Nachdenkzettel: Software-Entwicklung 2, Streams processing

Parameter. Die in Stream gibt 🡪beeinflussen das Verhalten der Streams🡪Verhalten paramerisieren

<->Collections: gibt Daten aus und parametisiert dann

Parameter=Funktion🡪ruft filter auf

1. Filtern sie die folgende Liste mit Hilfe eines Streams nach Namen mit „K“ am Anfang:

final List<String> names = Arrays.asList(“John”, “Karl”, “Steve”, “Ashley”, “Kate”);

names.stream()

.filter(s->s.startsWith(“K”))

(.map(String::toUpperCase)🡪(=s.toUpperCase)Transformator der Liste kann in jegliches Objekt umwandeln

.sorted()🡪wird sortiert

.forEach(System.out::println)🡪für jedes Element: wird outprinted)

. 🡪operiert immer auf Stream 🡪for Each = definiert als Terminalmethode 🡪beendet Stream 🡪Stream geht auch nur los, wenn for Each am Ende

2. Welche 4 Typen von Functions gibt es in Java8 und wie heisst ihre Access-Methode?

Tipp: Stellen Sie sich eine echte Funktion vor (keine Seiteneffekte=Daten des Streams werden nicht verändert) und variieren Sie die verschiedenen Teile der Funktion.

Prädikatsfunktion: Code muss true/false zurückgeben Filter(),

Transormierende Funktionen:map(),

Consumer Funktion: besitzt Parameter, gibt aber nichts zurück (kann zurückgebenes konsumieren z.B.in Konsole ausgeben aber mehr nicht) println()

Producerfunktion: z.B. ans Netzwerk: sind daten da? 🡪übergibt keine Parameter, erhält aber Daten zurück stream():

3. forEach() and peek() operieren nur über Seiteneffekte. Wieso?

Weil sie als ausnahmefunktionen Daten akkumulieren und verändern müssen, sie dürfen also nicht stateless sein🡪Ausführung dieser Funktionen nicht mehr unabhängig von der Reihenfolge

4. sort() ist eine interessante Funktion in Streams. Vor allem wenn es ein paralleler Stream ist. Worin liegt das Problem?

🡪Pipeline: überlappende Verarbeitungsphasen (CPU)

Sortiert Daten 🡪parallele Threads liefern Teilergebnisse, die man später noch zusammenführen muss

5. Achtung: Erklären Sie was falsch oder problematisch ist und warum.

a) Set<Integer> seen = new HashSet<>();

someCollection.parallel().map(e -> { if (seen.add(e)) return 0; else return e; })

Der Stream liefert keine Werte da keine foreach() Methode den Stream beendet.

b) Set<Integer> seen = Collections.synchronizedSet(new HashSet<>());

someCollection.parallel().map(e -> { if (seen.add(e)) return 0; else return e; })

die parallelen Streams geben unterschiedliche Ergebnisse zurück🡪parallele Prozesse

6. Ergebnis?

List<String> names = Arrays.asList(“1a”, “2b”, “3c”, “4d”, “5e”);

names.stream()

.map(x → x.toUppercase())

.mapToInt(x → x.pos(1)

.filter(x → x < 5)

1.—Alle Funktionen werden für jedes Element aufgerufen

Wenn Sie schon am Grübeln sind, erklären Sie doch bei der Gelegenheit warum es gut ist, dass Streams „faul“ sind.

🡪weil damit verhindert werden kann, dass Objekte zweimal geprüft werden 🡪wenn zuerst filetred und ergebnis schon erhält, bricht Pipeline ab

🡪die Listenelemete werden mehrermals ausgelsen, da die filter Option erst am Ende steht und vorher alle Objekte mapped und in Uppercase umwandelt

7. Wieso braucht es das 3. Argument in der reduce Methode?

Hat Basisfunktionalität, braucht aber noch Akkumulator 🡪Parameter 🡪collect-Function:returned collection (reduce nur einen Wert)

Bis jetzt nur single-threaded gearbeitet 🡪3.Parameter .parallelStream =Erlaubnis mehrere Cores zu verwenden 🡪deswegen Seiteneffekt frei und unabhängig🡪Threads erzeugen Teilergebnisse 🡪3.Parameter: combiner-Funktion bringt bei, wie macht

List<Person> persons = Arrays.*asList*(

**new** Person("Max", 18, 4000),

**new** Person("Peter", 23, 5000),

**new** Person("Pamela", 23, 6000),

**new** Person("David", 12, 7000));

**int** money = persons

.parallelStream()

.filter(p -> p.salary > 5000)

.reduce(0, (p1, p2) -> ( p1 + p2.salary), (s1, s2)-> (s1 + s2));

*log*.debug("salaries: " + money);

Tipp: Stellen Sie sich eine Streamsarchitektur vor (schauen Sie meine Slides an).

Am Anfang ist eine Collection. Sie haben mehrere Threads zur Verfügung. Mit was fangen Sie an? Dann haben die Threads gearbeitet. Was muss dann passieren?

8. Was ist der Effekt von stream.unordered() bei sequentiellen Streams und bei parallelen streams?

Keinen Effekt auf sequentielle Streams, aber bei parallelen Streams wird durch das unordered die Reihenfolge geändert und durch die parallel laufenden Streams der Output verändert

9. Fallen

a) IntStream stream = IntStream.of(1, 2);

stream.forEach(System.out::println);

stream.forEach(System.out::println);

die Elemente warden nur einmal ausgegeben, da nach dem ersten forEach der Stream beendet wird

b) IntStream.iterate(0, i -> i + 1)

.forEach(System.out::println);

Unendlicher Stream gestartet

c) IntStream.iterate(0, i -> ( i + 1 ) % 2)

.distinct() //.parallel()? 🡪parallele unendliche Streams

.limit(10)

.forEach(System.out::println);

Distinct weiß nicht, dass iterate nur zwei verschiedene Werte (0,1,0,1) liefert und damit wird trotzdem ein unedlicher Stream gestartet und alle Zahlen kosumiert

d) List<Integer> list = IntStream.*range*(0, 10)

.boxed()

.collect(Collectors.*toList*());

list.stream()

.peek(list::remove)

.forEach(System.***out***::println);

🡪nicht alle Elemente der Liste werden ausgegeben bzw. gelöscht, da die Elemente „vergessen“werden 🡪erst mit sorted() vor der remove() anweisung liefert das Ergbenis, dass alle Listenelemente vor dem Löschen ausgegeben werden

from: Java 8 Friday: http://blog.jooq.org/2014/06/13/java-8-friday-10-subtle-mistakes-when-using-the-streams-api/