Java Parallel Stream Internals: Non-Concurrent & Concurrent Collectors (Part 1)

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science
Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

• Understand parallel stream internals, e.g.
  • Know what can change & what can’t
  • Partition a data source into “chunks”
  • Process chunks in parallel via the common fork-join pool
  • Configure the Java parallel stream common fork-join pool
  • Perform a reduction to combine partial results into a single result
  • Recognize key differences between non-concurrent & concurrent collectors
Overview of Concurrent & Non-Concurrent Collectors
Collector defines an interface whose implementations can accumulate input elements in a mutable result container.

**Interface Collector<T,A,R>**

Type Parameters:

- T - the type of input elements to the reduction operation
- A - the mutable accumulation type of the reduction operation (often hidden as an implementation detail)
- R - the result type of the reduction operation

```java
public interface Collector<T,A,R>
```

A mutable reduction operation that accumulates input elements into a mutable result container, optionally transforming the accumulated result into a final representation after all input elements have been processed. Reduction operations can be performed either sequentially or in parallel.

Examples of mutable reduction operations include: accumulating elements into a Collection; concatenating strings using a StringBuilder; computing summary information about elements such as sum, min, max, or average; computing "pivot table" summaries such as "maximum valued transaction by seller", etc. The class Collectors provides implementations of many common mutable reductions.

A Collector is specified by four functions that work together to accumulate entries into a mutable result container, and optionally perform a final transform on the result. They are:

See [docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html](http://docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.html)
Overview of Concurrent & Non-Concurrent Collectors

- Collector implementations can either be concurrent or non-concurrent based on their characteristics.

**Enum Collector.Characteristics**

```java
java.lang.Object
java.lang.Enum<Collector.Characteristics>
java.util.stream.Collector.Characteristics
```

**All Implemented Interfaces:**
Serializable, Comparable<Collector.Characteristics>

**Enclosing Interface:**
Collector<T,A,R>

```java
public static enum Collector.Characteristics
extends Enum<Collector.Characteristics>
```

Characteristics indicating properties of a Collector, which can be used to optimize reduction implementations.

**Enum Constant Summary**

**Enum Constants**

<table>
<thead>
<tr>
<th>Enum Constant and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CONCURRENT</strong></td>
</tr>
<tr>
<td>Indicates that this collector is concurrent, meaning that the result container can support the accumulator function being called concurrently with the same result container from multiple threads.</td>
</tr>
<tr>
<td><strong>IDENTITY_FINISH</strong></td>
</tr>
<tr>
<td>Indicates that the finisher function is the identity function and can be elided.</td>
</tr>
<tr>
<td><strong>UNORDERED</strong></td>
</tr>
<tr>
<td>Indicates that the collection operation does not commit to preserving the encounter order of input elements.</td>
</tr>
</tbody>
</table>

See [docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.Characteristics.html](docs.oracle.com/javase/8/docs/api/java/util/stream/Collector.Characteristics.html)
Overview of Concurrent & Non-Concurrent Collectors

• Collector implementations can either be concurrent or non-concurrent based on their characteristics

• This distinction is only relevant for parallel streams

See "Java Streams: Introducing Non-Concurrent Collectors"
Overview of Concurrent & Non-Concurrent Collectors

• Collector implementations can either be concurrent or non-concurrent based on their characteristics
  • This distinction is only relevant for parallel streams
• A non-concurrent collector can be used for either a sequential stream or a parallel stream!

We just focus on parallel streams in this lesson
Structure & Functionality of Non-Concurrent Collectors
Structure & Functionality of Non-Concurrent Collectors

- A non-concurrent collector operates by merging sub-results

See stackoverflow.com/questions/22350288/parallel-streams-collectors-and-thread-safety
Structure & Functionality of Non-Concurrent Collectors

- A non-concurrent collector operates by merging sub-results
- The input is partitioned into chunks
Structure & Functionality of Non-Concurrent Collectors

- A non-concurrent collector operates by merging sub-results
  - The input is partitioned into chunks
  - Each chunk runs in parallel in the common fork-join pool
Structure & Functionality of Non-Concurrent Collectors

• A non-concurrent collector operates by merging sub-results
  • The input is partitioned into chunks
  • Each chunk runs in parallel in the common fork-join pool
• Chunk sub-results are collected into an intermediate mutable result container
  • e.g., list, set, map, etc.
Structure & Functionality of Non-Concurrent Collectors

- A non-concurrent collector operates by merging sub-results
  - The input is partitioned into chunks
  - Each chunk runs in parallel in the common fork-join pool
- Chunk sub-results are collected into an intermediate mutable result container
  - e.g., list, set, map, etc.

