Java 8 Parallel Stream Internals

(Part 6)

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
  - Configure the Java 8 parallel stream common fork-join pool
  - Avoid pool starvation & improve performance w/ManagedBlocker
  - Perform a reduction that combines partial results into a single result
- Learn to implement concurrent & non-concurrent collectors
Implementing Concurrent & Non-Concurrent Collectors
Implementing Concurrent & Non-Concurrent Collectors

- Collector defines an interface whose implementations can accumulate input elements in a mutable result container.

```java
public 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

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
Implementing Concurrent & Non-Concurrent Collectors

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

Implementing Concurrent & Non-Concurrent Collectors

• Collector implementations can either be non-concurrent or concurrent based on their characteristics
  • This distinction is only relevant for parallel streams

See "Overview of Java 8 Streams (Part 4)" for non-concurrent collector implementation
Implementing Concurrent & Non-Concurrent Collectors

- Collector implementations can either be non-concurrent or 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’ll just focus on parallel streams in the subsequent discussion
Implementing Concurrent & Non-Concurrent Collectors

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

Implementing Concurrent & Non-Concurrent Collectors

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

- A non-concurrent collector operates by merging sub-results
  - The input source is partitioned into chunks
  - Each chunk runs in parallel & is collected into an intermediate mutable result container
    - e.g., a list or a map

```
InputSource
  trySplit()
  \[\text{InputSource}_1\]
    trySplit()
    \[\text{InputSource}_{1,1}\] \(\rightarrow\) \(\rightarrow\) \(\rightarrow\)
    \(\text{Process sequentially}\)
  \[\text{InputSource}_2\]
    trySplit()
    \[\text{InputSource}_{2,1}\] \(\rightarrow\) \(\rightarrow\) \(\rightarrow\)
    \(\text{Process sequentially}\)
```

```
join
```

```
join
```

```
join
```

```
join
```
Implementing Concurrent & Non-Concurrent Collectors

- A non-concurrent collector operates by merging sub-results
  - The input source is partitioned into chunks
  - Each chunk runs in parallel & is collected into an intermediate mutable result container
    - e.g., a list or a map

Different threads operate on different instances of intermediate result containers
Implementing Concurrent & Non-Concurrent Collectors

- A non-concurrent collector operates by merging sub-results
  - The input source is partitioned into chunks
  - Each chunk runs in parallel & is collected into an intermediate mutable result container
  - These sub-results are then merged into a final mutable result container
    - Only one thread in the fork-join pool is used to merge any pair of intermediate sub-results
Implementing Concurrent & Non-Concurrent Collectors

- A non-concurrent collector operates by merging sub-results
  - The input source is partitioned into chunks
  - Each chunk runs in parallel & is collected into an intermediate mutable result container
  - These sub-results are then merged into a final 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
Implementing Concurrent & Non-Concurrent Collectors

- A non-concurrent collector operates by merging sub-results
  - The input source is partitioned into chunks
  - Each chunk runs in parallel & is collected into an intermediate mutable result container
  - These sub-results are then merged into a final mutable result container

*This process is safe & order-preserving, but costly for containers like maps & sets*
Implementing Concurrent & Non-Concurrent Collectors

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

See stackoverflow.com/questions/22350288/parallel-streams-collectors-and-thread-safety
Implementing Concurrent & Non-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 source is partitioned into chunks
Implementing Concurrent & Non-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 source is partitioned into chunks
- Each chunk runs in parallel & is collected into one concurrent mutable result container
  - e.g., a concurrent map or set
Implementing Concurrent & Non-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 source is partitioned into chunks
  - Each chunk runs in parallel & is collected into one concurrent mutable result container
    - e.g., a concurrent map or set

Different threads in a parallel stream share one concurrent result container
Implementing Concurrent & Non-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 source is partitioned into chunks.
- Each chunk runs in parallel & is collected into one concurrent mutable result container.

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

Of course, encounter order is not preserved..
A concurrent collector *may* perform better than a non-concurrent collector *if* merging costs are high.

See [github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex14](https://github.com/douglascraigschmidt/LiveLessons/tree/master/Java8/ex14)
Implementing Concurrent & Non-Concurrent Collectors

- A concurrent collector *may* perform better than a non-concurrent collector *if* merging costs are high
- Highly optimized result containers like ConcurrentHashMap may be more efficient than merging HashMaps
Implementing Concurrent & Non-Concurrent Collectors

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

Implementing Concurrent & Non-Concurrent Collectors

- The Collector interface defines three generic types

