
Java™
ORACLE

An inside view of Streams & Lazy Evaluation

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

Java Your Next (Cloud)

Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Agenda of the day …

1. Different Evaluation Strategy
2. Defining Streams
3. Streams Sources
4. Stream Pipeline - Building and Executing
5. Leveraging Laziness

Different Evaluation Strategy

- Eager Evaluation
  - expression is evaluated as soon as it is bound to a variable.
  - easy to control and no need to track the expression.
- Short-circuiting
  - && and || **(in case of java)**
  - special case of lazy evaluation.
  - careful with the side effects.
- Lazy Evaluation
  - call by need
  - ability to define potentially infinite data structures.

Understanding Streams

- 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

So, how it different from collections

- data structure vs computation
  - collection can be source or destination for streams.
- mutability vs transformation
  - filtering a stream will produce a new stream
- eager vs lazy evaluation
  - computation is done by need not on greed

Simple Example

```
int totalSalesFromNY
    = txns.stream()
          .filter(t -> t.getSeller().getAddr().getState().equals("NY"))
          .mapToInt(t -> t.getAmount())
          .sum();
```

Can you identify the source, intermediate operation and terminal operation ?

Code like Problem Statement

- Problem statement - "Print the names of distinct sellers in transactions with buyers over age 65, sorted by name."

```
Set<Seller> sellers = new HashSet<>();
for (Txn t : txns) {
    if (t.getBuyer().getAge() >= 65)
        sellers.add(t.getSeller());
}
List<Seller> sorted = new ArrayList<>(sellers);
Collections.sort(sorted, new Comparator<Seller>() {
    public int compare(Seller a, Seller b) {
        return a.getName().compareTo(b.getName());
    }
});
for (Seller s : sorted)
    System.out.println(s.getName());
```

```
txns.stream().parallel()
    .filter(t -> t.getBuyer().getAge() >= 65)
    .map(Txn::getSeller)
    .distinct()
    .sorted(comparing(Seller::getName))
    .map(Seller::getName)
    .forEach(System.out::println);
```

"Expressing a stream pipeline as a series of functional transformations enables several useful execution strategies, such as laziness, parallelism, short-circuiting, and operation fusion."

Accumulator Anti-pattern

Traditional way :-

```
int sum = 0;
for (Txn t : txns) {
    if (t.getSeller().getAddr().getState().equals("NY"))
        sum += t.getAmount();
}
```

Streams way :-

```
int totalSalesFromNY
    = txns.stream()
          .filter(t -> t.getSeller().getAddr().getState().equals("NY"))
          .mapToInt(t -> t.getAmount())
          .sum();
```

What's wrong in traditional style of coding, other than the fact that it's bad to read ?

Cont…

- Reduction is the general solution of all imperative style of accumulation.
- Sum is also a special case of reduction.
- To apply reduction, operator should follow associativity
  - ((a*b)*c) = (a*(b*c)) where * is the binary operator
  - (((a*b)*c)*d) = ((a*b)*(c*d)) => sequential to parallel

```
int reduce(int identity, IntBinaryOperator op);
```

```
int sum = integers.reduce(0, (a, b) -> a+b);
```

```
String concatenated = strings.stream().reduce("", String::concat);
```

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
```

Streams Under-hood - Cont…

- Even ignoring parallelism, Spliterator is a better approach than Iterator.
- Encounter Order is important.
- Collections implementation in the JDK has been furnished to use Spliterator.
- ArrayList a better split source, LinkedList not a better source.

Spliterator provides to split the source, so that 2 thread can work in parallel :-

```
Spliterator<T> trySplit();
```

Building a Stream Pipeline

- Stream pipeline is build by linkedList of its source and intermediate operations.
- Each stage of stream uses Flag for optimal construction and execution.
  - SIZED, DISTINCT, SORTED, ORDERED

```
TreeSet<String> ts = ...
String[] sortedAWords = ts.stream()
                          .filter(s -> s.startsWith("a"))
                          .sorted()        // no-op
                          .toArray();
```

Executing a Stream pipeline

- 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()**]

Cont…

- For Sequential execution, Stream construct “Machine”.
- Machine - a chain of consumer.
- For stateless ops, it finishes the work and pass it to next stage.
- For stateful ops, consumer has to see the end of the input stream.
- In the final stage, it will get a terminal operation (like reduce()).

```
blocks.stream()
      .map(block -> block.squash())
      .filter(block -> block.getColor() != YELLOW)
      .forEach(block -> block.display());
```

“When performance is critical, it's valuable to understand how the library works internally.”

Why to know the working …

Execution of stream pipelines can also be optimized through the use of stream flags. For example, the SIZED flag indicates that the size of the final result is known. The toArray() terminal operation can use this flag to preallocate the correct-size array; if the SIZED flag isn't present, it would have to guess at the array size and possibly copy the data if the guess is wrong.

Leveraging laziness by Streams

- Expressiveness of the code
  - Code use the expressive
- Run the infinite series
  - Infinite series with finite computation.
  - Lazy evaluation makes the deal.
- Demo

Must visit Reference

- Brain Goetz : https://www.ibm.com/developerworks/java/library/j-java-streams-1-brian-goetz/index.html?ca=drs-
  (Most of the slides are from this source)

- Venkat's Video : https://www.youtube.com/watch?v=uiaspJ2DIlU

- Brain Goetz Talks on : From Concurrency to Parallelism