Operaciones Terminales

Las operaciones terminales son aquellas operaciones que como resultado no generan un nuevo `Stream`. Su resultado puede variar según la operación. La utilidad de estas es poder generar un valor final a todas nuestras operaciones o consumir los datos finales. La razón principal para querer esto es que los datos deberán salir en algún punto de nuestro control y es con las operaciones terminales que hacemos esto.

Pensemos, por ejemplo, en un servidor web. Recibe una petición de datos, convierte la petición en un `Stream`, procesa los datos usando `filter` o `map`, convierte de JSON a datos locales que sean manipulables por código Java y hace consumo de una base de datos. Todo esto mediante streams de diferentes tipos. Pero eventualmente tiene que devolver una respuesta para quien le hizo la petición.

¿Qué pasa si quien hizo la petición no esta usando Java? No podemos enviarle un objeto de tipo `Stream` a un código hecho en Python o en JavaScript… es ahi donde una operación final nos ayuda a convertir nuestro `Stream` de Java en algún tipo de dato que sea mas comprensible.

Otro ejemplo claro es si estamos creando una librería o creando código que más gente en nuestro equipo usará. Al crear nuestros métodos y clases usamos streams por aquí y lambdas por allá, pero al exponer estos métodos para uso de otros desarrolladores no podemos obligarlos a usar `Stream`.

Las razones son variadas. No queremos obligar y limitar a quienes usen nuestro código a trabajar con un solo tipo dato. No sabemos qué versión de Java está usando quien use nuestro código. No sabemos si `Stream` está disponible en su parte del código (por ejemplo, en Android no estaba disponible del todo), etc.

Es por ello que quisiéramos proveer de algo mas simple: listas, primitivos o incluso dar algún mecanismo para poder usar código externo de nuestro lado.

Las operaciones terminales más comunes que se encuentran en `Stream` son:

- `anyMatch()`
- `allMatch()`
- `noneMatch()`
- `findAny()`
- `findFirst()`
- `min()`
- `max()`
- `reduce()`
- `count()`
- `toArray()`
- `collect()`
- `forEach()`

Revisaremos qué hacen y qué utilidad tienen durante esta lectura.

Operaciones terminales de coincidencia

anyMatch, allMatch, noneMatch

Las operaciones `anyMatch`, `allMatch` y `noneMatch` sirven para determinar si en un `Stream` hay elementos que cumplan con un cierto `Predicate`. Esto puede ser una forma simple de validar los datos de un `Stream`. Son terminales pues las tres retornan un `boolean`:

```
//Nos indica si un stream contiene un elemento según el Predicate que le
pasemos:
Stream numbersStream = Stream.of(1, 2, 3, 4, 5, 6, 7, 11);
```

```
boolean biggerThanTen = numbersStream.anyMatch(i -> i > 10); //true porque
tenemos el 11

//allMatch
//Nos indica si todos los elementos de un Stream cumplen con un cierto
Predicate:
Stream agesStream = Stream.of(19, 21, 35, 45, 12);
boolean allLegalDrinkingAge = agesStream.allMatch(age -> age > 18); //false,
tenemos un menor

//noneMatch
//Nos indica si todos los elementos de un Stream NO CUMPLEN un cierto
Predicate:
Stream oddNumbers = Stream.of(1, 3, 5, 7, 9, 11);
boolean allAreOdd = oddNumbers.noneMatch(i -> i % 2 == 0);
```

Operaciones terminales de búsqueda

findAny, findFirst

Estas operaciones retornan un `Optional` como resultado de buscar un elemento dentro del `Stream`.

La diferencia entre ambas es que `findFirst` retornara un `Optional` conteniendo el primer elemento en el `Stream` si el `Stream` tiene definida previamente una operación de ordenamiento o para encontrar elementos. De lo contrario, funcionará igual que `findAny`, tratando de devolver cualquier elemento presente en el `Stream` de forma no determinista (random)

Si el elemento encontrado es `null`, tendrás que lidiar con una molesta `NullPointerException`. Si el `Stream` esta vacío, el retorno es equivalente a `Optional.empty()`.

La principal razón para usar estas operaciones es poder usar los elementos de un `Stream` después haber filtrado y convertido tipos de datos. Con `Optional` nos aseguramos que, aún si no hubiera resultados, podremos seguir trabajando sin excepciones o escribiendo condicionales para validar los datos.

Operaciones terminales de reducción

min, max

Son dos operaciones cuya finalidad es obtener el elemento más pequeño (`min`) o el elemento más grande (`max`) de un `Stream` usando un `Comparator`. Puede haber casos de `Stream` vacíos, es por ello que las dos operaciones retornan un `Optional`para en esos casos poder usar `Optional.empty`.

