JAVA STREAM API

SIMPLES, PRÁTICA E PODEROSA

Aprenda quais são os tipos de operações mais utilizados na hora de manipular as streams do java

EDER FONSECA

ÍNDICE

01

INTRODUÇÃO

Uma breve explicação sobre o que vem a ser a Stream API do JAVA.

INTRODUÇÃO À STREAM API

A Stream API do Java, introduzida no Java 8, é uma ferramenta que facilita o processamento de coleções de dados de forma declarativa e eficiente. Ela permite realizar operações como filtragem, mapeamento e redução de maneira simples e concisa, transformando listas e outras coleções em fluxos de dados que podem ser manipulados com métodos fáceis de usar. Isso resulta em um código mais legível e fácil de manter, além de possibilitar o paralelismo de operações, aproveitando melhor os recursos do hardware para melhorar a performance.

Exemplo de Uso Básico

```java
List<String> nomes = Arrays.asList("Ana", "Bruno", "Carlos");
List<String> nomesComC = nomes.stream()
    .filter(nome → nome.startsWith("C"))
    .collect(Collectors.toList());
```

02

ESTRUTURA E ORGANIZAÇÃO DA STREAM API

A Stream API é organizada em três tipos principais de operações: criação de streams, operações intermediárias e operações finais. Esta seção fornece uma visão geral de cada um desses componentes.

ESTRUTURA E ORGANIZAÇÃO DA STREAM API

Criação de Streams

Streams podem ser criados a partir de diversas fontes, como coleções, arrays, e métodos de fábrica.

Operações Intermediárias

Operações intermediárias transformam ou filtram os elementos de um stream, retornando um novo stream.

Operações Finais

Operações finais produzem um resultado ou efeito colateral, encerrando o processamento do stream.

Coletores (Collectors)

Coletores são usados para acumular os elementos processados de um stream em uma coleção ou outro recipiente.

Paralelismo e Performance

Streams paralelos podem ser usados para melhorar a performance de operações de processamento intensivo.

03

CRIAÇÃO DE STREAMS

Aprenda as principais formas de criar Streams.

CRIAÇÃO DE STREAMS

A Partir de Coleções

Crie streams diretamente de coleções usando o método `***stream()***`.

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);
Stream<Integer> streamNumeros = numeros.stream();
```

A Partir de Arrays

Use o método `***Arrays.stream()***` para criar streams a partir de arrays.

```
String[] frutas = {"Maçã", "Banana", "Laranja"};
Stream<String> streamFrutas = Arrays.stream(frutas);
```

Com Métodos de Fábrica

Utilize métodos de fábrica como `***Stream.of()***` para criar streams.

```
Stream<String> streamAlfabeto = Stream.of("A", "B", "C", "D");
```

04

OPERAÇÕES INTERMEDIÁRIAS

Aprenda as principais formas de transformar e filtrar Streams.

OPERAÇÕES INTERMEDIÁRIAS

Filtrando Dados com `*filter()*`

Filtre elementos de acordo com um predicado.

```
List<String> palavras = Arrays.asList("casa", "carro", "bicicleta");
List<String> palavrasComC = palavras.stream()
    .filter(palavra → palavra.startsWith("c"))
    .collect(Collectors.toList());
```

Transformando Dados com `*map()*`.

Transforme elementos aplicando uma função.

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> numerosAoQuadrado = numeros.stream()
    .map(numero → numero * numero)
    .collect(Collectors.toList());
```

OPERAÇÕES INTERMEDIÁRIAS *(Cont.)*

Achatar Estruturas com `*flatMap()*`

Combine vários streams em um único stream.

```
List<List<String>> listaDeListas = Arrays.asList(
    Arrays.asList("a", "b"),
    Arrays.asList("c", "d")
);
List<String> todasAsLetras = listaDeListas.stream()
    .flatMap(List::stream)
    .collect(Collectors.toList());
```

05

OPERAÇÕES FINAIS

Aprenda as principais formas de produzir um resultado encerrando o processamento do stream.

OPERAÇÕES FINAIS

Coletando Resultados com `*collect()*`

Colete resultados em uma coleção ou outro recipiente.

```
List<String> frutas = Arrays.asList("Maçã", "Banana", "Laranja");
Set<String> frutasSet = frutas.stream()
    .collect(Collectors.toSet());
```

Iterando com `*forEach()*`

Execute uma ação para cada elemento.

```
List<String> nomes = Arrays.asList("Ana", "Bruno", "Carlos");
nomes.stream()
    .forEach(System.out::println);
```

Reduzindo com `*reduce()*`

Reduza o stream a um único valor.

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5);
int soma = numeros.stream()
    .reduce(0, Integer::sum);
```

06

COLETORES

Aprenda sobre as diferentes formas de coleta de elementos para uma nova coleção.

COLETORES

Coletando Resultados com `*collect()*`

Colete elementos em uma lista ou conjunto.

```
List<String> frutas = Arrays.asList("Maçã", "Banana", "Laranja");
List<String> frutasList = frutas.stream()
    .collect(Collectors.toList());
```

Agrupando Dados com `*groupingBy()*`

Agrupe elementos com base em uma característica.

```
List<String> palavras =
    Arrays.asList("Ana", "Bruno", "Carlos", "Daniela");

Map<Character, List<String>> palavrasPorLetraInicial =
    palavras
        .stream()
        .collect(Collectors.groupingBy(palavra → palavra.charAt(0)));
```

COLETORES *(Cont.)*

Particionando Dados com `*partitioningBy()*`

Particione elementos com base em um predicado.

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6);
Map<Boolean, List<Integer>> numerosPares =
    numeros.stream()
        .collect(
            Collectors.partitioningBy(numero → numero % 2 == 0)
        );
```

07

PARALELISMO E PERFORMANCE

Melhore a performance de operações de processamento intensivo

PARALELISMO E PERFORMANCE

Usando Streams Paralelas

Aproveite múltiplos núcleos do processador.

```
List<Integer> numeros = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

int somaParalela = numeros.parallelStream()
    .reduce(0, Integer::sum);
```

Considerações de Performance

Streams paralelas podem melhorar a performance em operações intensivas, mas devem ser usadas com cuidado, pois nem todas as operações são adequadas para paralelismo.

AGRADECIMENTOS

OBRIGADO POR LER ATÉ AQUI! 💜

⚠️ Atenção ⚠️

Esse Ebook foi gerado com auxílio de IA, e diagramado por humano. O passo a passo se encontra no meu Github.

Esse conteúdo foi gerado com fins didáticos de construção, não foi realizada uma validação cuidadosa humana no conteúdo e pode conter erros gerados por uma IA.

https://github.com/ederluisf/dio-desafio-ebook