие манипуляции с данными.
- Жизненный цикл стрима состоит из трех стадий: создание, применение промежуточных операций, терминальная операция.
- Промежуточные операции возвращают трансформированный стрим, к которому также можно применить ряд промежуточных операций.
- Терминальные операции возвращают конкретный результат и являются завершающими.
- Распараллеливание стримов задействует несколько ядер процессора и тем самым может повысить производительность и ускорить вычисления.

Домашнее задание

Давайте посмотрим ваше домашнее задание.

- Вопросы по домашней работе задавайте **в чате** мессенджера.
- Задачи можно сдавать **по частям**.
- Зачёт по домашней работе проставляется после того, как **приняты все задачи**.

нетология

Задавайте вопросы и пишите отзыв о лекции!

Григорий Вахмистров