Java Streams, Part 3: Streams under the hood

Understand java.util.stream internals

Brian Goetz

July 06, 2016
(First published May 09, 2016)

Explore the Java™ Streams library, introduced in Java SE 8, in this series by Java Language Architect Brian Goetz. By taking advantage of the power of lambda expressions, the `java.util.stream` package makes it easy to run functional-style queries on collections, arrays, and other data sets. In this installment, learn how to fine-tune your queries for maximum efficiency.

View more content in this series

The first two articles in this series explore how to use the `java.util.stream` library added in Java SE 8, which makes it easy to express a query on a data set declaratively. In many cases, the library figures out how to perform queries efficiently, with no help from the user. But when performance is critical, it's valuable to understand how the library works internally, so that you can eliminate possible sources of inefficiency. This third installment explores how the implementation of Streams works and explains some of the optimizations that the declarative approach makes possible.

Stream pipelines

A *stream pipeline* is composed of a *stream source*, zero or more *intermediate operations*, and a *terminal operation*. Stream sources can be collections, arrays, generator functions, or any other data source that can suitably provide access to its elements. Intermediate operations transform streams into other streams — by filtering the elements (`filter()`), transforming the elements (`map()`), sorting the elements (`sorted()`), truncating the stream to a certain size (`limit()`), and so on. Terminal operations include aggregations (`reduce()`, `collect()`), searching (`findFirst()`), and iteration (`forEach()`).

About this series
With the `java.util.stream` package, you can concisely and declaratively express possibly-parallel bulk operations on collections, arrays, and other data sources. In this series by Java Language Architect Brian Goetz, get a comprehensive understanding of the Streams library and learn how to use it to best advantage.

Stream pipelines are constructed lazily. Constructing a stream source doesn't compute the elements of the stream, but instead captures how to find the elements when necessary. Similarly, invoking an intermediate operation doesn't perform any computation on the elements; it merely adds another operation to the end of the stream description. Only when the terminal operation is invoked does the pipeline actually perform the work — compute the elements, apply the intermediate operations, and apply the terminal operation. This approach to execution makes several interesting optimizations possible.

Stream sources

A stream source is described by an abstraction called `Spliterator`. As its name suggests, `Spliterator` combines two behaviors: accessing the elements of the source (iterating), and possibly decomposing the input source for parallel execution (splitting).

Although `Spliterator` includes the same basic behaviors as `Iterator`, it doesn't extend `Iterator`, instead taking a different approach to element access. An `Iterator` has two methods, `hasNext()` and `next()`; accessing the next element can involve (but doesn't require) calling both of these methods. As a result, coding an `Iterator` correctly requires a certain amount of defensive and duplicative coding. (What if the client doesn't call `hasNext()` before `next()`? What if it calls `hasNext()` twice?) Additionally, the two-method protocol generally requires a fair amount of statefulness, such as peeking ahead one element (and keeping track of whether you've already peeked ahead). Together, these requirements add up to a fair degree of per-element access overhead.

Having lambdas in the language enables `Spliterator` to take an approach to element access that's generally more efficient — and easier to code correctly. `Spliterator` has two methods for accessing elements:

```
boolean tryAdvance(Consumer<? super T> action);
void forEachRemaining(Consumer<? super T> action);
```

The `tryAdvance()` method tries to process a single element. If no elements remain, `tryAdvance()` merely returns `false`; otherwise, it advances the cursor and passes the current element to the provided handler and returns `true`. The `forEachRemaining()` method processes all the remaining elements, passing them one at a time to the provided handler.

