Streams in Java

ANJU MUNOTH

Streams

- java.util.stream library provides a simple and flexible means to express possibly-parallel or sequential functional-style queries on various data sources, including collections, arrays, generator functions, ranges, or custom data structures.

Querying with streams

- Uses of streams is to represent *queries* over data in collections.
- Code shows an example of a simple stream pipeline.
- Pipeline takes a collection of transactions modeling a purchase between a buyer and a seller, and computes the total dollar value of transactions by sellers living in New York.

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

- filter() operation selects only transactions with sellers from New York.
- mapToInt() operation selects the transaction amount for the desired transactions.
- And the terminal sum() operation adds up these amounts

Streams vs for loop

- Imperative (for-loop) version of this query is also simple and takes fewer lines of code to express.
- **Why streams**
- Streams exploit that most powerful of computing principles: composition.
- By composing complex operations out of simple building blocks (filtering, mapping, sorting, aggregation), streams queries are more likely to remain straightforward to write and read as the problem gets complicated than are more ad-hoc computations on the same data sources.

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

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

Working with for loop with collections

- Imperative version of queries resort to materializing collections for the results of intermediate calculations, such as the result of filtering or mapping.
- Not only can these results clutter the code, but they also clutter the execution.
- Materialization of intermediate collections serves only the implementation, not the result, and it consumes compute cycles organizing intermediate results into data structures that will only be discarded.

Stream pipelines

- Stream pipelines fuse their operations into as few passes on the data as possible, often a single pass.
- Stateful intermediate operations, such as sorting, can introduce barrier points that necessitate multipass execution.
- Each stage of a stream pipeline produces its elements lazily, computing elements only as needed, and feeds them directly to the next stage.
- Don't need a collection to hold the intermediate result of filtering or mapping, so you save the effort of populating (and garbage-collecting) the intermediate collections.
- Also, following a "depth first" rather than "breadth first" execution strategy (tracing the path of a single data element through the entire pipeline) causes the data being operated upon to more often be "hot" in cache, so can spend more time computing and less time waiting for data.

Streams over for loop

- Far easier to read, because the user is neither distracted with "garbage" variables--like sellers and sorted—
- Doesn't have to keep track of a lot of context while reading the code;
- the code reads almost exactly like the problem statement.
- Code that's more readable is also less error prone, because maintainers are more likely to be able to correctly discern at first glance what the code does.

Design Approach of streams

- Practical separation of concerns.
- Client is in charge of specifying the "what" of a computation, but the library has control over the "how."
- This separation tends to parallel the distribution of expertise; the client writer generally has better understanding of the problem domain, whereas the library writer generally has more expertise in the algorithmic properties of the execution.
- Key enabler for writing libraries that allow this sort of separation of concerns is the ability to pass behavior as easily as passing data, which in turn enables APIs where callers can describe the structure of a complex calculation, and then get out of the way while the library chooses the execution strategy.

Anatomy of a stream pipeline

- All stream computations share a common structure:
- **Have a stream source, zero or more intermediate operations, and a single terminal operation.**
- Elements of a stream can be object references (Stream<String>) or they can be primitive integers (IntStream), longs (LongStream), or doubles (DoubleStream).
- Because most of the data that Java programs consume is already stored in collections, many stream computations use collections as their source.
- Collection implementations in the JDK have all been enhanced to act as efficient stream sources.
- But other possible stream sources also exist--such as arrays, generator functions, or built-in factories such as numeric ranges-- it's possible to write custom stream adapters so that any data source can act as a stream source

stream-producing methods

Method	Description
Collection.stream()	Create a stream from the elements of a collection.
Stream.of(T...)	Create a stream from the arguments passed to the factory method.
Stream.of(T[])	Create a stream from the elements of an array.
Stream.empty()	Create an empty stream.
Stream.iterate(T first, BinaryOperator<T> f)	Create an infinite stream consisting of the sequence first, f(first), f(f(first)), ...
Stream.iterate(T first, Predicate<T> test, BinaryOperator<T> f)	(Java 9 only) Similar to Stream.iterate(T first, BinaryOperator f), except the stream terminates on the first elements for which the test predicate returns false.