```java
interface Collector<T, A, R> {
    Supplier<A> supplier();
    BiConsumer<A, T> accumulator();
    BinaryOperator<A> combiner();
    Function<A, R> finisher();
    Set<Characteristics> characteristics();
}
```
Implementing Concurrent & Non-Concurrent Collectors

- The Collector interface defines three generic types
  - **T** – The type of objects available in the stream
    - e.g., Integer, String, etc.

<<Java Interface>>

```
Collector<T, A, R>
```

- supplier(): Supplier<A>
- accumulator(): BiConsumer<A, T>
- combiner(): BinaryOperator<A>
- finisher(): Function<A, R>
- characteristics(): Set<Characteristics>
Implementing Concurrent & Non-Concurrent Collectors

- The Collector interface defines three generic types
  - T
  - A – The type of a mutable accumulator object for collection
  - e.g., ConcurrentHashSet or List of T (implemented by ArrayList, LinkedList, etc.)

<<Java Interface>>

Collector<T, A, R>

- supplier(): Supplier<A>
- accumulator(): BiConsumer<A, T>
- combiner(): BinaryOperator<A>
- finisher(): Function<A, R>
- characteristics(): Set<Characteristics>

See Java8/ex14/src/main/java/utils/ConcurrentHashSet.java
Implementing Concurrent & Non-Concurrent Collectors

• The Collector interface defines three generic types
  • T
  • A
  • R – The type of a final result
  • e.g., ConcurrentHashSet or List of T
Implementing Concurrent & Non-Concurrent Collectors

- Five methods are defined in the Collector interface

<<Java Interface>>

```java
@interface Collector{
    Supplier<T> supplier();
    BiConsumer<A, T> accumulator();
    BinaryOperator<A> combiner();
    Function<A, R> finisher();
    Set<Characteristics> characteristics();
}
```
Implementing Concurrent & Non-Concurrent Collectors

- Five methods are defined in the Collector interface
  - `characteristics()` – provides a stream with additional information used for internal optimizations, e.g.
    - UNORDERED
      - The collector need not preserve the encounter order
- A concurrent collector *should* be UNORDERED, but a non-concurrent collector *can* be ORDERED
Implementing Concurrent & Non-Concurrent Collectors

- Five methods are defined in the Collector interface
  - `characteristics()` – provides a stream with additional information used for internal optimizations, e.g.
    - UNORDERED
    - IDENTIFY_FINISH
      - The finisher() is the identity function so it can be a no-op
        - e.g. finisher() just returns null

A concurrent collector *should* be IDENTIFY_FINISH, whereas a non-concurrent collector *could* be
Implementing Concurrent & Non-Concurrent Collectors

- Five methods are defined in the Collector interface
  - `characteristics()` – provides a stream with additional information used for internal optimizations, e.g.
    - UNORDERED
    - IDENTIFY_FINISH
    - CONCURRENT
    - `accumulator()` is called concurrently on result container

```
collector<T,A,R>
  supplier(): Supplier<A>
  accumulator(): BiConsumer<A,T>
  combiner(): BinaryOperator<A>
  finisher(): Function<A,R>
  characteristics(): Set<Characteristics>
```

The mutable result container must be synchronized!!

A concurrent collector *should* be CONCURRENT, but a non-concurrent collector should *not* be!
Implementing Concurrent & Non-Concurrent Collectors

- Five methods are defined in the Collector interface
  - `characteristics()` – provides a stream with additional information used for internal optimizations, e.g.
    - UNORDERED
    - IDENTIFY_FINISH
  - CONCURRENT
    - `accumulator()` is called concurrently on result container
    - The `combiner()` method is a no-op

<<Java Interface>>

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

- `supplier()`: Supplier<A>
- `accumulator()`: BiConsumer<A,T>
- `combiner()`: BinaryOperator<A>
- `finisher()`: Function<A,R>
- `characteristics()`: Set<Characteristics>
Five methods are defined in the Collector interface:

- **characteristics()** – provides a stream with additional information used for internal optimizations, e.g.
  - UNORDERED
  - IDENTIFY_FINISH
  - CONCURRENT
    - accumulator() is called concurrently on result container
    - The combiner() method is a no-op
    - A non-concurrent collector can be used with either sequential or parallel streams

Internally, the streams framework decides how to ensure correct behavior.
Implementing Concurrent & Non-Concurrent Collectors

- Five methods are defined in the Collector interface
  - `characteristics()` – provides a stream with additional information used for internal optimizations, e.g.