Different threads operate on different instances of intermediate result containers
A non-concurrent collector operates by merging sub-results

- The input is partitioned into chunks
- Each chunk runs in parallel in the common fork-join pool
- Chunk sub-results are collected into an intermediate mutable result container
- Sub-results are merged into one mutable result container
A non-concurrent collector operates by merging sub-results

1. The input is partitioned into chunks
2. Each chunk runs in parallel in the common fork-join pool
3. Chunk sub-results are collected into an intermediate mutable result container
4. Sub-results are merged into one mutable result container
5. Only one thread in the fork-join pool is used to merge any pair of intermediate sub-results
A non-concurrent collector operates by merging sub-results

- The input is partitioned into chunks
- Each chunk runs in parallel in the common fork-join pool
- Chunk sub-results are collected into an intermediate mutable result container
- Sub-results are merged into one mutable result container
  - Only one thread in the fork-join pool is used to merge any pair of intermediate sub-results

Thus there’s no need for any synchronizers in a non-concurrent collector collector.
**Structure & Functionality of Non-Concurrent Collectors**

- A non-concurrent collector operates by merging sub-results
  - The input is partitioned into chunks
  - Each chunk runs in parallel in the common fork-join pool
  - Chunk sub-results are collected into an intermediate mutable result container
- Sub-results are merged into one mutable result container

*This process is safe & order-preserving, but costly for containers like maps & sets*
Structure & Functionality of Concurrent Collectors
A concurrent collector creates one concurrent mutable result container & accumulates elements into it from multiple threads in a parallel stream.

A concurrent collector creates one concurrent mutable result container & accumulates elements into it from multiple threads in a parallel stream.

As usual, the input is partitioned into chunks.
Structure & Functionality of Concurrent Collectors

- A concurrent collector creates one concurrent mutable result container & accumulates elements into it from multiple threads in a parallel stream
- As usual, the input is partitioned into chunks
- Each chunk runs in parallel in the common fork-join pool

```
InputSource
  trySplit()  trySplit()
     InputSource₁    InputSource₂
         trySplit()  trySplit()
            InputSource₁₁  InputSource₁₂  InputSource₂₁  InputSource₂₂

Process sequentially  Process sequentially  Process sequentially  Process sequentially
accumulate()  accumulate()  accumulate()  accumulate()

Concurrent Result Container
```

A pool of worker threads
Structure & Functionality of Concurrent Collectors

- A concurrent collector creates one concurrent mutable result container & accumulates elements into it from multiple threads in a parallel stream
  - As usual, the input is partitioned into chunks
  - Each chunk runs in parallel in the common fork-join pool
  - Chunk sub-results are collected into one mutable result container
    - e.g., a concurrent collection

See [docs.oracle.com/javase/tutorial/essential/concurrency/collections.html](docs.oracle.com/javase/tutorial/essential/concurrency/collections.html)
A concurrent collector creates one concurrent mutable result container & accumulates elements into it from multiple threads in a parallel stream.

As usual, the input is partitioned into chunks.

Each chunk runs in parallel in the common fork-join pool.

Chunk sub-results are collected into one mutable result container.

E.g., a concurrent collection.

**Different threads in a parallel stream share one concurrent result container**
Structure & Functionality of Concurrent Collectors

- A concurrent collector creates one concurrent mutable result container & accumulates elements into it from multiple threads in a parallel stream
- As usual, the input is partitioned into chunks
- Each chunk runs in parallel in the common fork-join pool
- Chunk sub-results are collected into one mutable result container

Thus there’s no need to merge any intermediate sub-results!

Of course, encounter order is not preserved & synchronization is required..
A concurrent collector *may* outperform a non-concurrent collector *if* merging costs are higher than synchronization costs.
Structure & Functionality of Concurrent Collectors

• A concurrent collector *may* out-perform a non-concurrent collector *if* merging costs are higher than synchronization costs

• Highly optimized result containers like ConcurrentHashMap may be more efficient than merging HashMaps

See codepumpkin.com/hashtable-vs-synchronizedmap-vs-concurrent hashmap
Structure & Functionality of Concurrent Collectors

- A concurrent collector may out-perform a non-concurrent collector if merging costs are higher than synchronization costs
- Highly optimized result containers like ConcurrentHashMap may be more efficient than merging HashMaps
- ConcurrentHashMap is also more efficient than a SynchronizedMap

End of Java Parallel Stream Internals: Non-Concurrent & Concurrent Collectors (Part 1)