Java. Углубленное изучение

Урок 4

Java Stream API

Функциональное программирование на Java.

Что такое Java Stream API?

Чтобы начать изучение Java Stream API, разберем основные определения. В языке Java есть понятие потоков, но ни классы **InputStream**, ни **Thread** не имеют ничего общего с новшеством Java 8 — **Stream API**. Когда классы **InputStream** просто создают потоки ввода/вывода, **Stream API** работает не с потоком в прямом смысле слова, а с цепочкой функций, вызываемых из самих себя. Он обеспечивает функциональное программирование в Java 8.

Как работают потоки?

Поток — это последовательность элементов и функций, которые поддерживают различные виды операций:

```
List<String> myList = Arrays.asList("aa","cc", "bb", "dd", "ee");
myList
    .stream()
    .filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
// CC
```

Операции с потоком могут относиться к терминальным или промежуточным. Все промежуточные операции возвращают поток, так что их можно объединять, не используя символы закрывания функции (точка с запятой). Терминальные операции возвращают либо результат, не относящийся к потоку, либо ничего (**void**). В приведенном листинге **filter**, **map** и **sorted** — промежуточные операции, так как они возвращают модифицированный после последней операции поток. А **forEach** — терминальная операция и может возвращать коллекцию. Полный список таких операций доступен в **JavaDoc** к Java 8. Большинство операций потока могут принимать лямбда-выражения. В функциональный интерфейс передается точное поведение по каждой операции.

Виды потоков

Чтобы не путаться в понятиях, будем называть потоком параллельное выполнение программы, а **Stream API** обозначать как стрим. Стримы можно создавать из различных источников данных, но вызывают их в большинстве случаев из коллекций **List** и **Set**. Они поддерживают новый метод **stream()**. Есть новый метод **parallelStream()**, способный работать на нескольких нитях. Поговорим о нем позже, сейчас наша задача — ориентироваться на последовательные потоки.

```
Arrays.asList("aa1", "aa2", "aa3")
    .stream()
    .findFirst()
    .ifPresent(System.out::println);
// Результат выполнения: aa1
```

Чтобы вызвать стрим из коллекции, надо вызвать у нее метод **stream()** или использовать **Stream.of()** для создания стрима из произвольного набора элементов:

```
Stream.of("cc1", "cc2", "cc3").findFirst().ifPresent(System.out::println);
```

Помимо основных стримов в Java 8 можно вызвать особый вид потока, основывающийся на типе объекта. Например, для работы с примитивными данными применяются **IntStream**, **DoubleStream**, **LongStream**.

С помощью стрима **IntStream** и его встроенной функции **range()** можно заменить обход цикла всего одной строкой кода:

```
IntStream.range(1, 5).forEach(System.out::println);
```

Все примитивные потоки работают по тому же алгоритму, что и обычные объектные, но для них выведены специализированные лямбда-выражения. Например, **Function** в объектном стриме будет выглядеть как **IntFunction** в примитивном потоке целых чисел. Также примитивные стримы поддерживают расширенные методы, такие как **sum()** для целочисленных стримов и другие:

```
Arrays.stream(new int[] {1, 2, 3})
    .map(n -> 2 * n + 1)
    .average()
    .ifPresent(System.out::println);
// Результатом выполнения будет 5.0
```

Преобразование из объектного стрима в примитивный и наоборот полезно для расширенных операций с элементами коллекций. Для конвертирования типа стрима были также добавлены специальные конвертеры наподобие **mapToInt()**. Для обратной конвертации в объектный стрим используется функция **mapToObj()**.

```
Stream.of("c1", "c2", "c3")
    .map(s -> s.substring(1))           // Получаем цифры
    .mapToInt(Integer::parseInt)        // Парсим из текста в цифру
    .max()                              // Выделяем максимальное число
    .ifPresent(System.out::println);    // Выводим на экран
                                        // Результатом выполнения будет 3
```

```
IntStream.range(1, 4)
    .mapToObj(i -> "c" + i)
    .forEach(System.out::println);
// Результат выполнения операции:
// c1
// c2
// c3
```

Порядок обработки

Узнаем, что происходит внутри операций. Рассмотрим фрагмент кода, в котором нет терминальной операции:

```
Stream.of("dd2", "aa2", "bb1", "bb3", "cc4")
```

```
    .filter(s -> {
        System.out.println("Фильтр: " + s);
        return true;
    });
```

