Java 8: ламбда функции и Stream API

Любомир Чорбаджиев, Емил Гоцев

Технологично училище "Електронни системи"
Технически университет, София

6 април 2017 г.

Съдържание

Пример: филтриране на списък от хора

- Дефиниция на класа Person:

```
1  public class Person {
2
3    private String names;
4
5    private Gender gender;
6
7    private int age;
8
9      // all args constructor & getters...
10 }
```

- Как ще изглежда функция, която филтрира в един списък хората на възраст под 25 години?

Пример: филтриране на списък от хора

- Решение:

```
1  public static List<Person> filterByAgeLessThan(
2      List<Person> people, int age) {
3    List<Person> result = new ArrayList<>();
4    for (Person person : people) {
5      if (person.getAge() < age) {
6        result.add(person);
7      }
8    }
9    return result;
10 }
```

- А как ще изглежда функция, която филтрира хората на определена възраст и от определен пол?

Пример: филтриране на списък от хора

- Функция филтрираща по още един критерий:

```
1  public static List<Person> filterByAgeAndGender(
2      List<Person> people, int age, Gender gender) {
3    List<Person> result = new ArrayList<>();
4    for (Person person : people) {
5      if (person.getAge() == age
6          && person.getGender() == gender) {
7        result.add(person);
8      }
9    }
10   return result;
11 }
```

- Какви са разликите с предишната функция? Как може еднаквите части да се изнесат на едно място?

Решение преди Java 8

- Добавя се параметър на филтриращата функция от абстрактен тип, в чиито абстрактни методи да се дефинира при извикване поведението, което искаме да се променя динамично

```
1 public interface Filter {
2   boolean matches(Person person);
3 }
```

```
1 public static List<Person> filter(List<Person> people,
2     Filter filter) {
3   List<Person> result = new ArrayList<>();
4   for (Person person : people) {
5     if (filter.matches(person)) {
6       result.add(person);
7     }
8   }
9   return result;
10 }
```

Решение преди Java 8

- Извикването на функцията обикновено става с инстанцирането на анонимен клас:

```
1  List<Person> people = Arrays.asList(
2      new Person("Ivan", Gender.MALE, 22),
3      new Person("Ivanka", Gender.FEMALE, 34));
4
5  List<Person> result = filter(people, new Filter() {
6
7    public boolean matches(Person person) {
8      return person.getAge() > 12 && person.getAge() < 65
9          && person.getGender() == Gender.FEMALE;
10   }
11 }); // ще съдържа само инстанцията с име "Ivanka"
```

- Но този синтаксис е неудобен, дълъг и по-трудно четим...

Java 8 синтаксис

- Затова в Java 8 той е опростен чрез т. нар. ламбда функции - функции подадени като аргумент на други функции.

```
List<Person> result = filter(people,
    person -> person.getAge() > 12
        && person.getAge() < 65
        && person.getGender() == Gender.FEMALE)
}); // ще съдържа само инстанцията с име "Ivanka"
```

- Декларацията на функцията `filter` от примера остава същата
- Резултатът от операцията в ламбда функцията се връща като резултат при извикването на `Filter::matches` в имплементацията на `filter`

Функционални интерфейси

- Интерфейсът `Filter` наричаме функционален
- Функционален интерфейс е този, в който има точно един абстрактен метод
- За да може да се прилага ламбда синтаксисът, типът на аргумента, на който се подава ламбда функцията, трябва да бъде "функционален интерфейс"
- При абстрактни класове (дори и само с един абстрактен метод) или интерфейси с повече от един абстрактен метод, този синтаксис не може да се използва (в тези случаи могат да се използват анонимни класове)

Функционални интерфейси

- В стандартната библиотека в пакета `java.util.function` има дефинирани множество шаблонни функционални интерфейси, които се използват както в самата библиотеката, така и могат да се използват от всеки програмист.
- `Function` - дефинира метод, който приема един шаблонен аргумент и връща шаблонен резултат
- `Consumer` - метод с един шаблонен аргумент и връщащ **void**
- `Supplier` - метод без аргументи и връщащ шаблонен резултат
- `Predicate` - метод с един шаблонен аргумент и връщащ резултат от тип **boolean**
- Съществуват вариации на тези функционални методи, които дефинират същите методи, но с два аргумента и се казват съответно `BiFunction`, `BiConsumer` и т.н.