stream-producing methods

Method	Description
Stream.generate(Supplier<T> f)	Create an infinite stream from a generator function.
IntStream.range(lower, upper)	Create an IntStream consisting of the elements from lower to upper, exclusive.
IntStream.rangeClosed(lower, upper)	Create an IntStream consisting of the elements from lower to upper, inclusive.
BufferedReader.lines()	Create a stream consisting of the lines from a BufferedReader.
BitSet.stream()	Create an IntStream consisting of the indexes of the set bits in a BitSet.
CharSequence.chars()	Create an IntStream corresponding to the chars in a String.

Intermediate operations

- Are always lazy
- Invoking an intermediate operation merely sets up the next stage in the stream pipeline but doesn't initiate any work.
- Intermediate operations are further divided into stateless and stateful operations.
- **Stateless** operations (such as filter() or map()) can operate on each element independently,
- **Stateful** operations (such as sorted() or distinct()) can incorporate state from previously seen elements that affects the processing of other elements.

Intermediate operations

- filter() (selecting elements matching a criterion),
- map() (transforming elements according to a function),
- distinct() (removing duplicates),
- limit() (truncating a stream at a specific size),
- sorted()— transform a stream into another stream.
- Some operations, such as mapToInt(), take a stream of one type and return a stream of a different type;

Intermediate operations

Operation	Contents
filter(Predicate<T>)	The elements of the stream matching the predicate
map(Function<T, U>)	The result of applying the provided function to the elements of the stream
flatMap(Function<T, Stream<U>)	The elements of the streams resulting from applying the provided stream-bearing function to the elements of the stream
distinct()	The elements of the stream, with duplicates removed
sorted()	The elements of the stream, sorted in natural order
Sorted(Comparator<T>)	The elements of the stream, sorted by the provided comparator
limit(long)	The elements of the stream, truncated to the provided length
skip(long)	The elements of the stream, discarding the first N elements

Terminal stream operations

- Processing of the data set begins when a terminal operation is executed, such as a reduction (sum() or max()), application (forEach()), or search (findFirst()) operation.
- Terminal operations produce a result or a side effect.
- When a terminal operation is executed, the stream pipeline is terminated, and if you want to traverse the same data set again, you can set up a new stream pipeline.

Terminal stream operations

Operation	Description
forEach(Consumer<T> action)	Apply the provided action to each element of the stream.
toArray()	Create an array from the elements of the stream.
reduce(...)	Aggregate the elements of the stream into a summary value.
collect(...)	Aggregate the elements of the stream into a summary result container.
min(Comparator<T>)	Return the minimal element of the stream according to the comparator.
max(Comparator<T>)	Return the maximal element of the stream according to the comparator.
count()	Return the size of the stream.
{any,all,none}Match(Predicate<T>)	Return whether any/all/none of the elements of the stream match the provided predicate.
findFirst()	Return the first element of the stream, if present.
findAny()	Return any element of the stream, if present.

Streams versus collections

- While streams can resemble collections superficially;Both contain data—But in reality they differ

Collection	Streams
• Collection is a data structure; • Main concern is the organization of data in memory, • Collection persists over a period of time. • Collection might often be used as the source or target for a stream pipeline, but a stream's focus is on computation, not data.	• Data in a stream comes from elsewhere (a collection, array, generator function, or I/O channel) • Data is processed through a pipeline of computational steps to produce a result or side effect, at which point the stream is finished. • Streams provide no storage for the elements that they process, and the lifecycle of a stream is more like a point in time--the invocation of the terminal operation. • Unlike collections, streams can also be infinite; correspondingly, some operations (limit(), findFirst()) are short-circuiting and can operate on infinite streams with finite computation.