Even ignoring the possibility of parallel decomposition, the `Spliterator` abstraction is already a "better iterator" — simpler to write, simpler to use, and generally having lower per-element access overhead. But the `Spliterator` abstraction also extends to parallel decomposition. A spliterator describes a sequence of remaining elements; calling the `tryAdvance()` or `forEachRemaining()` element-access method advances through that sequence. To split the source, so that two threads can work separately on different sections of the input, `Spliterator` provides a `trySplit()` method:

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

The behavior of `trySplit()` is to try to split the remaining elements into two sections, ideally of similar size. If the `Spliterator` can be split, `trySplit()` slices off an initial portion of the described elements into a new `Spliterator`, which is returned, and adjusts its state to describe the elements following the sliced-off portion. If the source can't be split, `trySplit()` returns `null`, indicating that the splitting isn't possible and that the caller should proceed sequentially. For sources whose encounter order is significant (for example, arrays, `List`, or `SortedSet`), `trySplit()` must preserve this order; it must split off the initial portion of the remaining elements into the new `Spliterator`, and the current spliterator must describe the remaining elements in an order consistent with the original ordering.

The `Collection` implementations in the JDK have all been furnished with high-quality `Spliterator` implementations. Some sources admit better implementations than others: an `ArrayList` with more than one element can always be split cleanly and evenly; a `LinkedList` always splits poorly; and hash-based and tree-based sets can generally be split reasonably well.

Building a stream pipeline

A stream pipeline is built by constructing a linked-list representation of the stream source and its intermediate operations. In the internal representation, each stage of the pipeline is described by a bitmap of *stream flags* that describe what's known about the elements at this stage of the stream pipeline. Streams uses these flags to optimize both the construction and execution of the stream. Table 1 shows the stream flags and their interpretations.

Table 1. Stream flags

Stream flag	Interpretation
`SIZED`	The size of the stream is known.
`DISTINCT`	The elements of the stream are distinct, according to `Object.equals()` for object streams, or according to `==` for primitive streams.
`SORTED`	The elements of the stream are sorted in the natural order.
`ORDERED`	The stream has a meaningful encounter order (see the "Encounter order" section).

The stream flags for the source stage are derived from the `characteristics` bitmap of the spliterator (spliterators support a larger set of flags than do streams). A high-quality spliterator implementation not only provides efficient element access and splitting but also describes the characteristics of the elements. (For example, the spliterator for a `HashSet` reports the `DISTINCT` characteristic, since the elements of a `Set` are known to be distinct.)

> *“ In some cases, Streams can use knowledge of the source and preceding operations to elide an operation entirely. ”*

Each intermediate operation has a known effect on the stream flags; an operation can set, clear, or preserve the setting for each flag. For example, the `filter()` operation preserves the `SORTED`

and `DISTINCT` flags but clears the `SIZED` flag; the `map()` operation clears the `SORTED` and `DISTINCT` flags but preserves the `SIZED` flag; and the `sorted()` operation preserves the `SIZED` and `DISTINCT` flags and injects the `SORTED` flag. As the linked-list representation of stages is constructed, the flags for the previous stage are combined with the behavior of the current stage to arrive at a new set of flags for the current stage.

In some cases, the flags make it possible to elide an operation entirely, as in the stream pipeline in Listing 1.

Listing 1. Stream pipeline in which operations can be automatically elided

```
TreeSet<String> ts = ...
String[] sortedAWords = ts.stream()
                          .filter(s -> s.startsWith("a"))
                          .sorted()
                          .toArray();
```

The stream flags for the source stage include `SORTED`, because the source is a `TreeSet`. The `filter()` method preserves the `SORTED` flag, so the stream flags for the filtering stage also include the `SORTED` flag. Normally, the result of the `sorted()` method would be to construct a new pipeline stage, add it to the end of the pipeline, and return the new stage. However, because it's known that the elements are already sorted in natural order, the `sorted()` method is a no-op — it just returns the previous stage (the filtering stage), since sorting would be redundant. (Similarly, if the elements are known to be `DISTINCT`, the `distinct()` operation can be eliminated entirely.)

Executing a stream pipeline

When the terminal operation is initiated, the stream implementation picks an execution plan. Intermediate operations are divided into *stateless* (`filter()`, `map()`, `flatMap()`) and *stateful* (`sorted()`, `limit()`, `distinct()`) operations. A stateless operation is one that can be performed on an element without knowledge of any of the other elements. For example, a filtering operation only needs to examine the current element to determine whether to include or eliminate it, but a sorting operation must see all the elements before it knows which element to emit first.