```java
Set characteristics() {
    return Collections.unmodifiableSet
        (EnumSet.of(Collector.Characteristics.CONCURRENT,
            Collector.Characteristics.UNORDERED,
            Collector.Characteristics.IDENTITY_FINISH));
}
```

Any/all characteristics can be set using `EnumSet.of()`

See [docs.oracle.com/javase/8/docs/api/java/util/EnumSet.html](http://docs.oracle.com/javase/8/docs/api/java/util/EnumSet.html)
Implementing Concurrent & Non-Concurrent Collectors

• Five methods are defined in the Collector interface
  - characteristics()
  - supplier() – returns a supplier that acts as a factory to generate an empty result container
Implementing Concurrent & Non-Concurrent Collectors

- Five methods are defined in the Collector interface
  - `characteristics()`
  - `supplier()` – returns a supplier that acts as a factory to generate an empty result container, e.g.
    - `return ArrayList::new`

A non-concurrent collector provides a result container for each thread in a parallel stream.
Implementing Concurrent & Non-Concurrent Collectors

- Five methods are defined in the Collector interface
  - `characteristics()`
  - `supplier()` — a factory that returns supplier instance that generates an empty result container, e.g.
    - `return ArrayList::new`
    - `return ConcurrentHashMap::new`

A concurrent collector has one result container shared by all threads in a parallel stream.
Five methods are defined in the Collector interface:

- `characteristics()`
- `supplier()`
- `accumulator()` – returns a bi-consumer that adds a new element to an existing result container

<<Java Interface>>

`Collector<T,A,R>`

- `supplier()`: `Supplier<A>`
- `accumulator()`: `BiConsumer<A,T>`
- `combiner()`: `BinaryOperator<A>`
- `finisher()`: `Function<A,R>`
- `characteristics()`: `Set<Characteristics>`
Implementing Concurrent & Non-Concurrent Collectors

- Five methods are defined in the Collector interface
  - `characteristics()`
  - `supplier()`
  - `accumulator()` – returns a bi-consumer that adds a new element to an existing result container, e.g.
    - `return List::add`

A non-concurrent collector needs no synchronization
Five methods are defined in the Collector interface:

- characteristics()

- supplier()

- **accumulator()** – returns a bi-consumer that adds a new element to result container, e.g.
  
  - return `List::add`
  
  - return `ConcurrentHashSet::add`

A non-concurrent collector must be synchronized.
Implementing Concurrent & Non-Concurrent Collectors

- Five methods are defined in the Collector interface
  - characteristics()
  - supplier()
  - accumulator()
  - combiner() – returns a binary operator that merges two result containers together

<<Java Interface>>

```java
interface Collector<T, A, R> {
    Supplier<A> supplier();
    BiConsumer<A, T> accumulator();
    BinaryOperator<A> combiner();
    Function<A, R> finisher();
    Set<Characteristics> characteristics();
}
```
Implementing Concurrent & Non-Concurrent Collectors

- Five methods are defined in the Collector interface
  - characteristics()
  - supplier()
  - accumulator()
  - combiner() – returns a binary operator that merges two result containers together, e.g.
    - return (one, another) -> {
      one.addAll(another); return one;
    }

A combiner() is only used for a non-concurrent collector
Implementing Concurrent & Non-Concurrent Collectors

- Five methods are defined in the Collector interface
  - characteristics()
  - supplier()
  - accumulator()
  - combiner() – returns a binary operator that merges two result containers together, e.g.
    - return (one, another) -> {
      one.addAll(another); return one;
    }
    - return null

The combiner() method is not called when CONCURRENT is set
Implementing Concurrent & Non-Concurrent Collectors

- Five methods are defined in the Collector interface
  - `characteristics()`
  - `supplier()`
  - `accumulator()`
  - `combiner()`
  - `finisher()` – returns a function that converts the result container to final result type
Five methods are defined in the Collector interface:

- `characteristics()`
- `supplier()`
- `accumulator()`
- `combiner()`
- `finisher()` – returns a function that converts the result container to final result type, e.g.:

  - `Function.identity()`
Implementing Concurrent & Non-Concurrent Collectors

• Five methods are defined in the Collector interface
  • characteristics()
  • supplier()
  • accumulator()
  • combiner()
  • **finisher()** – returns a function that converts the result container to final result type, e.g.
    • Function.identity()
    • return null

*Should be a no-op if IDENTITY_FINISH characteristic is set*
Implementing Concurrent & Non-Concurrent Collectors

- Five methods are defined in the Collector interface
  - characteristics()
  - supplier()
  - accumulator()
  - combiner()
  - finisher() – returns a function that converts the result container to final result type, e.g.
    - Function.identity()
    - return null

```
Stream
  .generate(() ->
    makeBigFraction
    (new Random(), false))
  .limit(sMAX_FRACTIONS)
  .map(reduceAndMultiplyFraction)
  .collect(FuturesCollector
    .toFuture())
  .thenAccept
    (this::sortAndPrintList);
```

finisher() can also be much more interesting!

See Java8/ex19/src/main/java/utils/FuturesCollector.java
End of Java 8 Parallel Stream Internals (Part 6)