Infozettel: Parallelisierung von Datenströmen

Matthias Eurich – 2017-04-04

Table of Contents

Einführung

Datenströme lassen sich seriell oder parallel ausführen. Wird ein Datenstrom parallel ausgeführt, wird er von der *Java runtime* in mehrere Sub-Datenströme unterteilt. Nach der parallelen Ausführung der Sub-Datenströme werden die Ergebnisse am Ende zu einem Ergebnis zusammengeführt.

> In der Theorie wird zwar von **parallel** gesprochen, in der Praxis konkurrieren die Sub-Datenströme aber um die Ressourcen. Das bedeutet, dass, je nach Strategie, z.B. jeder Sub-Datenstrom eine feste Rechenzeit von z.B. 100ms bekommt. Danach bekommt der nächste Sub-Datenstrom die Ressourcen usw.. Es wird so dann so lange rotiert, bis jeder Sub-Datenstrom seine Aufgabe erfüllt hat. Nach außen hin wirkt diese Art der Verarbeitung wie eine parallele Verarbeitung. Man spricht hier auch von "Quasi-Parallelität".

Wird ein Datenstrom mit dem Aufruf von `Collection.stream()` oder `Arrays.stream()` begonnen, handelt es sich um einen seriellen Datenstrom. Um einen parallelen Datenstrom zu beginnen, muss bei einer `Collection` die Methode `Collection.parallelStream()` aufgerufen werden. Bei einem Array muss nach `Arrays.stream()` die Methode `BaseStream.parallel()` aufgerufen werden.

Beispiel:
Das erste Beispiel zeigt den Aufruf von `Collection.parallelStream()` am Beispiel einer Liste vom Typ `List<Integer>`.

```
jshell> List<Integer> numbers = Arrays.asList(1,2,3,4,5)
numbers ==> [1, 2, 3, 4, 5]

jshell> numbers.parallelStream()
$2 ==> java.util.stream.ReferencePipeline$Head@31a5c39e
```

Im zweiten Beispiel wird der Aufruf der Methode `BaseStream.parallel()` in Kombination mit `Arrays.stream()` gezeigt.

```
jshell> int[] numbers = new int[]{1,2,3,4,5}
numbers ==> int[5] { 1, 2, 3, 4, 5 }

jshell> Arrays.stream(numbers).parallel()
$2 ==> java.util.stream.IntPipeline$Head@e45f292
```

> Eine parallele Verarbeitung kostet zusätzliche Ressourcen für das Management der Sub-Datenströme. Wenn die zu bearbeitende Aufgabe weniger Ressourcen verbraucht als das Management der Sub-Datenströme, dann ist die parallele Ausführung sogar langsamer als eine serielle Ausführung.

Nebenläufige Reduktion

Das nachfolgende Beispiel zeigt einen seriellen Datenstrom, der Namen nach ihrem Anfangsbuchstaben in eine Map einsortiert:

```
jshell> List<String> names = Arrays.asList("Liam", "Kim", "Lisa", "Ben")
names ==> [Liam, Kim, Lisa, Ben]

jshell> Map<String, List<String>> byFirstChar = names.stream().
   ...> collect(Collectors.groupingBy(n -> n.substring(0,1)))
byFirstChar ==> {B=[Ben], K=[Kim], L=[Liam, Lisa]}
```

Das parallele Equivalent zum obrigen Beispiel ist:

```
jshell> ConcurrentMap<String, List<String>> byFirstChar = names.parallelStream().
   ...> collect(Collectors.groupingByConcurrent(n -> n.substring(0,1)))
byFirstChar ==> {B=[Ben], K=[Kim], L=[Lisa, Liam]}
```

Neben dem Aufruf von `parallelStream()` anstatt `stream()` hat sich ebenfalls der Aufruf der Methode `groupingBy()` zu `groupingByConcurrent()` geändert. Der Rückgabewert ist nun vom Typ `ConcurrentMap` anstatt `Map`. Diese Änderungen sind notwendig, da die Implementierung der Methode `groupingBy()` sehr schlecht im parallelen Betrieb arbeitet. Für die Methode `Collectors.toMap()` gibt es ebenfalls eine für die parallele Verarbeitung optimierte Variante: `Collectors.toConcurrentMap()`. `ConcurrentMap` ist eine für den parallelen Betrieb optimierte `Map`.