If the pipeline is executing sequentially, or is executing in parallel but consists of all stateless operations, it can be computed in a single pass. Otherwise, the pipeline is divided into sections (at stateful operation boundaries) and is computed in multiple passes.

Terminal operations are either *short-circuiting* (`allMatch()`, `findFirst()`) or *non–short-circuiting* (`reduce()`, `collect()`, `forEach()`). If the terminal operation is non–short-circuiting, the data can be processed in bulk (using the `forEachRemaining()` method of the source spliterator, further reducing the overhead of accessing each element); if it's short-circuiting, it must be processed one element at a time (using `tryAdvance()`).

For sequential execution, Streams constructs a "machine" — a chain of `Consumer` objects whose structure matches that of the pipeline structure. Each of these `Consumer` objects knows about the next stage; when it receives an element (or is notified that there are no more elements), it sends zero or more elements to the next stage in the chain. For example, the `Consumer` associated with a `filter()` stage applies the filter predicate to the input element and either does or doesn't

send it on to the next stage; the `Consumer` associated with a `map()` stage applies the mapping function to the input element and sends the result to the next stage. The `Consumer` associated with a stateful operation such as `sorted()` buffers elements until it sees the end of the input, and then it sends the sorted data to the next stage. The final stage in the machine implements the terminal operation. If this operation produces a result, such as `reduce()` or `toArray()`, this stage acts as accumulator for the result.

Figure 1 shows an animation (or, in certain browsers, a snapshot) of the "stream machine" for the following stream pipeline. (In Figure 1, yellow, green, and blue blocks enter the machine's first stage from the top, in sequence. In the first stage, each block is compressed into a smaller block and then falls into the second stage. There, a Pacman-like character swallows each yellow block, letting only the green and blue blocks fall into the third stage. Compressed blue and green blocks are alternately displayed on a computer screen.)

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

Figure 1. The stream machine (animations courtesy of Tagir Valeev)

Parallel execution does something similar, except that instead of creating a single machine, each worker thread gets its own copy of the machine and feeds its section of the data to it, and then the result of each per-thread machine is merged with the results of other machines to produce a final result.

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.

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

The presizing optimization is even more effective in parallel stream executions. In addition to the `SIZED` flag, another spliterator characteristic, `SUBSIZED`, means that not only is the size known, but that if the spliterator is split, the split sizes will be also known. (This is true for arrays and `ArrayList`, but not necessarily true for other splittable sources such as trees.) If the `SUBSIZED` characteristic is present, in a parallel execution the `toArray()` operation can allocate a single correct-sized array for the entire result, and individual threads (each working on a separate section of the input) can write their results directly into the correct section of the array — with no synchronization or copying needed. (In the absence of the `SUBSIZED` flag, each section is collected to an intermediate array and then copied to the final location.)

Encounter order

Another subtle consideration that influences the library's ability to optimize is *encounter order*. Encounter order refers to whether or not the order in which a source dispenses elements is significant to the computation. Some sources (such as hash-based sets and maps) have no meaningful encounter order. A stream flag, `ORDERED`, describes whether the stream has a meaningful encounter order or not. The spliterators for the JDK collections set this flag based on the specification of the collection; some intermediate operations might inject `ORDERED` (`sorted()`) or clear it (`unordered()`).

If the stream does have an encounter order, most stream operations must respect that order. For sequential executions, preserving encounter order is essentially free, because elements are naturally processed in the order in which they're encountered. Even in parallel, for many operations (stateless intermediate operations and certain terminal operations such as `reduce()`), respecting the encounter order doesn't impose any real costs. But for others (stateful intermediate operations, and terminal operations whose semantics are tied to encounter order, such as `findFirst()` or `forEachOrdered()`), the obligation to respect the encounter order in a parallel execution can be significant. If the stream has a defined encounter order, but that order isn't significant to the result, it might be possible to speed up parallel execution of pipelines containing order-sensitive operations by removing the `ORDERED` flag with the `unordered()` operation.

