Collecting Data in Complex Containers Using Collectors

José Paumard

@JosePaumard | blog.paumard.org

Agenda

What is a Collector

The Collectors class

Collectors in action

What is a Collector

Reduction in a container

About the Reduction Step

- The reductions we saw were aggregations: sum, max, average, etc…
- A collector is a special type of reduction
- It is a terminal operation, that triggers the computation of the Stream

A First Example

- Let us collect a stream of strings in a list

```
List<String> peopleNames = ... ;

List<String> result = new ArrayList<>() ;

peopleNames.stream()
           .filter(s -> !s.isEmpty())
           .forEach(s -> result.add(s)) ;
```

- What is wrong in this pattern?

A First Example

- Let us collect a stream of strings in a list

```
List<String> peopleNames = ... ;

List<String> result = new ArrayList<>() ;

peopleNames.stream()
           .filter(s -> !s.isEmpty())
           .forEach(s -> result.add(s)) ;
```

- We add elements to an external list
- Which is final… and that cannot be accessed concurrently!

A First Example

- Let us collect a stream of strings in a list

```
List<String> peopleNames = ... ;

List<String> result = new ArrayList<>() ;

peopleNames.stream().parallel()
           .filter(s -> !s.isEmpty())
           .forEach(s -> result.add(s)) ;
```

- We add elements to an external list
- Which is final… and that cannot be accessed concurrently!

A First Example

- Let us collect a stream of strings in a list

```
List<String> peopleNames = ... ;

List<String> result = new ArrayList<>() ;

peopleNames.stream()
           .filter(s -> !s.isEmpty())
           .forEach(s -> result.add(s)) ;
```

A First Example

- Let us collect a stream of strings in a list

```
List<String> peopleNames = ... ;



peopleNames.stream()
           .filter(s -> !s.isEmpty())
           .collect(Collectors.toList()) ;
```

A First Example

- Let us collect a stream of strings in a list

```
List<String> peopleNames = ... ;


List<String> result =
peopleNames.stream()
           .filter(s -> !s.isEmpty())
           .collect(Collectors.toList()) ;
```

A First Example

- Let us collect a stream of strings in a list

```
List<String> peopleNames = ... ;


List<String> result =
peopleNames.stream().parallel()
           .filter(s -> !s.isEmpty())
           .collect(Collectors.toList()) ;
```

Collectors

- This step is called *mutable collection*
- Just because it collects the data in a *mutable* container

The Collectors Class

Ready to use collectors from the JDK

The Collectors Class

- The JDK provides a factory class: Collectors
- Collecting data is about gathering data in a mutable container
  - A String (concatenation)
  - A Collection (adding)
  - A HashMap (grouping by a criteria)

We Have Collectors for That

- Collector is the interface that models the collectors

- Collectors is the class factory to build collectors
  - Most collectors can be built through the factory
  - If needed there are other patterns

Collecting a Max

- Pattern:

```
List<Person> people = ... ;

Optional<Person> oldest =
people.stream()
      .collect(
         Collectors.maxBy(Comparator.comparing(p -> p.getAge())))
      ) ;
```

Collecting an Average

- Pattern :

```
List<Person> people = ... ;

double average =
people.stream()
      .collect(
         Collectors.averagingDouble(p -> p.getAge())
      ) ;
```

Collecting in a String

- The pattern is:

```
List<Person> people = ... ;

String names =
people.stream()
      .map(p -> p.getName())
      .collect(Collectors.joining(", ")) ;
```

```
Barbara, Charles, Sharon, Peter
```

Collecting in a Set

- The pattern is:

```
List<Person> people = ... ;

Set<String> names =
people.stream()
      .map(p -> p.getName())
      .collect(Collectors.toSet()) ;
```

Collecting in a Collection

- The pattern is:

```
List<Person> people = ... ;

TreeSet<String> names =
people.stream()
      .map(p -> p.getName())
      .collect(Collectors.toCollection(() -> new TreeSet())) ;
```

Collecting in a Map

- Partioning by a predicate:

```
List<Person> people = ... ;

Map<Boolean, List<Person>> peopleByAge =
people.stream()
      .collect(Collectors.partitioningBy(person -> person.getAge() > 21)) ;
```

Collecting in a Map

- Grouping by a function:

```
List<Person> people = ... ;

Map<Integer, List<Person>> peopleByAge =
people.stream()
      .collect(Collectors.groupingBy(person -> person.getAge())) ;
```

Collecting in a Map

- Grouping and counting:

```
List<Person> people = ... ;

Map<Integer, Long> peopleByAge =
people.stream()
      .collect(
         Collectors.groupingBy(person -> person.getAge()),
         Collectors.counting()
      ) ;
```

Collecting in a Map

- Grouping and counting:

```
List<Person> people = ... ;

Map<Integer, Long> peopleByAge =
people.stream()
      .collect(
         Collectors.groupingBy(person -> person.getAge()),
         Collectors.counting()  // the « downstream » collector
      ) ;
```

Collecting in a Map

- Grouping and mapping:

```
List<Person> people = ... ;

Map<Integer, List<String>> namesByAge =
people.stream()
      .collect(
         Collectors.groupingBy(person -> person.getAge()),
         Collectors.mapping(
            person -> person.getName()
         )
      ) ;
```

Collecting in a Map

- Grouping, mapping and collecting in a TreeSet:

```
List<Person> people = ... ;

Map<Integer, TreeSet<String>> namesByAge =
people.stream()
      .collect(
         Collectors.groupingBy(person -> person.getAge()),
         Collectors.mapping(
            person -> person.getName(),
            Collectors.toCollection(() -> TreeSet())
         )
      ) ;
```

Collecting in a Map

- Grouping in a TreeMap, mapping and collecting in a TreeSet:

```
List<Person> people = ... ;

TreeMap<Integer, TreeSet<String>> namesByAge =
people.stream()
      .collect(
         Collectors.groupingBy(person -> person.getAge()),
         () -> new TreeMap(),
         Collectors.mapping(
            person -> person.getName(),
            Collectors.toCollection(() -> TreeSet())
         )
      ) ;
```

Collecting in an Immutable Map

- Collecting in an immutable Map:

```
List<Person> people = ... ;

Map<Integer, List<Person>> peopleByAge =
people.stream()
      .collect(
         Collectors.collectingAndThen(
            Collectors.groupingBy(person -> person.getAge()),
            Collections::unmodifiableMap
         )
      ) ;
```

Live Coding

The Shakespeare plays Scrabble example

Live Coding Summary

- How to set up a data processing stream
  - Built on a large set of data
- Collecting the result in a hashmap
  - groupingBy, building histograms
  - Building streams on hashmap entrySet
- Extracting the meaningful information using a postprocessing
  - Sorting streams
  - Getting the first elements

Summary

- Collectors to reduce streams in containers: Strings, Collections, Maps
  - Concatenating strings with separators
  - Building collections: toList, toSet, toCollection
  - Building HashMaps: partitionningBy, groupingBy, downstream collectors
- Collectors in action on the Shakespeare example
  - Complex processing based on the use of histograms
  - Openning streams on hashmap entrySet to postprocess our data
- All of these in parallel!