> Bei der parallelen Ausführung von Code muss auf das Verhalten (Performance) der verwendeten Datenstrukturen und Algorithmen im parallelen Betrieb geachtet werden.

Reihenfolge von Elementen bei Nebenläufigkeit

Die Reihenfolge, in der die Elemente eines Datenstroms verarbeitet werden, hängt davon ab, ob der Datenstrom seriell oder parallel ist.

Beispiel:
Das nachfolgende Beispiel zeigt die Ausgabe der Zahlen einer Liste vom Typ `List<Integer>`. Die Reihenfolge der Elemente in der Ausgabe bleibt gleich, egal wie oft sie ausgegeben werden.

```
jshell> List<Integer> ints = Arrays.asList(1,2,3,4,5,6,7,8,9)
ints ==> [1, 2, 3, 4, 5, 6, 7, 8, 9]

jshell> ints.stream().forEach(i -> System.out.print(i + " "))
1 2 3 4 5 6 7 8 9
```

Wird die Methode `parallelStream()` genutzt, ist die Ausgabe jedes mal unterschiedlich, je nachdem in welcher Reihenfolge die Sub-Datenströme ihre Ausgabe machen:

```
jshell> ints.parallelStream().forEach(i -> System.out.print(i + " "))
6 5 8 3 2 9 4 1 7
jshell> ints.parallelStream().forEach(i -> System.out.print(i + " "))
6 5 2 1 4 7 9 3 8
```

Eine Ausgabe, bei der die Elemente die gleiche Reihenfolge wie die Ausgangsliste aufweisen, ist durch den Aufruf der Methode `forEachOrdered()` möglich.

Beispiel:

```
jshell> ints.parallelStream().forEachOrdered(i -> System.out.print(i + " "))
1 2 3 4 5 6 7 8 9
```

Seiteneffekte

Eine Methode erzeugt einen Seiteneffekt, wenn sie neben der Erzeugung oder Rückgabe eines Wertes den Status des Computers verändert. Ein Beispiel für einen Seiteneffekt ist das Schreiben auf `System.out` durch einen Aufruf von `System.out.println()` innerhalb einer Methode.

Zustandsbehaftete Lambda-Ausdrücke

Ein Lambda-Ausdruck ist zustandsbehaftet, wenn sein Ergebnis von einem Zustand abhängt, der sich im Laufe der Verarbeitung ändern kann.

Beispiel:

Im nachfolgenden Beispiel werden Elemente innerhalb von `map()` der Liste `newList` hinzugefügt.

```
jshell> List<Integer> newList = new ArrayList<Integer>()
newList ==> []

jshell> ints.stream().map(i -> { newList.add(i); return i; }).forEach(i ->
System.out.print(i + " "));
1 2 3 4 5 6 7 8 9
```

Bei `i → { newList.add(i); return i; }` handelt es sich um einen zustandsbehafteten Lambda-Ausdruck, da sich das Ergebnis bei jeder Ausführung des Datenstroms ändern kann. Außerdem ist die Verwendung innerhalb von `map()` missverständlich.

Mehrmalige Ausgabe von `newList` bei Ausführung des obrigen Datenstroms mit `stream()`:
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9

Mehrmalige Ausgabe von `newList` bei Ausführung des obrigen Datenstroms mit `parallelStream()`:
6 5 2 1 4 7 9 3 8

1 3 6 2 4 5 8 7 9

Zustandsbehaftete Lambda-Ausdrücke gilt es zu vermeiden!

Last updated 2017-05-27 10:11:04 CEST