Streamapi Document (v1)

Created by

Vineet Semwal (vineetsemwal82@gmail.com)

Stream api

1) Allows developers process data in a declarative way
2) Leverage multicore architecture without writing a single line of multithread code
3) Not a data structure or storage
4) Designed for lambdas
5) Do not support indexed access
6) Lazy
7) Original source doesn't get changed ever

Operations Type

1) Intermediate
2) Terminal

Stream operations can be piped ie. Result(Stream) of previous operation passed to next operation which will result in another Stream, All this declaration is lazy and will not execute unless there is a terminal operation

A terminal operation will consume the stream to produce the result foreg. max(), min(), forEach(), reduce(), collect(), count()

A stream gets closed after terminal operation so can't be used again, if it is used again then IllegalStateException will be thrown

stream() and parallelStream() methods are provided in the specification Collection itself so all the implementations have them

```
List<Integer>list=Arrays.asList(1,2,3);
Stream<Integer>stream=list.stream();
```

```
int numbers[]={5,6,7}
```
A stream can also be obtained from array
```
Stream<Integer> stream = Arrays.stream(numbers);
```

Original data source does NOT get changed ever

Important Operations Performed on Stream

1) filter(*) : intermediate

1) map(*) : intermediate(Useful for converting a stream of one type to stream of another type)
2) skip(long):intermediate (returns a stream where first n elements mention in method arg are skipped)
3) limit(long): intermediate(returns a stream where size of stream is limited to count mentioned in method arg)
4) reduce(*) : terminal operation(used for reducing stream to a single value)
5) collect(*): terminal operation(used for obtaining collection from stream)
6) count(): terminal operation(returns count of elements in stream)

Collecting

Like Stream can be obtained from a collection, we can obtain collection from stream using collect method , collect is a terminal operation
<R, A> R collect(Collector<? super T, A, R> func)
R specifies the type of result, T specifies element type, the accumulated type is specified by A
interface Collector<T, A, R>
Collectors utility class defines a methods that returns collectors for list or set

```
static Collector<T, ?, List<T>> toList()
static Collector<T, ?, Set<T>> toSet()
static Collector<T, ?, Map<K,U>> toMap(Function<? super T, ? extends K> keyMapper,
                    Function<? super T, ? extends U> valueMapper)
```

Examples

```
Stream<Integer>stream=Stream.of(1,2,3,4);
List<Integer>desired=stream.collect(Collectors.toList());

Stream<Integer>stream=Stream.of(1,2,3,4);
Set<Integer> set=stream.collect(Collectors.toSet());

Stream<String>stream=Stream.of("hi","hello");
Map<String,Integer>map=stream.collect(Collectors.toMap(
input->input,input->input.length()));
```

For parallel stream combiner will be required too

Filtering

filter method of stream takes a predicate as an argument and returns a stream including all elements that matched predicate

```
Stream<T> filter(Predicate<? super T> predicate)
```

```
List<Integer>list=Arrays.asList(1,2,3,4,5,6);
Stream<Integer>desiredStream=list.stream().filter((input)->input%2==0);//desired stream
List<Integer>desired=desiredStream.collect(Collectors.toList());
```

Distinct Elements

distinct method of stream returns the stream with unique elements

```
Stream<T> distinct();
```

Limit size of stream

limit method returns the stream of max size specified as an argument

```
Stream<T> limit(long maxSize);
```

Skipping elements

skip method returns the stream with first n elements discarded

```
Stream<T> skip(long n);
```

map

2) Useful for returning a stream of objects returned by result

3) Useful for converting a stream of one type to stream of another type

Function that is used by map method takes element of one type and can return element of other type

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper)
```

From the method definition it can be seen

```
Stream<String>stream=Stream.of("hi","hello");
Function<String,Integer>function=input->input.length();
Stream<Integer>lengthStream=stream.map(function);
```

Or in one line

```
Stream<Integer> lengthStream = Stream.of("hi", "hello").map(input -> input.length());
```

In above example we converted a stream of string to a stream of integer

ie. converted Stream<String> to Stream<Integer>

Another Example

Stream of integers to stream of square of integers

```
Stream<Integer>stream= Stream.of(1, 2,3).map(input -> input*input);
```

flatMap

Used for flattening the stream, Function that is used by flatMap takes an argument and returns the stream

```
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper);
```

```
List<Integer> list1 = Arrays.asList(1, 2, 3);
List<Integer> list2 = Arrays.asList(4, 5, 6);
List<List<Integer>> listOfList = new ArrayList<>();
listOfList.add(list1);
listOfList.add(list2);
Stream<List<Integer>>inputStream=listOfList.stream();
Stream<Integer> desiredStream =inputStream.flatMap(list -> list.stream());
```

Flattened a Stream<List<Integer>to Stream<Integer>using flatMap

Another example

```
List<String>lines=new ArrayList<>();
lines.add("How are you");
lines.add("i am good");
Stream<String>wordStream=lines.stream().flatMap(line->{
   String[] words=line.split(" +");
  return Arrays.stream(words);});
```

In the above example we flattened Stream<String[]>to Stream<String>

reduce

1) Reduction operations reduces stream to a single value
 foreg inbuilt max(), min()

2) Reduce operation is also called fold in functional programming

3) Generic reduce method is also provided so we can reduce as per need

```
Optional<T> reduce(BinaryOperator<T> accumulator)
```

```
Stream<Integer>stream=Stream.of(1,2,3,4);
Optional<Integer>sum=stream.reduce( (a,b)->a+b);
```