5장. 스트림 활용

이 장에서는 스트림 API가 지원하는 다양한 연산을 살펴본다.

필터링

스트림의 요소를 선택하는 방법, 즉 프레디케이트 필터링 방법과 고유 요소만 필터링하는 방법을 배운다,

프레디케이트로 필터링

filter 메서드는 프레디케이트를 인수로 받아서 프레디케이트와 일치하는 모든 요소를 포함하는 스트림을 반환한다.

```
IntStream intStream = IntStream.of(3, 7, 1, 6, 2, 13, 13, 7, 10, 13, 26);
intStream.filter(number -> number >= 7)
        .boxed()
        .forEach(System.out::println); // 7, 13, 13, 7, 10, 13, 26
```

고유 요소 필터링

고유 요소로 이루어진 스트림을 반환하는 필터링이다. 고유 요소 판단은 hashcode와 equals를 사용한다.

```
IntStream intStream = IntStream.of(3, 7, 1, 6, 2, 13, 13, 7, 10, 13, 26);
intStream.filter(number -> number >= 7)
        .distinct()
        .boxed()
        .forEach(System.out::println); // 7, 13, 10, 26
```

실제로 equals와 hashcode를 기준으로 고유 요소를 선택하는지 아래 코드를 통해 간단히 테스트 해봤다. 아래 코드를 실행해보면 100개가 아닌 21개가 나오는 것을 확인할 수 있다.

```
public class UniqueElementFiltering {
    public static void main(String[] args) {
        IntStream.rangeClosed(1, 100)
                .mapToObj(number -> new Remainder(number))
                .distinct()
                .forEach(remainder -> System.out.println(remainder));

    }
}

class Remainder {
    public int value;

    public Remainder(int value) {
        this.value = value;
    }

    @Override
    public boolean equals(Object o) {
        return true;
    }

    @Override
    public int hashCode() {
        return value / 5;
    }

    @Override
    public String toString() {
        return "Remainder{" +
                "value=" + value +
                '}';
    }
}
```

스트림 슬라이싱

스트림의 요소를 선택하거나 스킵하는 다양한 방법을 설명한다.

프레디케이트를 이용한 슬라이싱

정렬된 스트림인 경우 프레디케이트에 참 혹은 거짓인 데이터가 발견 됐을 때 굳이 마지막 요소까지 탐색을 하거나, 불필요한 특정 요소에 대해 처리를 할 필요가 있을까?

TAKEWHILE 활용

takewhile은 스트림 처리 과정에서 거짓을 만나면 스트림이 중단된다.

```
int result = (int) IntStream.rangeClosed(1, 10)
                .takeWhile(num -> {
                    System.out.print("#");
                    return num <= 3;
                })
                .count();
System.out.println("\nresult : " + result);

// console
####
3
```

DROPWHILE 활용

dropwhile은 스트림 처리 과정에서 참인 요소는 버리고 거짓인 요소부터 처리를 한다.

```
int result = (int) IntStream.rangeClosed(1, 10)
                .takeWhile(num -> {
                    System.out.print("#");
                    return num <= 3;
                })
                .count();
System.out.println("\nresult : " + result);

// console
####
7
```

스트림 축소

limit 함수에 파라미터로 넣은 건수 만큼의 데이터가 확인되면 이후 연산을 생략한다.

요소 건너뛰기

처음 n개 요소를 제외한 스트림을 반환하는 skip(n) 메소드를 지원한다.

매핑

특정 객체에서 특정 데이터를 선택하는 작업이다.

스트림의 각 요소에 함수 적용하기

인수로 제고된 함수는 각 요소에 적용되며 함수를 적용한 결과가 새로운 요소로 매핑된다.

```
List<String> dishNames = menus.stream()
                        .map(Dish::getName)
                        .collect(toList());
```

스트림 평면화

간단히 Stream 안에 있는 요소를 풀어서 하나로 연결시켜주는 기능이다.

아래 코드를 보자.

```
Stream<String> childStreamOne = Stream.of("H", "e", "l", "l", "o");
Stream<String> childStreamTwo = Stream.of("w", "o", "r", "l", "d");
Stream<Stream<String>> parentStream = Stream.of(childStreamOne, childStreamTwo);

parentStream.flatMap(stream -> stream)
        .forEach(result -> System.out.println(result));
```

별도의 스트림인 `childStreamOne`과 `childStreamTwo`를 `parentStream`에 넣고 단순하게 `flatMap`만 해준 뒤 `forEach`로 콘솔에 출력이 되도록 했다.

