Java™

ORACLE®

Working through Streams Implementation, Debugging and many more …

Java
Your
Next
(Cloud)

Vaibhav Choudhary (@vaibhav_c)
Java Platforms Team
https://blogs.oracle.com/vaibhav

Safe Harbor Statement

Agenda of the day …

Basics of Streams and ParallelStreams

Introduction

- Java 8 came with the biggest change ever in terms of Streams and Lambda.
- Advocates Functional Programming
- Focus on “what to do” rather than “how to do”
- Can leverage the beauty of cores with ParallelStreams.
- IDE’s are supporting it and started providing enough information around it.
- Some IDE’s like (eclipse) support to convert existing code into Lambda code.

Please follow our series of Talks …

- An inside view of Streams - http://bangalorejug.org/wp-content/uploads/2017/06/An-inside-view-of-Streams-And-Lazy-Evaluation.pdf
- Streams vs ParallelStreams - http://files.meetup.com/3189882/streamsvsparallelstreams.pdf

Small talk on internals (From prev. Talks)

Understanding Streams (From prev. Talks)

- Streams exploits the most powerful computation principal - Composition
- Before entering into the laziness of Streams, first understand the basic API's and it's internals.
- All streams share a common structure
  - A stream source
    - Intermediate operations
      - Single terminal operation

Streams Under-hood - Source

- Source of the stream is a Spliterator
- Spliterator = Iterator + Split (if possible)
- Not extended from Iterator. Iterator as in defensive and duplicative.
- hasNext() and next() - count is problem.

Using Lambdas, Spliterator has 2 methods to access element :-

```
boolean tryAdvance(Consumer<? super T> action); // single element op
void forEachRemaining(Consumer<? super T> action);  // bulk op
```

Executing a Stream pipeline - Intermediate/Terminal operations

- When terminal operation is initiated, the stream picks the execution plan.
- Intermediate operations :-
  - stateless : filter(), map() ..
  - stateful : sort(), limit(), distinct()
  - If stateless operation, it can compute in single pass.
  - If stateful, pipelines are divided into sections and computer in multiple pass.
- Terminal Operations :-
  - short-circuiting : allMatch(), findFirst() [**tryAdvance()**]
  - non-short-circuiting : reduce(), collect(), forEach() [**forEachRemaining()**]

Debugging Streams API

Streams simplified …

- List of element -> find c letter guys -> change to upper case -> sort it -> print it

```
List<String> myList =
        Arrays.asList("apple", "mango", "orange", "cabbage", "capsicum");
myList
        .stream()
        .filter(s -> s.startsWith("c"))
        .map(String::toUpperCase)
        .sorted()
        .forEach(System.out::println);
```

Let’s peek it...

- Need to know the floating values in the pipe

```
List<String> myList =
        Arrays.asList("apple", "mango", "orange", "cabbage", "capsicum");
myList
        .stream()
        .filter(s -> s.startsWith(“c"))
        .peek(s -> System.out.println(s))
        .map(String::toUpperCase)
        .sorted()
        .forEach(System.out::println);
```

Definition for peek: Stream<T> peek(Consumer<? super T> action)
JavaDoc specify that we should use peek for debugging purpose.

Let’s peek it…

- Need to know the floating values in the parallel pipe

```
List<String> myList =
        Arrays.asList("apple", "mango", "orange", "cabbage", "capsicum");
myList
        .stream().parallel()
        .filter(s -> s.startsWith(“c"))
        .peek(s -> System.out.println(Thread.currentThread()))
        .map(String::toUpperCase)
        .sorted()
        .forEach(System.out::println);
```

Parallel stream – You can clearly see how many threads has been created for you by FJ Framework

Intellij Stream Debugger View - A plugin

```
Stream.of("Apple","Mango","Banana","Capsicum", "Cabbage", "Cauliflower")
        .filter(s -> s.startsWith("C"))
        .map(s -> s.toLowerCase())
        .sorted()
        .forEach(s -> System.out.println(s));
```

Common Mistakes while using Streams

Example 1

```
IntStream stream = IntStream.of(1, 2);
stream.forEach(System.out::println);

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

Guess the output ?

Example 2

```
IntStream.iterate(0, i -> ( i + 1 ) % 2)
        .distinct()
        .limit(10)
        .forEach(System.out::println);
```

Guess the output ?

Example 2

```
IntStream.iterate(0, i -> ( i + 1 ) % 2)
         .parallel()
         .distinct()
         .limit(10)
         .forEach(System.out::println);
```

Guess the output ?

Example 3

```
ArrayList<Integer> integers = new ArrayList<>(Arrays.asList(1,2,3));

for (Integer integer : integers) {
    integers.remove(2);
}
```

Guess the output ?

Example 4

```
 final List<Integer> ints = new ArrayList<>();
    ints.add(1);
    ints.add(2);
    ints.add(3);
    ints.add(4);

    ints.stream()
            .map(a -> addSomething( a, ints ))
            .collect(Collectors.toList());

}

private static int addSomething(int current, List<Integer> ints) {
    ints.add(1);
    return current;
}
```

Guess the output ?

Example 5

```
IntStream.iterate(0, i -> i + 1)
        .limit(10) // LIMIT
        .skip(5)   // OFFSET
        .forEach(System.out::println);
```

Guess the output ?

Example 6

```
IntStream.range(1, 5)
        .peek(System.out::println)
        .peek(i -> {
            if (i == 5)
                throw new RuntimeException("bang");
        });
```

Guess the output ?

Example 7

```
IntStream.of(1, 2, 3, 4).forEach(
        element -> System.out.println(element + 1));
```

Guess the output ?

Must visit References

- Java Stream Debugger plugin - https://plugins.jetbrains.com/plugin/9696-java-stream-debugger
- Top 10 Streams Mistakes - https://blog.jooq.org/2014/06/13/java-8-friday-10-subtle-mistakes-when-using-the-streams-api/
- An inside view of Streams - http://bangalorejug.org/wp-content/uploads/2017/06/An-inside-view-of-Streams-And-Lazy-Evaluation.pdf
- Streams vs ParallelStreams - http://files.meetup.com/3189882/streamsvsparallelstreams.pdf