Пример: сортиране на списък с `Comparator`

- В `java.util.List` има метод `sort` с единствен аргумент от тип `Comparator`, който е функционален интерфейс

```
1 public interface Comparator<T> {
2   int compare(T o1, T o2);
3 }
```

```
1 List<Person> people = Arrays.asList(
2     new Person("Ivan", Gender.MALE, 22),
3     new Person("Ivanka", Gender.FEMALE, 34),
4     new Person("Peter", Gender.FEMALE, 11));
5 people.sort((p1, p2) -> p1.getAge() - p2.getAge());
```

Ламбда функция на много редове

- В предишните примери ламбда функциите се състояха от само един израз
- Понякога обаче е необходимо да се извършат няколко операции в тях преди да се върне резултат
- Тогава тялото на ламбда функцията е оградено с {} и задължително има **return** израз, когато се очаква функцията да върне резултат
- Следващият пример е еквивалентен на предишния:

```
1 people.sort((p1, p2) -> {
2   int delta = p1.getAge() - p2.getAge();
3   return delta;
4 });
```

Ламбда функция на много редове

- Методът `forEach` в `java.util.List` може да се използва за обхождане на всички елементи
- Той приема като аргумент `Consumer` и затова не очаква **return** израз, дори и когато ламбда функцията е на много редове

```
1 people.forEach(person -> {
2   if (person.getAge() > 65) {
3     System.out.printf("%s, 65+\n", person.getNames());
4   } else {
5     System.out.printf("%s, %d\n", person.getNames(),
6         person.getAge());
7   }
8 });
```

Необходимостта от Stream API

- Как ще изглежда код, който намира списък с имената на всички жени под 25 години, подреден по възрастта им?

```
1  List<Person> womenUnder25 = new ArrayList<>();
2  for (Person person : people) {
3    if (person.getGender() == Gender.FEMALE
4        && person.getAge() < 25) {
5      womenUnder25.add(person);
6    }
7  }
8  Collections.sort(womenUnder25,
9      (p1, p2) -> p1.getAge() - p2.getAge());
10 List<String> womenUnder25Names = new ArrayList<>();
11 for (Person person : people) {
12   womenUnder25Names.add(person.getNames());
13 }
```

Необходимостта от Stream API

- Решенията на различни задачи за филтриране, трансформиране на списък с обекти, редуцирането му до единствена стойност и други са с много подобен код
- Установяването на целите му, от човек, който не го е виждал, няма да е лесно и бързо
- Не можем ли просто да декларираме какво искаме да се случи и итерациите по списъка да бъдат скрити като имплементационен детайл, който не ни интересува?

Необходимостта от Stream API

- Решение, използвайки добавения в Java 8 Stream API:

```
1 List<String> womenUnder25Names = people.stream()
2   .filter(person -> person.getGender() == Gender.FEMALE)
3   .filter(person -> person.getAge() < 25)
4   .sorted((p1, p2) -> p1.getAge() - p2.getAge())
5   .map(person -> person.getNames())
6   .collect(Collectors.toList());
```

Какво е Stream? Stream vs Collection

- **Поредица от елементи** - подобно на колекцията, потокът предоставя интерфейс към поредица от елементи от определен тип, но за разлика от нея не се фокусира върху запазването и достъпа им от паметта, а върху изчисленията, които трябва се извършат с тях
- **Базиран на източник на данни** - често това е колекция, масив или I/O ресурс, като този източник може да бъде безкраен
- **Композиращ операции върху данните** - поддържат се различни операции като `filter`, `map`, `reduce`, `sort` и други, които могат да се композират в различен ред взависимост от целта

Крайни и междинни операции

```
1 people.stream()
2     .filter(person -> person.getAge() < 25)
3     .map(person -> person.getNames())
4     .collect(Collectors.toList());
```

- Извиквайки новият за `List` метод `stream` създаваме от списъка обект от тип `Stream`
- Операциите `filter` и `map` са **междинни**
- Всяка от междинните операции връща нов обект от тип `Stream` и това позволява лесното им композиране една след друга
- Когато композираме междинни операции, никоя от тях не се изпълнява, докато не извикаме накрая **терминална** операция
- Операцията `collect` е **крайна** и трансформира потока в списък
- Крайна операция върху поток може да се изпълни само веднъж и нейното извикване кара всички междинни операции да се изпълнят

Крайни и междинни операции (примери)

