01

Mais Streams, Collectors e APIs

Transcrição

Os Streams possibilitam trabalhar com dados de uma maneira funcional. Normalmente, são dados e objetos que vêm de uma collection do Java. Por que não adicionaram esses métodos diretamente nas Collections? Justo para não ser dependente delas, não ter efeitos colaterais e não entupir de métodos as interfaces.

Vamos conhecer outros métodos interessantes dos Streams. Um exemplo seria: quero um curso que tenha mais de 100 alunos! Pode ser qualquer um deles. Há o método `findAny`

```
cursos.stream()
   .filter(c -> c.getAlunos() > 100)
   .findAny();
```

O que será que devolve o `findAny` ? Um `Curso` ? Não! Um `Optional<Curso>` .

Optional

`Optional` é uma importante nova classe do Java 8. É com ele que poderemos trabalhar de uma maneira mais organizada com possíveis valores `null` . Em vez de ficar comparando `if(algumaCoisa == null)` , o `Optional` já fornece uma série de métodos para nos ajudar nessas situações. Por que o `findAny` utiliza esse recurso? Pois pode não haver nenhum curso com mais de 100 alunos! Nesse caso, o que seria retornado? `null` ? uma exception?

Vamos ver as vantagens de se trabalhar com `Optional` . Primeiro vamos atribuir o resultado do `findAny` a uma variável:

```
Optional<Curso> optional = cursos.stream()
   .filter(c -> c.getAlunos() > 100)
   .findAny();
```

Dado um `Optional` , podemos pegar seu conteúdo invocando o `get` . Ele vai devolver o `Curso` que queremos. Mas e se não houver nenhum? Uma exception será lançada.

```
Curso curso = optional.get();
```

Há métodos mais interessantes. O `orElse` diz que ele deve devolver o curso que existe dentro desse optional, *ou então* o que foi passado como argumento:

```
Curso curso = optional.orElse(null);
```

Nesse caso ou ele devolve o curso encontrado, ou `null` , caso nenhum seja encontrado. Mesmo assim, ainda não está tão interessante. Há como evitar tanto o `null` , quanto as exceptions, quanto os `ifs` . O método `ifPresent` executa um lambda (um `Consumer`) no caso de existir um curso dentro daquele optional:

```
optional.ifPresent(c -> System.out.println(c.getNome()));
```

Claro que, no dia a dia, não teríamos a variável temporária `curso`. Podemos fazer isso

```
cursos.stream()
   .filter(c -> c.getAlunos() > 100)
   .findAny()
   .ifPresent(c -> System.out.println(c.getNome()));
```

Outros métodos devolvem `Optional` nos `Streams`. Um deles é o `average` em `IntStream`. Por que? Pois pode não existir nenhum element, e aí a média poderia realizar uma divisão por zero.

Você vai encontrar Optional não somente na API de Streams. Vale a pena conhecer e utilizá-la no seu próprio código e entidades.

Gerando uma coleção a partir de um Stream

Invocar métodos no `stream` de uma coleção não altera o conteúdo da coleção original. Ele não gera efeitos colaterais. Como então obter uma coleção depois de alterar um `Stream`?

Tentar fazer `List<Curso> novaLista = lista.stream().filter(...)` não compila, pois um `Stream` **não é** uma coleção. Para fazer algo parecido com isso, utilizamos o método `collect`, que *coleta* elementos de um `Stream` para produzir um outro objeto, como uma coleção.

O método `Collect` recebe um `Collector`, uma interface não tão trivial de se implementar. Podemos usar a classe `Collectors` (repare o `s` no final), cheio de *factory methods* que ajudam na criação de coletores. Um dos coletores mais utilizados é o retornado por `Collectors.toList()`:

```
List<Curso> resultados = cursos.stream()
   .filter(c -> c.getAlunos() > 100)
   .collect(Collectors.toList());
```

Pronto! É através dos coletores que podemos "retornar" de um `Stream` para uma `Collection`. Certamente poderia ter usado a mesma variável, a `List<Curso> cursos` que temos:

```
cursos = cursos.stream()
   .filter(c -> c.getAlunos() > 100)
   .collect(Collectors.toList());
```

Pronto. Alteramos a referência antiga para apontar para essa nova coleção, depois de filtrada!

Um exemplo mais complicado? Podemos gerar mapas! Queremos um mapa que, dado o nome do curso, o valor atrelado é a quantidade alunos. Um `Map<String, Integer>`. Utilizamos o `Collectors.toMap`. Ele recebe duas `Functions`. A primeira indica o que vai ser a chave, e a segunda o que será o valor:

```
Map mapa = cursos
.stream()
```

```
    .filter(c -> c.getAlunos() > 100)
    .collect(Collectors.toMap(c -> c.getNome(), c -> c.getAlunos()));
```

Outras vantagens do Stream

Os Streams foram desenhados de uma forma a tirar proveito da programação funcional. Se você utilizá-los da forma que vimos por aqui, eles nunca gerarão efeitos colaterais. Isso é, apenas o stream será alterado, e nenhum outro objeto será impactado.

Dada essa premissa, podemos pedir para que nosso `stream` seja processado em paralelo. Ele mesmo vai decidir quantas threads usar e fazer todo o trabalho, utilizando APIs mais complicadas (como a de fork join) para ganhar performance. Para fazer isso, basta utilizar `parallelStream()` em vez de `stream()` !

Tome cuidado. Para streams pequenos, o custo de cuidado dessas threads e manipular os dados entre elas é alto e pode ser bem mais lento que o `Stream` tradicional.

Não deixe de investigar a API de `Stream` e conhecer os outros métodos que ela possui. Certamente você vai parar de escrever os diversos `fors` encadeados que estamos acostumados, podendo fazer tudo de uma maneira mais legível, fácil e funcional.