Theory: Grouping collectors

22 minutes 0 / 5 problems solved

Skip this topic Start practicing

287 users solved this topic. Latest completion was about 8 hours ago.

We have learned how to accumulate stream elements into a collection or a single value by using `collect` operation and `Collectors` class. However, besides that, the `collect` can offer other useful operations such as dividing stream elements into two or more groups or applying a collector to the result of another collector. In this topic, we will see how to sort the elements of a stream by using `Collectors.partitioningBy` and `Collectors.groupingBy` methods. We will also learn what a downstream collector is and how to use it.

§1. Partitioning

Imagine that we want to divide a collection of accounts into two groups: accounts whose balance is greater than or equal to 10000, and accounts with a balance lower than 10000. In other words, we need to partition accounts into two groups based on a specified condition. It becomes possible by using *a partitioning* operation.

The partitioning operation is presented by the `Collectors.partitioningBy` method that accepts a predicate. It splits input elements into a `Map` of two lists: one list contains elements for which the predicate is true, and the other contains elements for which it is false. The keys of the `Map` has the `Boolean` type.

To illustrate the idea, let's create the following list of accounts:

```
List<Account> accounts = List.of(
        new Account(3333, "530012"),
        new Account(15000, "771843"),
        new Account(0, "681891")
);
```

And partition them into two lists by a `balance >= 10000` predicate:

```
Map<Boolean, List<Account>> accountsByBalance = accounts.stream()
        .collect(Collectors.partitioningBy(account -
> account.getBalance() >= 10000));
```

The `accountsByBalance` map contains the following entries:

```
{
    false=
[Account{balance=3333, number='530012'}, Account{balance=0, number='681891'}],
    true=[Account{balance=15000, number='771843'}]
}
```

The partitioning operation can produce a `Map` with empty lists, but they will always exist.

§2. Grouping

The grouping operation is similar to the partitioning. However, instead of splitting data into two groups based on a predicate, the grouping operation can produce any number of groups based on a *classification function* that maps elements to some key.

The grouping operation is presented by the `Collectors.groupingBy` method that accepts a classification function. The collector `groupingBy` also produces a `Map`. The keys of the `Map` are values produced by applying the classification function to the input elements. The corresponding values of the `Map` are lists containing elements mapped by the classification function.

Let's create the `Status` enum and add field `status` to the `Account` class:

Current topic:

Grouping collectors

Topic depends on:

× Collectors

Table of contents:

```
enum Status {
    ACTIVE,
    BLOCKED,
    REMOVED
}

public class Account {
    private long balance;
    private String number;

    private Status status;



    // constructors

    // getters and setters

}
```

Also, let's update the list of accounts:

```
List<Account> accounts = List.of(
        new Account(3333L, "530012", Status.REMOVED),
        new Account(15000L, "771843", Status.ACTIVE),
        new Account(0L, "681891", Status.BLOCKED)
);
```

Now, we can divide all account into groups by its `status`:

```
Map<Status, List<Account>> accountsByStatus = accounts.stream()
        .collect(Collectors.groupingBy(Account::getStatus));
```

The `accountsByStatus` map contains the following entries:

```
{
    BLOCKED=[Account{balance=0, number='681891'}],
    REMOVED=[Account{balance=3333, number='530012'}],
    ACTIVE=[Account{balance=15000, number='771843'}]
}
```

The grouping operation produces entries when needed, which means that the resulting `Map` may contain any number of entries. For example, if the input is an empty stream, the resulting `Map` will contain no entries.

§3. Downstream collectors

In addition to a predicate or a classification function, `partitioningBy` and `groupingBy` collectors can accept a **downstream** collector. Such a collector is applied to the results of another collector. For instance, `groupingBy` collector, which accepts a classification function and a downstream collector, groups elements according to a classification function, and then applies a specified downstream collector to the values associated with a given key.

To illustrate how it works, let's create the following list of accounts:

```
List<Account> accounts = List.of(
        new Account(3333L, "530012", Status.ACTIVE),
        new Account(15000L, "771843", Status.BLOCKED),
        new Account(15000L, "234465", Status.ACTIVE),
        new Account(8800L, "110011", Status.ACTIVE),
        new Account(45000L, "462181", Status.BLOCKED),
        new Account(0L, "681891", Status.REMOVED)
);
```

And calculate the total balances of `blocked`, `active`, and `removed` accounts using a downstream collector:

```
Map<Status, Long> sumByStatuses = accounts.stream()
        .collect(groupingBy(Account::getStatus, summingLong(Account::getBalance)))
;
```

The code above groups accounts by the `status` field and applies a downstream `summingLong` collector to the `List` values created by the `groupingBy` operator. The resulting map contains the following entries:

```
{ REMOVED=0, ACTIVE=24133, BLOCKED=60000 }
```

§4. Conclusion

To divide stream elements into exactly two groups based on a specified condition, we can use `Collectors.partitioningBy` collector. It accepts a predicate and produces a `Map` with `Boolean` keys and `List` values. If we need to divide stream elements into more than two groups, we can use `Collectors.groupingBy` collector. It accepts a classification function and groups elements according to it. The `groupingBy` also produces a `Map` with `Lists` values and keys whose type is a return type of the classification function. Both collectors can take a predicate or a classification function accordingly and a downstream collector that is applied to the results of partitioning or grouping.

Report a typo

29 users liked this theory. **1** didn't like it. **What about you?**

😍 🙂 😐 🙁 😡

Start practicing

Comments (4) Hints (0) Useful links (0) Show discussion