Streams versus collections

- Operations on collections are eager and mutative;
- When the remove() method is called on a List, after the call returns, you know that the list state was modified to reflect the removal of the specified element.
- For streams, only the terminal operation is eager; the others are lazy.
- Stream operations represent a functional transformation on their input (also a stream), rather than a mutative operation on a data set (filtering a stream produces a new stream whose elements are a subset of the input stream but doesn't remove any elements from the source).

Stream pipeline

- Expressing a stream pipeline as a sequence of functional transformations enables several useful execution strategies,
- ***Laziness*, *short circuiting*, and *operation fusion*.**
- **Short-circuiting** enables a pipeline to terminate successfully without examining all the data; queries such as "find the first transaction over $1,000" needn't examine any more transactions after a match is found.
- **Operation fusion** means that multiple operations can be executed in a single pass on the data;
  - Operations are combined into a single pass on the data--rather than first selecting all the matching transactions, then selecting all the corresponding amounts, and then adding them up.

Streams and lambdas

- Streams library is orchestrating the computation,
- But performing the computation involves callbacks to lambdas provided by the client, what those lambda expressions can do is subject to certain contraints.
- Violating these constraints could cause the stream pipeline to fail or compute an incorrect result.
- Additionally, for lambdas with side effects, the timing (or existence) of these side effects might be surprising in some cases.

Streams and lambdas

- Most stream operations require that the lambdas passed to them be non-interfering and stateless.
- **Non-interfering** --Won't modify the stream source;
- **Stateless --**Won't access (read or write) any state that might change during the lifetime of the stream operation.
- For reduction operations (for example, computing summary data such as sum, min, or max) the lambdas passed to these operations must be **associative** (or conform to similar requirements).
- Stream library might, if the pipeline executes in parallel, access the data source or invoke these lambdas concurrently from multiple threads

Lambda -- modifying the stream source

- Root of all concurrency risks is shared mutable state.
- One possible source of shared mutable state is the stream source.
- If the source is a traditional collection like ArrayList, the Streams library assumes that it remains unmodified during the course of a stream operation.
- Collections explicitly designed for concurrent access, such as ConcurrentHashMap, are exempt from this assumption.
- Not only does the noninterference requirement exclude the source being mutated by other threads during a stream operation, but the lambdas passed to stream operations themselves should also refrain from mutating the source.

lambdas passed to stream operations-- stateless

- Tries to eliminate any element that's twice some preceding element, violates this rule -- so, **don't** do this!

```
HashSet<Integer> twiceSeen = new HashSet<>();
int[] result = elements.stream()
.filter(e -> {
    twiceSeen.add(e * 2);
    return twiceSeen.contains(e);
}) .toArray();
```

- If executed in parallel, this pipeline would produce incorrect results.
- First, access to the twiceSeen set is done from multiple threads without any coordination and therefore isn't thread safe.
- Second, because the data is partitioned, there's no guarantee that when a given element is processed, all elements preceding that element were already processed.

Lambdas – side effect free

- Lambdas passed to stream operations are entirely side effect free— that is, that they don't mutate any heap-based state or perform any I/O during their execution.
- If they do have side effects, it's their responsibility to provide any required coordination to ensure that such side effects are thread safe.

Stream pipeline Optimisation

```
int count = anArrayList.stream()
    .map(e -> { System.out.println("Saw " + e); e })
    .count();
```

- Not even guaranteed that all side effects will be executed.
- Library is free to avoid executing the lambda passed to map() entirely.
- Because the source has a known size, the map() operation is known to be size preserving, and the mapping doesn't affect the result of the computation, the library can optimize the calculation by not performing the mapping at all!
- This optimization can turn the computation from O(n) to O(1), in addition to eliminating the work associated with invoking the mapping function

Stream pipelines

- 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.
- 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.

Iterator

- 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.

Spliterator

- 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:
- **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.
- **forEachRemaining**() method processes all the remaining elements, passing them one at a time to the provided handler.

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

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

Spliterator

- For parallel decomposition, the Spliterator 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();

- trySplit()--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.