Во время выполнения этого блока консоль будет оставаться пустой, так как все промежуточные операции будут выполняться, только если присутствует хотя бы одна терминальная. Расширим пример терминальной операцией **forEach()**, и в консоли начнет выводиться наш блок:

```
Stream.of("1", "2", "3", "4", "5")
    .filter(s -> {
        System.out.println("Фильтр: " + s);
        return true;
    })
    .forEach(s -> System.out.println("Печать с использованием forEach: " + s));
```

Что появится в консоли после выполнения блока:

```
Фильтр:  1
Печать с использованием forEach: 1
Фильтр: 2
Печать с использованием forEach: 2
Фильтр: 3
Печать с использованием forEach: 3
Фильтр: 4
Печать с использованием forEach: 4
Фильтр: 5
Печать с использованием forEach: 5
```

Порядок выполнения может удивить. На первый взгляд, все операции будут выполняться по горизонтали одна за другой по всем элементам потока. Но в нашем примере первая строка «**dd2**» полностью проходит фильтр **forEach**, потом обрабатывается вторая строка «**aa2**» и так далее.

Почему порядок работы имеет значение?

Следующий пример состоит из двух промежуточных операций: **map** и **filter**, — и выполнения терминала **forEach**. Посмотрим еще раз на порядок выполнения этих операций.

Рассмотрим следующий пример, который состоит из операций **map** и **filter** с закрывающим **forEach()**. Обратите внимание на порядок выполнения программы:

```
Stream.of("dd2", "aa2", "bb1", "bb3", "cc4")
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("A");
    })
```

```
    .forEach(s -> System.out.println("forEach: " + s));
// Результат выполнения
// map:     dd2
// filter:  DD2
// map:     aa2
// filter:  AA2
// forEach: AA2
// map:     bb1
// filter:  BB1
// map:     bb3
// filter:  BB3
// map:     cc
// filter:  CC
```

map и **filter** называются 5 раз для каждой строчки в коллекции, а **forEach** вызывается только один раз.

Но можно сократить количество выполнений, изменив порядок операций. Посмотрим, что произойдет, если **filter()** окажется в начале цепи:

```
Stream.of("dd2", "aa2", "bb1", "bb3", "cc4")
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("a");
    })
    .map(s -> {
        System.out.println("map: " + s);
        return s.toUpperCase();
    })
    .forEach(s -> System.out.println("forEach: " + s));
// filter:  dd2
// filter:  aa2
// map:     aa2
// forEach: AA2
// filter:  bb1
// filter:  bb3
// filter:  cc
```

Теперь **map** вызывается только один раз и выполняется быстрее для большого количества входных элементов. Это стоит иметь в виду при составлении комплексного метода цепи.
Есть правило при работе со стримами в Java 8: их нельзя использовать дважды. Как только вызывается терминальная операция, поток надо закрыть.

```
Stream<String> stream =
    Stream.of("a1", "b2", "a3", "c4", "d5")
        .filter(s -> s.startsWith("d"));
stream.anyMatch(s -> true);     // Пройдет без проблем
stream.noneMatch(s -> true);    // Выдаст исключение
```

Вызов операции **noneMatch()** после выполнения терминальной операции — в данном случае **anyMatch()** — в одном потоке вызовет исключение:

```
Exception in thread "main" java.lang.IllegalStateException: stream has already
been operated upon or closed
      at java.util.stream.AbstractPipeline.evaluate(AbstractPipeline.java:229)
      at
java.util.stream.ReferencePipeline.noneMatch(ReferencePipeline.java:459)
```

Чтобы избежать этого, надо создать новую цепь для каждой терминальной операции.

Продвинутые операции

Познакомимся с более сложными операциями: **collect**, **flatMap** и **reduce**.

Для работы с ними напишем следующий код:

```
class Human {
    String name;
    int age;
    Human(String name, int age) {
        this.name = name;
        this.age = age;
    }
    @Override
    public String toString() {
        return name;
    }
}
List<Human> humans =
    Arrays.asList(
        new Human("Andrew", 20),
        new Human("Roman", 23),
        new Human("Tatiana", 23),
        new Human("Sasha", 12));
```

Обратите внимание на короткую инициализацию списка человек — в одной строке.