La interfaz `Comparator` es una `@FunctionalInterface`, por lo que es sencillo usar `min` y `max` con lambdas:

```
Stream bigNumbers = Stream.of(100L, 200L, 1000L, 5L);
Optional minimumOptional = bigNumbers.min((numberX, numberY) -> (int)
Math.min(numberX, numberY));
```

reduce

Esta operación existe en tres formas:

- `reduce(valorInicial, BinaryOperator)`
- `reduce(BinaryAccumulator)`
- `reduce(valorInicial, BinaryFunction, BinaryOperator)`

La diferencia entre los 3 tipos de invocación:

reduce(BinaryAccumulator)

Retorna un Optional del mismo tipo que el Stream, con un solo valor resultante de aplicar el `BinaryAccumulator` sobre cada elemento o `Optional.empty()` si el stream estaba vacío. Puede generar un `NullPointerException` en casos donde el resultado de `BinaryAccumulator` sea `null`.

```
Stream aLongStoryStream = Stream.of("Cuando", "despertó,", "el",
"dinosaurio", "todavía", "estaba", "allí.");
Optional longStoryOptional = aLongStoryStream.reduce((previousStory,
nextPart) -> previousStory + " " + nextPart);
longStoryOptional.ifPresent(System.out::println); //"Cuando despertó, el
dinosaurio todavía estaba allí."
```

reduce(valorInicial, BinaryOperator)

Retorna un valor del mismo tipo que el `Stream` después de aplicar `BinaryOperator`sobre cada elemento del `Stream`. En caso de un `Stream` vacío, el `valorInicial` es retornado.

```
Stream firstTenNumbersStream = Stream.iterate(0, i -> i + 1).limit(10);
int sumOfFirstTen = firstTenNumbersStream.reduce(0, Integer::sum); //45 -> 0
+ 1 + … + 9
```

Y el caso mas interesante…

reduce(valorInicial, BinaryFunction, BinaryOperator)

Genera un valor de tipo `V` después de aplicar `BinaryFunction` sobre cada elemento de tipo `T` en el `Stream` y obtener un resultado `V`.

Esta version de reduce usa el `BinaryFunction` como `map + reduce`. Es decir, por cada elemento en el `Stream` se genera un valor `V` basado en el `valorInicial` y el resultado anterior de la `BinaryFunction`. `BinaryOperator` se utiliza en streams paralelos (`stream.parallel()`) para determinar el valor que se debe mantener en cada iteración.

```
Stream aLongStoryStreamAgain = Stream.of("Cuando", "despertó,", "el",
"dinosaurio", "todavía", "estaba", "allí.");
int charCount = aLongStoryStreamAgain.reduce(0, (count, word) -> count +
word.length(), Integer::sum);
```

count

Una operación sencilla: sirve para obtener cuantos elementos hay en el `Stream`.

```
Stream yearsStream = Stream.of(1990, 1991, 1994, 2000, 2010, 2019, 2020);
long yearsCount = yearsStream.count(); //7, solo nos dice cuantos datos tuvo
el stream.
```

La principal razón de usar esta operación es que, al aplicar `filter` o `flatMap`, nuestro `Stream` puede crecer o disminuir de tamaño y, tal vez, de muchas operaciones solo nos interese saber cuántos elementos quedaron presentes en el `Stream`. Por ejemplo, cuantos archivos se borraron o cuantos se crearon por ejemplo.

toArray

Agrega todos los elementos del `Stream` a un arreglo y nos retorna dicho arreglo. La operación genera un `Object[]`, pero es sposible hacer castings al tipo de dato del `Stream`.

collect

Mencionamos la operación `collect` en la lectura sobre *operaciones y collectors*, donde mencionamos que:

> `Collector` es una interfaz que tomara datos de tipo `T` del `Stream`, un tipo de dato mutable `A`, donde se irán agregando los elementos (mutable implica que podemos cambiar su contenido, como un `LinkedList`) y generara un resultado de tipo `R`.
> Usando `java.util.stream.Collectors` podemos convertir sencillamente un `Stream` en un `Set`, `Map`, `List`, `Collection`, etc. La clase `Collectors` ya cuenta con métodos para generar un `Collector` que

> corresponda con el tipo de dato que tu `Stream` esta usando. Incluso vale la pena resaltar que `Collectors` puede generar un `ConcurrentMap` que puede ser de utilidad si requieres de multiples threads.

```
public List getJavaCourses(Stream coursesStream) {
    List javaCourses =
        coursesStream.filter(course -> course.contains("Java"))
            .collect(Collectors.toList());

    return javaCourses;
}
```

Operaciones terminales de iteración

forEach

Tan simple y tan lindo como un clásico `for`. `forEach` es una operación que recibe un `Consumer` y no tiene un valor de retorno (`void`). La principal utilidad de esta operación es dar un uso final a los elementos del `Stream`.

```
Stream> courses = getCourses();
courses.forEach(courseList -> System.out.println("Cursos disponibles: " +
courseList));
```

Conclusiones

Las operaciones terminales se encargan de dar un fin y liberar el espacio usado por un `Stream`. Son también la manera de romper los encadenamientos de métodos entre streams y regresar a nuestro código a un punto de ejecución lineal. Como su nombre lo indica, por lo general, son la ultima operación presente cuando escribes chaining:

```
Stream infiniteStream = Stream.iterate(0, x -> x + 1);
List numbersList = infiniteStream.limit(1000)
```

```
        .filter(x -> x % 2 == 0) // Operación intermedia
        .map(x -> x * 3) //Operación intermedia
        .collect(Collectors.toList()); //Operación final
```

Por ultimo, recuerda que una vez que has agregado una operación a un `Stream`, el `Stream` original ya no puede ser utilizado. Y más aun al agregar una operación terminal, pues esta ya no crea un nuevo `Stream`. Internamente, al recibir una operación, el `Stream` en algún punto llama a su método `close`, que se encarga de liberar los datos y la memoria del `Stream`.