As an example of an operation that's sensitive to encounter order, consider `limit()`, which truncates a stream at a specified size. Implementing `limit()` in a sequential execution is trivial: Keep a counter of how many elements have been seen, and discard any elements after that. But in a parallel execution, implementing `limit()` is much more complicated; you have to keep the **first** `N` elements. This requirement greatly constrains the ability to exploit parallelism; if the input is divided into sections, you don't know if the result of a section will be included in the final result until all the sections preceding that section have been completed. As a result, the implementation generally has the bad choice of not using all the cores that are available, or buffering the entire tentative result until you hit the target length.

If the stream has no encounter order, the `limit()` operation is free to choose **any** `N` elements, which admits a much more efficient execution. Elements can be sent downstream as soon as they're known, without any buffering, and the only coordination needed between threads is a semaphore to ensure that the target stream length isn't exceeded.

Another, more subtle example of the costs of encounter order is sorting. If encounter order is significant, the `sorted()` operation implements a *stable* sort (equal elements appear in the same order in the output as they do in the input), whereas for an unordered stream, stability — which has a cost — isn't required. A similar story exists for `distinct()`: If the stream has an encounter order, then for multiple equal input elements, `distinct()` must emit the **first** of them, whereas for an unordered stream, it can emit any of them — which again admits a much more efficient parallel implementation.

A similar situation arises when you aggregate with `collect()`. If you execute a `collect(groupingBy())` operation on an ordered stream, the elements corresponding to any key must be presented to the downstream collector in the order in which they appear in the input. Often, this order isn't significant to the application, and any order would do. In these cases, it might be preferable to select a *concurrent* collector (such as `groupingByConcurrent()`), which is allowed to ignore encounter order and let all threads collect directly into a shared concurrent data structure (such as `ConcurrentHashMap`) rather than having each thread collecting into its own intermediate map, and then merging the intermediate maps (which can be expensive).

Creating streams

> *“ It's easy to adapt existing data structures to dispense streams. ”*

While many of the classes in the JDK have been retrofitted to serve as stream sources, it's also easy to adapt existing data structures to dispense streams. To create a stream from an arbitrary data source, you need to create a `Spliterator` for the stream's elements and pass the spliterator to `StreamSupport.stream()`, along with a `boolean` flag indicating whether the resulting stream should be sequential or parallel.

`Spliterator` implementations can range considerably in quality, making trade-offs between the effort of implementation and the performance of stream pipelines that use spliterators as a source. The `Spliterator` interface has several methods that are essentially optional, such as `trySplit()`. If you don't want to implement splitting, you can always return `null` from `trySplit()`— but this means that streams using this `Spliterator` as a source will be unable to exploit parallelism to speed up the computation.

Considerations that affect the quality of a spliterator include:

- Does the spliterator report an accurate size?
- Can the spliterator split the input at all?
- Can it split the input into roughly equal sections?
- Are the sizes of the splits predictable (reflected through the `SUBSIZED` characteristics)?

- Does the spliterator report all relevant characteristics?

The easiest way to make a spliterator, but which results in the worst-quality result, is to pass an `Iterator` to `Spliterators.spliteratorUnknownSize()`. You can obtain a slightly better spliterator by passing an `Iterator` and a size to `Spliterators.spliterator`. But if stream performance is important — especially, parallel performance — implement the full `Spliterator` interface, including all applicable characteristics. The JDK sources for collection classes such as `ArrayList`, `TreeSet`, and `HashMap` provide examples of high-quality spliterators that you can emulate for your own data structures.

Conclusion to Part 3

While the performance of Streams out of the box is generally good (sometimes even better than the corresponding imperative code), having a firm grasp on how Streams works under the hood enables you to use the library with maximum efficiency, and to create custom adapters for deriving a stream from any data source. The next two *Java Streams* series installments explore parallelism in depth.

Related topics

- Package documentation for java.util.stream
- Functional Programming in Java: Harnessing the Power of Lambda Expressions (Venkat Subramaniam, Pragmatic Bookshelf, 2014)
- Mastering Lambdas: Java Programming in a Multicore World (Maurice Naftalin, McGraw-Hill Education, 2014)
- Should I return a Collection or a Stream?
- RxJava library
- IBM Code: Java journeys