그 결과 하나의 스트림으로 연결된 데이터 처럼 화면에 출력된 것을 확인할 수 있었다.

정리하자면, `flatMap`은 동일한 제네릭인 스트림인 서로 다른 스트림을 하나의 스트림으로 연결시켜주는 중간 연산이라고 정리할 수 있겠다.

검색과 매칭

특정 속성이 데이터 집합에 있는지 여부를 검색하는 데이터 처리도 자주 사용된다. 스트림 API는 `allMatch`, `anyMatch`, `noneMatch`, `findFirst`, `findAny` 등 다양한 유틸리티 메소드를 제공한다.

anyMatch

프레디케이스 연산으로 하나라도 조건에 일치하는 항목이 있는지 확인하고, true가 확인되는 즉시 반환한다. 쇼트 서킷이 되는데, 아래 코드의 경우 두 번째 true가 확인되는 즉시 종료되어서 ###이 두번만 출력된다.

```
Stream<Boolean> booleanStream = Stream.of(false, true, false, false);

booleanStream.anyMatch(s -> {
    System.out.print("#");
    return s;
});
// console: ##
```

allMatch

스트림 내 **모든 요소**가 조건에 부합하는지 확인하는데, 하나라도 거짓인 요소가 확인되면 쇼트 서킷이 된다.

```
Stream<Boolean> booleanStream = Stream.of(true, true, false, true);

booleanStream.allMatch(s -> {
    System.out.print("#");
    return s;
});

// console: ###
```

noneMatch

allMatch와 반대 연산을 수행하는데, 하나라도 참인 요소가 확인되면 쇼트 서킷이 된다.

```
Stream<Boolean> booleanStream = Stream.of(false, true, false, false);

booleanStream.anyMatch(s -> {
    System.out.print("#");
    return s;
});

// console: ##
```

요소 탐색

findAny

조건에 부합하는 요소를 발견 하는 즉시 스트림을 종료하고 해당 요소를 반환한다.

```
IntStream intStream = IntStream.of(5, 3, 1, 2, 4, 6);

OptionalInt result = intStream
        .filter(num -> {
            System.out.print("#");
            return num == 1;
        })
        .findAny();

System.out.println(result.getAsInt());

// console: ###1
```

findFirst

리스트 또는 정렬된 연속 데이터로부터 생성된 스트림처럼 일부 스트림에는 논리적인 아이템 순서가 정해져 있을 수 있다. 이런 스트림에서 첫번째 요소를 찾기 위해서 써야한다.

```
IntStream intStream = IntStream.of(5, 3, 1, 2, 4, 6);

OptionalInt result = intStream
        .filter(num -> {
            System.out.print("#");
            return num == 1;
        })
        .findFirst();

System.out.println(result.getAsInt());

// console: ###1
```

findAny vs FindFirst

위 두 메소드에 예시 코드의 결과는 모두 동일하다. 그럼에도 필요한 이유는 병렬성 때문이다. 병렬 실행에서는 첫 번째 요소를 찾기 어렵다. 따라서 **요소의 반환 순서가 상관없다면 병렬 스트림에서는 제약이 적은 findAny를 사용한다.**

아래 코드는 병렬 처리시 findAny가 결과에 미치는 영향을 확인 하기 위한 테스트다.

findFirst면 7만 반환이 되지만, findAny면 6과 7이 번갈아서 반환됨을 알 수 있다.

```
IntStream intStream = IntStream.of(3, 5, 3, 5, 3, 5, 3, 1, 2, 7, 6);

OptionalInt result = intStream
        .parallel()
        .filter(num -> {
            System.out.print("#");
            return num >= 6;
        })
        .findAny();

System.out.println(result.getAsInt());
```

리듀싱

모든 스트림 요소를 처리해서 값으로 도출하는 연산을 말한다.

요소의 합

아래 코드와 같이 연산을 위한 함수 디스크립터를 전달해주면 되는데, 첫번째 파라미터는 누적되는 요소이며 두번째 파라미터는 변경되는 요소의 값이다.

```
IntStream intStream = IntStream.of(3, 1, 2, 7, 6);

OptionalInt result = intStream
        .reduce((num1, num2) -> {
            System.out.println("num1: " + num1 + "\t num2:   " + num2);
            return num1 + num2;
        });

System.out.println(result);

// console
/*
num1: 3  num2:   1
num1: 4  num2:   2
num1: 6  num2:   7
num1: 13   num2:   6
OptionalInt[19]
*/
```

아래와 같이 초기 값을 주는 것도 가능하다.

