Stream Gatherers

Stream gatherers enable you to create custom intermediate operations, which enables stream pipelines to transform data in ways that aren't easily achievable with existing built-in intermediate operations.

> **Note:** This is a preview feature. A preview feature is a feature whose design, specification, and implementation are complete, but is not permanent. A preview feature may exist in a different form or not at all in future Java SE releases. To compile and run code that contains preview features, you must specify additional command-line options. See Preview Language and VM Features.
> For background information about stream gatherers, see JEP 461.

Topics

What Is a Gatherer?

A gatherer is an intermediate operation that transforms a stream of input elements into a stream of output elements, optionally applying a final action when it reaches the end of the stream of input elements.

Remember that an intermediate operation, such as `Stream.map(Function)`, produces a new stream, while a terminal operation, such as `Stream.forEach(Consumer)`, produces a non-stream result. A non-stream result could be a primitive value (like a double value), a collection, or in the case of `forEach`, no value at all.

Gatherers can do the following:

- Transform elements in a one-to-one, one-to-many, many-to-one, or many-to-many fashion
- Track previously seen elements to influence the transformation of later elements
- Short-circuit, or stop processing input elements to transform infinite streams to finite ones
- Process a stream in parallel

> **Note:** A gather will process a stream in parallel only if you specify a combiner function when you create the gatherer. See The Combiner Function in Creating a Gatherer. A gatherer's default combiner turns parallelization off even if you call `parallel()`.

Examples of gathering operations include the following:

- Grouping elements into batches
- Deduplicating consecutively similar elements
- Incremental accumulation functions
- Incremental reordering functions

Creating a Gatherer

To create a gatherer, implement the `Gatherer` interface.

The following example creates a gatherer that returns the largest integer from a stream of integers. However, if the gatherer encounters an integer equal or larger to its argument `limit`, then it returns that integer and stops processing the stream's integers.

```
record BiggestInt(int limit) implements Gatherer<Integer, List<Integer>, Integer> {

    // The initializer creates a new private ArrayList to keep track of the
    // largest integer across elements.
    @Override
    public Supplier<List<Integer>> initializer() {
        return () -> new ArrayList<Integer>(1);
    }

    // The integrator

    @Override
    public Integrator<List<Integer>, Integer, Integer> integrator() {
        return Integrator.of(
            (max, element, downstream) -> {

                // Save the integer if it's the largest so far.
                if (max.isEmpty()) max.addFirst(element);
                else if (element > max.getFirst()) max.set(0, element);

                // If the integer is equal or greater to the limit,
                // "short-circuit": emit the current integer downstream
                // and return false to stop processing stream elements
                if (element >= limit) {
                    downstream.push(element);
                    return false;
                }

                // Return true to continue processing stream elements
                return true;
            }
        );
    }

    // The combiner, which is used during parallel evaluation

    @Override
    public BinaryOperator<List<Integer>> combiner() {
        return (leftMax, rightMax) -> {

            // If either the "left" or "right" ArrayLists contain
            // no value, then return the other
            if (leftMax.isEmpty()) return rightMax;
            if (rightMax.isEmpty()) return leftMax;

            // Return the ArrayList that contains the larger integer
            int leftVal = leftMax.getFirst();
            int rightVal = rightMax.getFirst();
            if (leftVal > rightVal) return leftMax;
            else return rightMax;
        };
    }

    @Override
    public BiConsumer<List<Integer>, Downstream<? super Integer>> finisher() {

        // Emit the largest integer, if there is one, downstream
        return (max, downstream) -> {
            if (!max.isEmpty()) {
                downstream.push(max.getFirst());
            }
        };
    }
}
```

You can use this gather as follows:

```
System.out.println(Stream.of(5,4,2,1,6,12,8,9)
                         .gather(new BiggestInt(11))
                         .findFirst()
                         .get());
```

It prints the following output:

```
12
```

You can also use this gatherer in parallel:

```
System.out.println(Stream.of(5,4,2,1,6,12,8,9)
                         .gather(new BiggestInt(11))
                         .parallel()
                         .findFirst()
                         .get());
```

The `Gatherer<T,A,R>` interface has three type parameters:

- `T`: The type of input elements to the gather operation. This example process a stream of `Integer` elements.
- `A`: The type of the gatherer's private state object, which the gatherer can use to track previously seen elements to influence the transformation of later elements. This example uses a `List<Integer>` to store the largest `Integer` it has encountered so far in the input stream.
- `R`: The type of output elements from the gatherer operation. This example returns an `Integer` value.