- Следният код няма да доведе до изпълнение на операциите `filter` и `map` върху данните от списъка `people`, тъй като липсва крайна операция

```
1 Stream<String> stream = people.stream()
2     .filter(person -> person.getAge() < 25)
3     .map(person -> person.getNames());
```

Крайни и междинни операции (примери)

- Следният пример ще доведе до `java.lang.IllegalStateException`, защото върху един поток са изпълнени две крайни операции - `collect` и `forEach`

```
1 Stream<String> stream = people.stream()
2     .filter(person -> person.getAge() < 25)
3     .map(person -> person.getNames());
4 stream.collect(Collectors.toList());
5 stream.forEach(name -> System.out.println(name));
```

Междинни операции: `filter`

- `filter` оставя в потока само елементи, за които условието се изчислява до **true**

```
1 List<Person> youngPeople = people.stream()
2     .filter(person -> person.getAge() < 25)
3     .collect(Collectors.toList());
4 // в youngPeople ще останат само хора под 25 години
```

Междинни операции: map

- map трансформира всеки обект в един поток до друг обект посредством правилото подадено като ламбда функция
- Потокът, който се връща, е от тип Stream<T>, където T е типът на върнатия от ламбда функцията обект (в случая - String)

```
1 List<String> names = people.stream()
2     .map(person -> person.getNames())
3     .collect(Collectors.toList());
4 // в names ще останат само имената на хората от people
```

Междинни операции: `flatMap`

- За следващия пример, нека добавим в класа `Person` поле, което да съдържа списък с всички приятели на дадения човек

```
class Person {
  private List<Person> friends;
  ...
}
```

- `flatMap` се използва, за да трансформираме един поток от списъци от елементи в поток от елементи
- Задължително е ламбда функцията, подадена на `flatMap`, да връща поток

```
List<Person> friends = people.stream()
    .flatMap(person -> person.getFriends().stream())
    .collect(Collectors.toList());
// friends ще съдържа всички приятели на хората в people
```

Междинни операции: `distinct`

- `distinct` премахва повтарящите се елементи от потока

```
1 long uniqueCount = people.stream()
2     .map(person -> person.getNames().split("␣")[0])
3     .distinct()
4     .count();
5 // uniqueCount ще съдържа броят на уникалните първи имена
```

Междинни операции: sorted

- sorted има две версии - тази без аргументи очаква елементите в потока да имплементират интерфейса Comparable
- Другата версия приема като аргумент Comparator

```
1 List<String> names = people.stream()
2     .sorted((p1, p2) -> p1.getAge() - p2.getAge())
3     .map(person -> person.getNames())
4     .collect(Collectors.toList());
5 // подрежда хората по възраст и връща само имената им
```

Междинни операции: `limit` и `skip`

- `limit` лимитира елементите до подаденото число
- `skip` премахна първите n елемента от потока

```
1 List<Person> result = people.stream()
2     .skip(1)
3     .limit(5)
4     .collect(Collectors.toList());
5 // ще пропусне първия елемент и след това ще добави максимум
6 // следващите 5 елемента в result
```

Междинни операции: peek

- peek не променя потока, но позволява да се извършват някакви операции с елементите в него
- Това го прави удобен за “debug” цели

```
1 people.stream()
2     .map(person -> person.getNames())
3     .peek(names -> System.out.println(names))
4     .disctinct()
5     .count();
6 // ще изброи хората с уникални имена, като ще изпринтира
7 // имената на всички хора
```

Крайни операции: `count` и `forEach`

- `count` ще върне броя на елементите в потока

```
1 long count = people.stream()
2     .filter(person -> person.getAge() % 2 == 0)
3     .count();
```

- `forEach` ще изпълни подадената ламбда функция за всеки елемент в потока

```
1 people.stream()
2     .map(person -> person.getNames())
3     .forEach(name -> System.out.println(name));
```

Крайни операции: `reduce`

- `reduce` ще комбинира елементите чрез подадената ламбда функция, докато не остане само един
- Като първи аргумент се подава първоначалната стойност, от която да започне комбинирането на елементи

```
1 long ageSum = people.stream()
2     .map(person -> person.getAge())
3     .reduce(0, (l, r) -> l + r);
```

```
1 String namesSeparatedByComma = people.stream()
2     .map(person -> person.getNames())
3     .reduce("", (l, r) -> l + ",␣" + r);
```