초기값이 있는 경우에는 null이 반환된 확률이 없으므로 OptionalInt가 아닌 int가 반환 타입이 된다.

```
IntStream intStream = IntStream.of(3, 1, 2, 7, 6);

int result = intStream
        .reduce(5, (num1, num2) -> {
            System.out.println("num1: " + num1 + "\t num2:   " + num2);
            return num1 + num2;
        });

System.out.println(result);

// console
/*
num1: 5  num2:   3
num1: 8  num2:   1
num1: 9  num2:   2
num1: 11   num2:   7
num1: 18   num2:   6
24
*/
```

최대값과 최소값

아래 코드와 같이 Integer.max와 min을 사용해 최대/최소값을 구할 수 있다.

```
IntStream intStream = IntStream.of(3, 1, 2, 7, 6);

OptionalInt result = intStream
        .reduce(Integer::max);

System.out.println(result);

intStream = IntStream.of(3, 1, 2, 7, 6);
result = intStream
        .reduce(Integer::min);

System.out.println(result);;
```

병렬 처리시 주의 사항

reduce의 초기값으로 인해서 의도 되지 않은 결과가 나올 수 있다.

parallel() 키워드를 붙임으로서 병렬로 처리되도록 할 수 있는데, 각 쓰레드에서의 초기의 값이 5로 설정되어서 `(5 x 5) + (3 + 1 + 2 + 7 + 6) = 41` 이란 결과가 산출된다.

reduce가 아니더라도 병렬 처리를 했을 때 결과에 영향을 줄지에 대한 확인이 반드시 필요하다.

```
IntStream intStream = IntStream.of(3, 1, 2, 7, 6);

int result = intStream
        .parallel()
        .reduce(5, (num1, num2) -> {
            System.out.println("num1: " + num1 + "\t num2:   " + num2);
            return num1 + num2;
        });

System.out.println(result);
```

스트림 연산: 상태 있음과 상태 없음

filter와 map과 같은 중간 연산은 전달받은 요소에 대한 처리를 한 후 결과를 출력 스트림으로 보낸다. 이러한 연산을 내부 상태를 갖지 않으므로 **상태 없는 연산**이라고 한다.

반면, sorted나 distinct 같은 연산은 전체 요소에 대하여 정렬을 하거나 중복을 제거하므로 과거의 이력을 알고 있어야 하며 결론적으로 상태가 있는 연산이라고볼 수 있다.

아래 코드의 경우 중간에 정렬이 없으면 filter → map → limit → reduce 순서로 3건에 대해서만 진행된다. 하지만 limit 연산 위에 sorted가 들어가면 요소에 대하여 filter와 map이 수행된 후에 limit과 reduce가 수행된다. 이러한 sorted와 같은 연산을 상태가 있는 연산이라고 한다.

(distinct도 상태가 있는 연산 중 하나인데 **distinct는 이전 연산이 모두 수행되지 않아도 수행이 가능하다. 왜 그런지에 대해서는 확인이 필요하다.**)

```
IntStream intStream = IntStream.of(3, 1, 2, 7, 6);

int result = intStream
        .filter(num -> {
            System.out.println("filter");
            return true;
        })
        .map(num -> {
            System.out.println("map");
            return num;
        })
        .limit(3L)
        .reduce(5, (num1, num2) -> {
            System.out.println("num1: " + num1 + "\t num2:   " + num2);
            return num1 + num2;
        });

System.out.println(result);
```

Stream을 만드는 여러가지 방법

of 팩토리 메소드

`IntStream.of(1,2,3,4,5);`

nullable 팩토리 메소드

`Stream.ofNullable(new String());`

배열로 스트림 만들기

```
int[] ints = new int[]{1,2,3,4,5};
Arrays.stream(ints)
```

파일로 스트림 만들기

```
long uniqueWords = 0L;
        try (Stream<String> lines = Files.lines(Paths.get(System.getProperty("user.dir") + "/data.txt"), Charset.defaultCharset())) {
            uniqueWords = lines.flatMap(line -> Arrays.stream(line.split("")))
                    .peek(data -> System.out.println(data))
                    .distinct()
                    .count();
        } catch (IOException e) {
            e.printStackTrace();
        }
        System.out.println(uniqueWords);
```

함수로 무한 스트림 만들기

```
Stream.iterate(0, n-> n + 2)
          .limit(10)
          .forEach(System.out::println);
```