Advanced Java 8 Stream Patterns: FlatMap, Streams of Numbers

José Paumard

@JosePaumard | blog.paumard.org

Agenda

Concatenating Streams

State of a Stream

Stream of numbers

Concatenating Streams

Concat, FlatMap

Use Cases

- Suppose we have two text files : text1.txt and text2.txt
- We can open a stream on the lines of a text file

```
Path path = Paths.get("files/text1.txt");
try (Stream<String> s1 = Files.lines(path)) { // Stream is autocloseable

   // handle the file line by line

} catch (Exception e) {
   // handle the exception
}
```

Use Cases

- Suppose we have two text files : text1.txt and text2.txt
- We can open a stream on the lines of a text file
- And then concatenate the two streams

```
Stream<String> s1 = Files.lines(path1));
Stream<String> s2 = Files.lines(path2));

Stream<String> s10 = Stream.concat(s1, s2); // only two
Stream<String> s11 = Stream.concat(Stream.concat(s1, s2), s3);
```

- Not that great… risk of a StackOverflow exception

Stream.concat

- The documentation mentions that

> *the elements of the first stream*
> *are followed by all the elements of the second stream*

- So the order of the elements is preserved, which has a cost
- If it is not needed, then we should use the other pattern

Streams of Streams

- There is another method to merge streams

```
? s = Stream.of(s1, s2, s3); // varargs
```

Stream<String>

Streams of Streams

- There is another method to merge streams

```
Stream<Stream<String>> s = Stream.of(s1, s2, s3); // varargs
```

- This is not quite what we want!
- But we have the flatMap() call

Method `flatMap()`

- The flatMap operation is a special operation that takes a function
  - So it is just like a regular mapping…
- But this function has to return a stream
  - Special case: if it can be an identity function

```
Function<Stream<String>, Stream<String>> idFlatMapper = stream -> stream ;
```

Method flatMap()

- Of course a flat mapper can be used as a regular mapper
- But if passed to the flatMap() method, then…
- The resulting stream will be « flattened »

```
Stream<Stream<String>> streamOfStreams = Stream.of(s1, s2, s3); // varargs
Stream<String> stream = s.flatMap(idFlatMapper);
```

```
Stream<String> stream = Stream.of(s1, s2, s3)
                             .flatMap(Function.identity());
```

Other `flatMap()` Use Cases

- We could merge the lines of two files into one stream
- Could we split those lines into words, and merge them into one stream of words?
- Let us write a function that splits a line into words

```
Function<String, Stream<String>> splitIntoWords =
    line -> Stream.of(line.split(" "));
```

Other `flatMap()` Use Cases

- We could merge the lines of two files into one stream
- Could we split those lines into words, and merge them into one stream of words?
- Let us write a function that splits a line into words

```
Function<String, Stream<String>> splitIntoWords =
    line -> Pattern.compile(" ")
```

pattern

Other `flatMap()` Use Cases

- We could merge the lines of two files into one stream
- Could we split those lines into words, and merge them into one stream of words?
- Let us write a function that splits a line into words

```
Function<String, Stream<String>> splitIntoWords =
    line -> Pattern.compile(" ").splitAsStream(line);
```

Other `flatMap()` Use Cases

- We could merge the lines of two files into one stream
- Could we split those lines into words, and merge them into one stream of words?
- We can now use this function in a flat mapper

```
Stream.of(s1, s2, s3)
    .flatMap(Function.identity())
    .flatMap(splitIntoWords);
```

Other `flatMap()` Use Cases

- We could merge the lines of two files into one stream
- Could we split those lines into words, and merge them into one stream of words?
- We can now use this function in a flat mapper

```
Stream<String> words =
   Stream.of(s1, s2, s3)                  // stream of streams of lines
      .flatMap(Function.identity())       // stream of lines
      .flatMap(splitIntoWords);           // stream of words
```

Other `flatMap()` Use Cases

- We could merge the lines of two files into one stream
- Could we split those lines into words, and merge them into one stream of words?
- We can now use this function in a flat mapper

```
Stream.of(s1, s2, s3)                  // stream of streams of lines
    .flatMap(Function.identity())      // stream of lines
    .flatMap(splitIntoWords)           // stream of words
    .collect(Collectors.toSet());
```

Other `flatMap()` Use Cases

- We could merge the lines of two files into one stream
- Could we split those lines into words, and merge them into one stream of words?
- We can now use this function in a flat mapper

```
Set<String> words =
    Stream.of(s1, s2, s3)                  // stream of streams of lines
        .flatMap(Function.identity())      // stream of lines
        .flatMap(splitIntoWords)           // stream of words
        .collect(Collectors.toSet());
```

State of a Stream

Sorted, distinct

A Stream Has a State

- The implementation of the stream() method for the ArrayList class is the following

```
default Stream<E> stream() {
    return StreamSupport.stream(spliterator(), false);
}
```

- The spliterator() method is redefined in the ArrayList class

```
public Spliterator<E> spliterator() {
    return new ArrayListSpliterator<>(this, 0, -1, 0);
}
```

A Stream Has a State

- The Spliterator interface describes how to access the data of the source
  - This is the interface we want to implement if we need to consume data from a custom source
- Let us have a look at the characteristics() method of this implementation

```
public int characteristics() {
   return Spliterator.ORDERED | Spliterator.SIZED | Spliterator.SUBSIZED;
}
```

- Ordered, sized and subsized are part of a set of bits that describes a stream

A Stream Has a State

- The characteristics() method for ArrayList

```
public int characteristics() {
    return Spliterator.ORDERED | Spliterator.SIZED | Spliterator.SUBSIZED;
}
```

- The characteristics() method for HashSet

```
public int characteristics() {
    return (fence < 0 || est == map.size ? Spliterator.SIZED : 0) |
            Spliterator.DISTINCT;
}
```