Операция Collect

Эта операция интересна, поскольку ее главная задача — превращение элементов потока в список или **Set**. Опция принимает в себя **Collectors**, выполняющие роль правила, по которому будут собраны элементы из стрима.

```
List<Human> filtered =
    persons
        .stream()
        .filter(p -> p.name.startsWith("A"))
        .collect(Collectors.toList());
 System.out.println(filtered);    // [Andrew]
```

Collectors — универсальный инструмент для построения коллекций. Позволяют создавать подгруппы объектов из стрима для общей цели. Например, чтобы определить средний возраст людей, представленных в стриме:

```
Double averageyears = persons
    .stream()
    .collect(Collectors.averagingInt(p -> p.age));
System.out.println(averageyears);    // 19.5
```

Можно выбрать людей по определенному признаку и вывести строкой:

```
String phrase = persons
    .stream()
    .filter(p -> p.age >= 18)
    .map(p -> p.name)
    .collect(Collectors.joining(" и ", "В Германии ", " совершеннолетние."));
System.out.println(phrase);
// В Германии Andrew и Roman и Tatiana совершеннолетние.
```

Коллектор **join** принимает разделитель, дополнительный префикс и суффикс.

Reduce

Эта операция выполняет роль сумматора по всем элементам стрима. Всего в данный момент поддерживается три частных случая такой операции, но мы рассмотрим один. Определим «старший» объект:

```
persons
    .stream()
    .reduce((p1, p2) -> p1.age > p2.age ? p1 : p2)
    .ifPresent(System.out::println);    // Ira
```

Reduce-метод принимает функцию аккумулятора **BinaryOperator**. На самом деле, это **BiFunction** — когда оба операнда имеют один и тот же тип (в этом случае — **Person**). **BiFunctions** похожи на **Function**, но принимают два аргумента. Пример функции сравнивает людей по возрасту и возвращает самого старшего.

Параллельные потоки

Потоки могут выполняться параллельно. Такие меры требуются, чтобы увеличить производительность при огромных количествах входных элементов. Они используют общий **ForkJoinPool**, доступный через статический метод **ForkJoinPool.commonPool()**. Размер основного пула — не более пяти нитей, так как количество ядер процессора — величина ограниченная.

```
ForkJoinPool commonPool = ForkJoinPool.commonPool();
System.out.println(commonPool.getParallelism());    // 3
```

На обыкновенном компьютере общий пул выставляется в значение 3 по умолчанию. Эти параметры можно изменять, как и любые другие, как описано ниже:

```
-Djava.util.concurrent.ForkJoinPool.common.parallelism=5
```

Коллекции поддерживают метод **parallelStream()**, чтобы создать параллельный поток элементов. Можно вызвать промежуточный метод **parallel()**, чтобы преобразовать последовательный поток в параллельной копии.

Чтобы занизить поведение параллельного выполнения параллельного потока, печатаем информацию о текущем потоке в **Sout**:

```
Arrays.asList("a1", "a2", "b1", "c2", "c1")
    .parallelStream()
    .filter(s -> {
        System.out.format("filter: %s [%s]\n",
            s, Thread.currentThread().getName());
        return true;
    })
    .map(s -> {
        System.out.format("map: %s [%s]\n",
            s, Thread.currentThread().getName());
        return s.toUpperCase();
    })
    .forEach(s -> System.out.format("forEach: %s [%s]\n",
        s, Thread.currentThread().getName()));
```

После дебага лучше понимаем, какие потоки на самом деле используется для выполнения операций потока:

```
filter:  b1 [main]
filter:  a2 [ForkJoinPool.commonPool-worker-1]
map:  a2 [ForkJoinPool.commonPool-worker-1]
filter:  c2 [ForkJoinPool.commonPool-worker-3]
map:  c2 [ForkJoinPool.commonPool-worker-3]
filter:  c1 [ForkJoinPool.commonPool-worker-2]
map:  c1 [ForkJoinPool.commonPool-worker-2]
forEach: C2 [ForkJoinPool.commonPool-worker-3]
forEach: A2 [ForkJoinPool.commonPool-worker-1]
map:  b1 [main]
forEach: B1 [main]
filter:  a1 [ForkJoinPool.commonPool-worker-3]
map:  a1 [ForkJoinPool.commonPool-worker-3]
forEach: A1 [ForkJoinPool.commonPool-worker-3]
forEach: C1 [ForkJoinPool.commonPool-worker-2]
```

Параллельный поток использует все доступные темы из общей **ForkJoinPool** для выполнения операций потока. Вывод может отличаться при последовательном запуске, потому что поведение, которое использует конкретный поток, не является детерминированным.