You create a gatherer by defining four functions that work together that process input elements. Some of these functions are optional depending on your gatherer's operation:

- `initializer()`: Creates the gatherer's private state object
- `integrator()`: Integrates a new element from the input stream, possibly inspects the private state object, and possibly emits elements to the output stream
- `combiner()`: Combines two private state objects into one when the gatherer is processing the stream in parallel
- `finisher()`: Optionally performs an action after the gatherer has processed all input elements; it could inspect the private state object or emit additional output elements

The Initializer Function

The optional initializer function creates the gatherer's private state object. This example creates an empty `ArrayList` with a capacity of only one `Integer` as its meant to store the largest `Integer` the gatherer has encountered so far.

```
@Override
public Supplier<List<Integer>> initializer() {
    return () -> new ArrayList<Integer>(1);
}
```

The Integrator Function

Every gatherer requires an integrator function. To create an integrator function, call either `Gatherer.Integrator.of(Gatherer.Integrator)` or `Gatherer.Integrator.ofGreedy(Gatherer.Integrator)`. These methods take as an argument a lambda expression that contains three parameters. This example uses the following lambda expression:

```
(max, element, downstream) -> {

            // Save the integer if it's the largest so far.
            if (max.isEmpty()) max.addFirst(element);
            else if (element > max.getFirst()) max.set(0, element);

            // If the integer is equal or greater to the limit,
            // "short-circuit": emit the current integer downstream
            // and return false to stop processing stream elements
            if (element >= limit) {
                downstream.push(element);
                return false;
            }

            // Return true to continue processing stream elements
            return true;
        }
```

The parameter `max` is the private state object.

The parameter `element` is the input element that the integrator function is currently processing.

The parameter `downstream` is a `Gatherer.Downstream` object. When you call its `push` method, it passes its argument to the next stage in the pipeline.

An integrator function returns a `boolean` value. If it returns `true`, then it will process the next element of the input stream. If it returns `false`, then it will short-circuit and stop processing input elements.

> **Tip:** The `Downstream::push` method returns `true` if the downstream is willing to push additional elements, so your integrator function can return its return value if you want to continue processing stream elements.

In this example, if `element` is equal or greather than `limit`, the integrator function passes `element` to the next stage in the pipeline, then returns `false`. The integrator won't process any more input elements, and the `Downstream` object can no longer push values.

> **Note:** If you don't expect your integrator function to short-circuit and you want it to process all elements of your input stream, use `Integrator::ofGreedy` instead of `Integrator::of`.

The Combiner Function

The optional combiner function is called only if you're running the gatherer in parallel. The combiner function is a lambda expression that contains two parameters, which represent two private state objects.

```
@Override
public BinaryOperator<List<Integer>> combiner() {
    return (leftMax, rightMax) -> {

        // If either the "left" or "right" ArrayLists contain
        // no value, then return the other
        if (leftMax.isEmpty()) return rightMax;
        if (rightMax.isEmpty()) return leftMax;

        // Return the ArrayList that contains the larger integer
        int leftVal = leftMax.getFirst();
        int rightVal = rightMax.getFirst();
        if (leftVal > rightVal) return leftMax;
        else return rightMax;
    };
}
```

This example returns the private state object (an `ArrayList`) that contains the largest integer.

The Finisher Function

The optional finisher function is a lambda expression that contains two parameters:

```
@Override
public BiConsumer<List<Integer>, Downstream<? super Integer>> finisher() {

    // Emit the largest integer, if there is one, downstream
    return (max, downstream) -> {
        if (!max.isEmpty()) {
            downstream.push(max.getFirst());
        }
    };
}
```

The parameter `max` is the private state object and `downstream` is a `Gatherer.Downstream` object.

In this example, the finisher function pushes the value contained in the private state object. Note that this value won't be pushed if the integrator function returned `false`. You can check whether a `Downstream` object is no longer processing input elements by calling the method `Gatherer.Downstream::isRejecting`. If it returns `true`, it's no longer processing input elements.

> **Note:** If the finisher function pushes a value downstream, then that value is contained in an `Optional` object.

Creating Gatherers with Factory Methods

Instead of implementing the `Gatherer` interface, you can call one of the factory methods in the `Gatherer` interface to create a gatherer.

The following example is the same one as described in Creating a Gatherer except it calls the `Gatherer::of` method:

```
static Gatherer<Integer, List<Integer>, Integer> biggestInt(int limit) {

    return Gatherer.of(

        // Supplier

        () -> { return new ArrayList<Integer>(1); },

        // Integrator

        Gatherer.Integrator.of(
            (max, element, downstream) -> {
                System.out.println("Processing " + element);
                if (max.isEmpty()) max.addFirst(element);
                else if (element > max.getFirst()) max.set(0, element);

                if (element >= limit) {
                    downstream.push(element);
                    return false;
                }
                return true;
            }
        ),

        // Combiner

        (leftMax, rightMax) -> {
            if (leftMax.isEmpty()) return rightMax;
            if (rightMax.isEmpty()) return leftMax;
            int leftVal = leftMax.getFirst();
            int rightVal = rightMax.getFirst();
            if (leftVal > rightVal) return leftMax;
            else return rightMax;
        },

        // Finisher

        (max, downstream) -> {
            if (!max.isEmpty()) {
                downstream.push(max.getFirst());
            }
        }
    );
}
```

You can call this gatherer as follows:

```
System.out.println(Stream.of(5,4,2,1,6,12,8,9)
                         .gather(biggestInt(11))
                         .parallel()
                         .findFirst()
                         .get());
```

Previous Page
Next Page