A Stream Has a State

Characteristic	
ORDERED	The order matters
DISTINCT	No duplication
SORTED	Sorted
SIZED	The size is known
NONNULL	No null values
IMMUTABLE	Immutable
CONCURRENT	Parallelism is possible
SUBSIZED	The size is known

A Stream Has a State

- Some methods will change the value of those bits
- Either implicitly (map, filter)
- Either explicitly (sorted, distinct)

A Stream Has a State

Method	Set to 0	Set to 1
filter()	SIZED	-
map()	DISTINCT, SORTED	-
flatMap()	DISTINCT, SORTED, SIZED	-
sorted()	-	SORTED, ORDERED
distinct()	-	DISTINCT
limit()	SIZED	-
peek()	-	-
unordered()	ORDERED	-

Example

- It is all about performance!

```
people.stream()
      .distinct() // returns a stream = intermediate method
      .sorted()   // the same as distinct()
      .forEach(System.out::println);
```

Example

- It is all about performance!

```
// HashSet
HashSet<Person> people = ... ;
people.stream()
      .distinct() // no processing is triggered
      .sorted()   // quicksort is triggered
      .forEach(System.out::println);
```

Example

- It is all about performance!

```
// SortedSet
SortedSet<Person> people = ... ;
people.stream()
      .distinct() // no processing is triggered
      .sorted()   // no quicksort is triggered
      .forEach(System.out::println);
```

Remark on the `sorted()` Method

- The `sorted()` method can be called on a stream of `Comparable` objects
  - But it is not verified at compile time!
- It can also take a `Comparator` as a parameter, that will be used to compare the objects of the stream

Live Coding

How to merge large amount of text and cut it into words using `flatMap()`

Live Coding Summary

- We saw how to merge streams together using the Stream.of and flatMap patterns
- We saw how to efficiently split streams using the flatMap pattern

Streams of Numbers

IntStream, LongStream, DoubleStream

What Is a Stream of Numbers?

- Let us take another example, we would like to compute the average of the ages of our list of people

```
// average of the ages of the people from our list
List<Person> people = ... ;
people.stream()
      .map(person -> person.getAge())
      .filter(age -> age > 20)
      .average();
```

What Is a Stream of Numbers?

- Let us take another example: we would like to compute the average of the ages our list of people

```
// average of the ages of the people from our list
List<Person> people = ... ;
people.stream()
      .map(person -> person.getAge())
      .filter(age -> age > 20)
      .average();
```

- Unfortunately, the `average()` method does not exist on `Stream<T>`

What Is a Stream of Numbers?

- Let us take another example: we would like to compute the average of the ages our list of people

```
// average of the ages of the people from our list
List<Person> people = ... ;
people.stream()
      .map(person -> person.getAge())
      .filter(age -> age > 20)
      .average();
```

- But it does exist on `IntStream`…

What Is a Stream of Numbers?

- Let us take another example: we would like to compute the average of the ages our list of people

```
// average of the ages of the people from our list
List<Person> people = ... ;
people.stream()                                  // average
      .map(person -> person.getAge())            // Stream<Integer>
      .filter(age -> age > 20)
      .average();
```

- How could one convert a `Stream<Integer>` to an `IntStream`?

What Is a Stream of Numbers?

- Let us take another example: we would like to compute the average of the ages our list of people

```
// average of the ages of the people from our list
List<Person> people = ... ;
people.stream()                                  // average
      .map(person -> person.getAge())            // Stream<Integer>
      .filter(age -> age > 20)
      .mapToInt(i -> i)                          // IntStream
      .average();
```

- How could one convert a `Stream<Integer>` to an `IntStream`?

What Is a Stream of Numbers?

- Let us take another example: we would like to compute the average of the ages our list of people

```
// average of the ages of the people from our list
List<Person> people = ... ;
people.stream()                                  // average
      .mapToInt(person -> person.getAge()) // IntStream
      .filter(age -> age > 20)
      // .mapToInt(i -> i)
      .average();
```

- How could one convert a `Stream<Integer>` to an `IntStream`?

What Is a Stream of Numbers?

- Streams of numbers are there to avoid the cost of boxing / unboxing
- Three types: `IntStream`, `LongStream` and `DoubleStream`
- The patterns to build them are simple:

```
// build from a varargs
LongStream streamOfLongs = LongStream.of(1L, 2L, 3L);
```

```
// convert from a Stream<Integer>
IntStream streamOfInts = people.stream().mapToInt(Person::getAge);
```

What Is a Stream of Numbers?

- One can also « box » a Stream of numbers:

```
// box a Stream if needed
Stream<Long> boxedStream = LongStream.of(1L, 2L, 3L).boxed();
```

```
// box a Stream if needed
Stream<Long> boxedStream = LongStream.of(1L, 2L, 3L).mapToObj(l -> l);
```

Methods on Streams of Numbers

- Stream of numbers have special methods, not on `Stream<T>`

```
// on IntStream
int sum = intStream.sum();

OptionalInt min = intStream.min();
OptionalInt max = intStream.max();

OptionalDouble average = intStream.average();

IntSummaryStatistics stats = intStream.summaryStatistics();
```

- Summary statistics compute *sum*, *min*, *max*, *count* and *average* in one pass

Live Coding

How to use streams of `int` on the Scrabble example

Live Coding Summary

- We saw how to use `IntStream` to compute the score of a word at Scrabble
- We used it on the Shakespeare file
- Use of the summary statistics pattern

- We are not done with this example!

Summary

- Advanced Stream patterns: flatMap, concatenation
- The state of a stream, how it is used to improve perfomance
- Specialized streams: streams of numbers, once again for perfomance reasons