Infozettel: Stream Evaluation

Matthias Eurich – 2017-04-04

Table of Contents

Einführung
Eager Evaluation
Lazy Evaluation
    Anwendungsbeispiel

Einführung

In diesem Infozettel sollen verschiedenen Arten der Evaluierung von Ausdrücken am Beispiel von Java vorgestellt werden.

Java verwendet für die Evaluierung von Datenströmen, wie bereits im Infozettel *Stream Grundlagen* beschrieben, ein als **Lazy Evaluation** bezeichnetes Verfahren. Frei übersetzt bedeutet **Lazy Evaluation** so viel wie **bequeme (oder faule) Auswertung**. Der **Lazy Evaluation** steht die **Eager Evaluation** gegenüber. **Eager Evaluation** bedeutet, frei übersetzt, so viel wie **eifrige Auswertung**. Bei **Eager Evaluation** wird ein Ausdruck so früh wie möglich ausgewertet. Die Idee von **Lazy Evaluation** ist, das Ergebnis eines Ausdrucks nur so weit zu berechnen, wie es gerade benötigt wird.

Eager Evaluation

Die **Eager Evaluation** findet ihre Anwendung bei gewöhnlichen Methodenaufrufen. Die Evaluation wird sofort beim Aufruf durchgeführt.

Beispiel:
Zuerst legen wir die Methode `int doubleIt(int x)` an. Diese Methode verdoppelt den Wert `x` und gibt das Ergebnis zurück.

```
jshell> int doubleIt(int x){return x * 2;}
|  created method doubleIt(int)
```

Rufen wir die Methode anschließend auf, findet eine direkte Auswertung des Aufrufs statt. Dies sieht man daran, dass direkt das Ergebnis geliefert wird.

```
jshell> doubleIt(4)
$2 ==> 8
```

Lazy Evaluation

Im nachfolgenden Beispiel wird die erste gerade Zahl, die größer als `3` ist, verdoppelt.

```
jshell> List<Integer> values = Arrays.asList(1,2,3,5,4,6,7,8,9,10);
values ==> [1, 2, 3, 5, 4, 6, 7, 8, 9, 10]

jshell> values.stream().
   ...> filter(n -> n > 3).
   ...> filter(n -> n % 2 == 0).
   ...> map(n -> n * 2).
   ...> findFirst()
$5 ==> Optional[8]
```

Schaut man sich das Beispiel an, denkt man zunächst, dass die Zahlen der Liste `values` zuerst durch den Aufruf von `filter(n → n > 3)` auf `5,4,6,7,8,9,10` und anschließend durch `filter(n → n % 2 == 0)` auf `4,6,8,10` gefiltert, bevor sie mit `map(n → n * 2)` zu `8,12,16,20` verdoppelt werden. Am Ende wird dann das erste Element, also `8`, zurückgegeben. Der Code liest sich zwar wie zuvor beschrieben, jedoch wäre eine solche Auswertung höchst ineffizient.

Durch Auslagern der `Predicate` der beiden Filter-Operationen sowie der `Function` der `map()`-Operation kann der Zeitpunkt der Evaluierung durch einen Log sichtbar gemacht werden.

Nachfolgend der Code mit den besagten Änderungen:

```
jshell> Predicate<Integer> isGreaterThan3 = n -> {
   ...> System.out.println("isGreaterThan3 " + n);
   ...> return n > 3;}
isGreaterThan3 ==> $Lambda$29/1414521932@31610302

jshell> Predicate<Integer> isEven = n -> {
   ...> System.out.println("isEven " + n);
   ...> return n % 2 == 0;}
isEven ==> $Lambda$30/1899073220@21213b92

jshell> Function<Integer, Integer> doubleIt = n -> {
   ...> System.out.println("doubleIt " + n);
   ...> return n * 2;}
doubleIt ==> $Lambda$31/174573182@3327bd23

jshell> values.stream().
   ...> filter(isGreaterThan3).
   ...> filter(isEven).
   ...> map(doubleIt).
   ...> findFirst();
isGreaterThan3 1
isGreaterThan3 2
isGreaterThan3 3
isGreaterThan3 5
isEven 5
isGreaterThan3 4
isEven 4
doubleIt 4
$17 ==> Optional[8]
```

Sobald eine Zahl gefunden wurde, die größer als `3` ist, wird geprüft, ob diese Zahl gerade ist. Da 5 ungerade ist, wird die nächste Zahl (4) aus `values` genommen, die größer als `3` ist. Die Zahl 4 ist größer als `3` und eine gerade Zahl. Also wird sie verdoppelt und zurückgegeben.

> Eine ähnliche Verarbeitung lässt sich auch mit einer `for` -Schleife lösen. Wenn sie die `for` -Schleife beim Fund einer passenden Zahl nicht mit einem `break` oder `return` frühzeitig terminieren, werden die Überprüfungen, die Transformation und Ausgabe für alle Werte durchgeführt. Das mag bei 10 Werten nicht tragisch klingen, bei 10^10 Werten wird es jedoch zum Problem.

Wird der verkettete Ausdruck ohne die terminale Operation `findFirst()` aufgerufen, findet keine Evaluierung statt, wie die fehlenden Logs der nachfolgenden Ausgabe auf der JShell zeigen. Die JShell merkt sich lediglich den Ausdruck in der Variablen `$18` .

```
jshell> values.stream().filter(isGreaterThan3).filter(isEven).map(doubleIt)
$18 ==> java.util.stream.ReferencePipeline$3@5383967b
```

Anwendungsbeispiel

Die **Lazy Evaluation** erlaubt das Schreiben von performantem Code für besonders große oder unendliche Datenmengen. Im nachfolgenden Beispiel werden die ersten fünf Primzahlen einer unendlichen Folge von Zahlen bestimmt.

Das `Predicate isPrime` bestimmt, ob eine Zahl eine Primzahl ist oder nicht.

```
jshell> IntPredicate isPrime = number ->
   ...> number > 1 &&
   ...> IntStream.range(2, number).noneMatch(index -> number % index == 0);
isPrime ==> $Lambda$37/1735934726@335eadca

jshell> IntStream.iterate(0, i -> i +
1).filter(isPrime).limit(5).forEach(System.out::println)
2
3
5
7
11
```

Last updated 2017-05-26 10:34:40 CEST