- Забележка: `reduce` има и още две версии с различни аргументи

Крайни операции: `collect`

- `collect` приема като аргумент имплементация на интерфейса `java.util.stream.Collector`
- В класа `java.util.stream.Collectors` има дефинирани няколко статични метода, които връщат различни имплементации на `Collector` интерфейса
- Всеки програмист може и сам да дефинира своя имплементация, но това е извън обхвата на тази лекция

collect: трансформиране в колекция

- Collectors.toList() записва елементите от потока в списък

```
1 List<String> nameList = people.stream()
2     .map(person -> person.getNames())
3     .collect(Collectors.toList());
```

- Collectors.toSet() записва елементите от потока в множество

```
1 Set<String> nameSet = people.stream()
2     .map(person -> person.getNames())
3     .collect(Collectors.toSet());
```

- Collectors.toCollection(Supplier<C>) записва елементите в колекцията, която се създава от ламбда функцията

```
1 Queue<String> nameQueue = people.stream()
2     .map(person -> person.getNames())
3     .collect(Collectors.toCollection(() ->
4         new SynchronousQueue<>()));
```

collect: трансформиране в java.util.Map

- Чрез Collectors.toMap() можем да трансформираме елементите от потока в двойки ключ-стойност
- Първият аргумент е ламбда функция, която връща ключ за всеки обект
- Вторият аргумент е ламбда функция, която връща стойност за всеки обект
- Третият аргумент е опционален и се използва за разрешаване на конфликти (когато има повторение на ключове)

```
1 Map<String, Gender> genderByPerson = people.stream()
2     .collect(Collectors.toMap(
3         person -> person.getNames(),
4         person -> person.getGender(),
5         (g1, g2) -> g1));
```

collect: групиране на елементите

- Collectors.groupingBy() може да се използва за групиране на елементите по даден критерий

```
1 Map<Gender, List<Person>> byGender = people.stream()
2     .collect(Collectors.groupingBy(
3         person -> person.getGender()));
```

- Подобен на този колектор е и Collectors.partitioningBy(), който групира елементите в 2 групи - в зависимост от това дали изпълняват, или не дадено условие

```
1 Map<Boolean, List<Person>> byAgeLt25 = people.stream()
2     .collect(Collectors.partitioningBy(
3         person -> person.getAge() < 25));
```

`collect`: комбиниране на колектори

- Проблем: как може да се групират имената на хората по пол?
- За целта `Collectors.groupingBy()` и `Collectors.partitioningBy()` имат версии, които приемат следващ колектор като параметър
- За решение на проблема използваме колектора `Collectors.mapping()`, който приема като аргумент ламбда функция, която да вземе имената на всички хора, и колектор, който да трансформира потокът от имена

```
1 Map<Gender, List<String>> byGender = people.stream()
2     .collect(Collectors.groupingBy(
3         person -> person.getGender(),
4         Collectors.mapping(person -> person.getNames(),
5             Collectors.toList())));
```

Други методи от `Collectors`

- `collectingAndThen(Collector, Function)` - адаптира подадения колектор да изпълнява допълнителна операция за трансформиране на елементите
- `counting()` - връща колектор, който изброява елементите в потока
- `minBy(Comparator)` и `maxBy(Comparator)` - колекторът намира съответно най-малкия и най-големия елемент в потока, използвайки за сравнение подадения `Comparator`
- `summingInt(ToIntFunction)` - колекторът, използвайки подадената ламбда функция, трансформира елементите в **int** и ги събира; има аналогични колектори за **long** и **double**
- `averagingInt(ToIntFunction)` - аналогичен колектор, но смята средното аритметично; също има за **long** и **double**

Задачи за упражнение

- За всички задачи приемете, че имате списък с хора
- Трансформирайте списъка в стринг, в следния формат: “<names> (<age>), . . . ”
- Филтрирайте хората, които нямат приятели и пресметнете за останалите средната възраст на приятелите им; запишете резултата в масив с ключ имената на човека и стойност средната възраст на приятелите му
- Пресметнете средната възраст по пол на хората
- Пресметнете броят на хората от всеки пол
- Групирайте хората първо по пол, а след това по това дали са на повече или по-малко от 25 години
- Групирайте хората по възраст над/под 25 години, като върнете само множество от имената им (тип на резултата: `Map<Boolean, Set<String>>`)