Расширим пример с помощью дополнительной операции потока — **sort**:

```
Arrays.asList("a1", "a2", "b1", "c2", "c1")
    .parallelStream()
    .filter(s -> {
        System.out.format("filter: %s [%s]\n",
            s, Thread.currentThread().getName());
        return true;
    })
    .map(s -> {
        System.out.format("map: %s [%s]\n",
            s, Thread.currentThread().getName());
        return s.toUpperCase();
    })
    .sorted((s1, s2) -> {
        System.out.format("sort: %s <> %s [%s]\n",
            s1, s2, Thread.currentThread().getName());
        return s1.compareTo(s2);
    })
    .forEach(s -> System.out.format("forEach: %s [%s]\n",
        s, Thread.currentThread().getName()));
```

Результат может на первый взгляд показаться странным:

```
filter:  c2 [ForkJoinPool.commonPool-worker-3]
filter:  c1 [ForkJoinPool.commonPool-worker-2]
map:  c1 [ForkJoinPool.commonPool-worker-2]
filter:  a2 [ForkJoinPool.commonPool-worker-1]
map:  a2 [ForkJoinPool.commonPool-worker-1]
filter:  b1 [main]
map:  b1 [main]
filter:  a1 [ForkJoinPool.commonPool-worker-2]
map:  a1 [ForkJoinPool.commonPool-worker-2]
map:  c2 [ForkJoinPool.commonPool-worker-3]
sort:    A2 <> A1 [main]
sort:    B1 <> A2 [main]
sort:    C2 <> B1 [main]
sort:    C1 <> C2 [main]
sort:    C1 <> B1 [main]
sort:    C1 <> C2 [main]
forEach: A1 [ForkJoinPool.commonPool-worker-1]
forEach: C2 [ForkJoinPool.commonPool-worker-3]
forEach: B1 [main]
forEach: A2 [ForkJoinPool.commonPool-worker-2]
forEach: C1 [ForkJoinPool.commonPool-worker-1]
```

Кажется, что **sort** выполняется последовательно только основной нитью. Но на самом деле на параллельном потоке **sort** использует под капотом новый метод Java 8 — **Arrays.parallelSort()**. Отладочный вывод относится только к исполнению переданного лямбда-выражения. **Sort**-компаратор выполнен только на главном потоке, но фактически алгоритм сортировки выполняется параллельно.

Вернемся к **reduce** (например, из последней секции). Мы уже выяснили, что функция-комбайнер вызывается только параллельно, но не в последовательных потоках. Посмотрим, какие потоки фактически участвуют:

```
List<Person> persons =
                Arrays.asList(
                        new Person("Andrew", 20),
                        new Person("Igor", 23),
                        new Person("Ira", 23),
                        new Person("Vitia", 12));
        persons
                .parallelStream()
                .reduce(0,
                        (sum, p) -> {
                            System.out.format("accumulator: sum=%s; person=%s
[%s]\n",
                                    sum, p, Thread.currentThread().getName());
                            return sum += p.age;
                        },
                        (sum1, sum2) -> {
                            System.out.format("combiner: sum1=%s; sum2=%s
[%s]\n",
                                    sum1, sum2,
```

```
Thread.currentThread().getName());
                            return sum1 + sum2;
                        });
```

Вывод в консоль показывает, что оба *аккумулятора* и *комбайнера* функции выполняются параллельно на всех доступных потоках:

```
accumulator: sum=0; person=Ira [main]
accumulator: sum=0; person=Andrew [ForkJoinPool.commonPool-worker-2]
accumulator: sum=0; person=Vitia [ForkJoinPool.commonPool-worker-3]
accumulator: sum=0; person=Igor [ForkJoinPool.commonPool-worker-1]
combiner: sum1=23; sum2=12 [ForkJoinPool.commonPool-worker-3]
combiner: sum1=20; sum2=23 [ForkJoinPool.commonPool-worker-1]
combiner: sum1=43; sum2=35 [ForkJoinPool.commonPool-worker-1]
```

Констатируем, что параллельное выполнение может дать хороший прирост производительности в потоках с большим количеством входных элементов. Но некоторые параллельные операции потока **reduce** и **collect** требуют дополнительных расчетов (комбинированные операции), которые не нужны при последовательном выполнении.

Домашнее задание

1. Сдать проект в текущем состоянии на Code Review.

Дополнительные материалы

1. Многопоточность в Java;
2. Многопоточное программирование в Java 8;
3. Java Streams.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Шпаргалка Java-программиста;
2. Stream API;
3. Java 8 Stream Tutorial - Benjamin Winterberg.