Java Streams - Introdução ao Java Stream

Um fluxo é uma sequência de valores de dados suporte a operações agregadas sequenciais e paralelas.

Usamos as funções agregadas em SQL com mais frequência. Por exemplo, podemos somar todos os números de vendas por um mês ou um ano. Também podemos obter o valor máximo para um determinado intervalo.

Uma operação agregada funciona em uma lista de itens e resulta em um único valor.

O resultado de uma operação agregada em fluxo pode ser um valor primitivo, um objeto ou um vazio para Stream. Como o SQL, podemos calcular a soma para todos os inteiros em um fluxo de inteiros.

Coleções vs Streams

As Coleções Java se concentram em como armazenar elementos de dados para acesso eficiente.

Os fluxos Java se concentram em operações agregadas em elementos de dados de uma fonte de dados.

Recursos do Streams

Java Streams têm seus próprios recursos.

Sem armazenamento

Java Streams não têm armazenamento.

Uma coleção é uma estrutura de dados na memória que armazena todos os seus elementos.

Um fluxo não tem armazenamento. Um fluxo extrai elementos de uma fonte de dados sob demanda e os passa para um pipeline de operações para processamento.

Para uma coleção falamos sobre o armazenamento ou como os dados elementos são armazenados, como acessar elementos de dados.

Para um fluxo, nos concentramos nas operações, por exemplo, como somar um fluxo.

Fluxos infinitos

Uma coleção não pode representar um grupo de elementos infinitos, enquanto um fluxo pode.

Um fluxo pode extrair seus elementos de uma fonte de dados. A fonte de dados pode ser uma coleção, uma função que gera dados, um canal de E/S, etc.

Um fluxo pode extrair dados de uma função que gera infinito número de elementos.

Não reutilizável

Os fluxos não são reutilizáveis

Um fluxo não pode ser reutilizado depois de chamar uma operação de terminal.

Para executar uma computação nos mesmos elementos da mesma fonte de dados, temos que recriar o pipeline de fluxo.

Um fluxo pode lançar um IllegalStateException em caso de reutilização

Java Streams - Operação Java Stream

Iteração externa

Ao operar com Coleções Java, usamos iteração externa.

Na iteração externa, usamos um para ou para cada loop ou Obter um iterador para uma coleção e elementos de processo das coletas em sequência.

O código a seguir calcula a soma dos quadrados de todos os inteiros ímpares na lista.

Ele usa para cada loop para acessar cada elemento na lista em seguida: usa a instrução if para filtrar inteiros ímpares.

Depois disso, ele calcula o quadrado e, finalmente, armazena a soma do quadrado com a variável. sum

**import** java.util.Arrays;

**import** java.util.List;

/\*from w w w. j a va 2 s .co m\*/

**public** **class** Main {

**public** **static** **void** main(String[] args) {

List<**Integer**> numbers = Arrays.asList(1, 2, 3, 4, 5);

**int** sum = 0;

**for** (**int** n : numbers) {

**if** (n % 2 == 1) {

**int** square = n \* n;

sum = sum + square;

}

}

System.out.println(sum);

}

}

O código acima gera o seguinte resultado.



Iteração interna

Podemos reescrever o código acima usando stream. Faz exatamente o mesmo.

**import** java.util.Arrays;

**import** java.util.List;

/\* w ww.ja v a2s .c om\*/

**public** **class** Main {

**public** **static** **void** main(String[] args) {

List<**Integer**> numbers = Arrays.asList(1, 2, 3, 4, 5);

**int** sum = numbers.stream()

.filter(n -> n % 2 == 1)

.map(n -> n \* n)

.reduce(0, **Integer**::sum);

System.out.println(sum);

}

}

No código acima, não usamos a instrução loop para iterar pela List. Fizemos o loop internamente por stream.

Para o cálculo do número inteiro ímpar, foi utilizada a expressão lambda. Nós primeiro fez o filtro então mapear e depois reduzir.

O código acima gera o seguinte resultado.



Sequencial

A iteração externa normalmente significa código sequencial. O código sequencial pode ser executado apenas por um thread.

Os fluxos são projetados para processar elementos em paralelo.

O código a seguir calcula a soma dos quadrados de ímpar inteiros na lista em paralelo.

**import** java.util.Arrays;

**import** java.util.List;

/\* w ww .j a v a2s. c o m\*/

**public** **class** Main {

**public** **static** **void** main(String[] args) {

List<**Integer**> numbers = Arrays.asList(1, 2, 3, 4, 5);

**int** sum = numbers.parallelStream()

.filter(n -> n % 2 == 1)

.map(n -> n \* n)

.reduce(0, **Integer**::sum);

System.out.println(sum);

}

}

O que fizemos foi apenas substituir por . stream()parallelStream()

A computação paralela é fácil ao usar o iteração fornecida pelo fluxo.

O código acima gera o seguinte resultado.



Imperativo x Funcional

Na programação imperativa, controlamos não apenas o que fazer, mas também como fazê-lo.

Por exemplo, ao usar programação imperativa para somar inteiros em uma lista. Temos que decidir como iterar cada elemento na lista. Nós podemos usar para loop, para cada loop, ou podemos obter um objeto Iterator da lista e use enquanto loop. Então também temos que fazer a soma.

Na programação declarativa só precisamos dizer o que fazer, a forma como a parte é tratada pelo próprio sistema.

As coleções suportam a programação imperativa, considerando que Os fluxos suportam programação declarativa.

A API Streams oferece suporte à programação funcional usando a expressão lambda.

Quais operações queremos executar nos elementos de fluxo são feitas normalmente passando expressões lambda.

As operações em um fluxo produzem um resultado sem modificar a fonte de dados.

Operações intermediárias Operações terminais

Um fluxo oferece suporte a dois tipos de operações:

* Operações intermediárias
* Operações em terminais

As operações intermediárias também são chamadas de operações preguiçosas.

As operações de terminal também são chamadas de operações ansiosas.

Uma operação lenta não processa os elementos até que uma operação ansiosa seja chamada no fluxo.

Uma operação intermediária em um fluxo produz outro fluxo.

Operações de link de fluxos para criar um pipeline de fluxo.

No código a seguir, filter() e map() são todas operações lentas. Enquanto reduce() é operação ansiosa.

**import** java.util.Arrays;

**import** java.util.List;

/\* ww w .j a v a 2 s . co m\*/

**public** **class** Main {

**public** **static** **void** main(String[] args) {

List<**Integer**> numbers = Arrays.asList(1, 2, 3, 4, 5);

**int** sum = numbers.parallelStream()

.filter(n -> n % 2 == 1)

.map(n -> n \* n)

.reduce(0, **Integer**::sum);

System.out.println(sum);

}

}

O código acima gera o seguinte resultado.



Java Streams - Java Stream API

As interfaces e classes relacionadas ao fluxo estão no pacote java.util.stream.

A interface AutoCloseable é do pacote java.lang.

Todas as interfaces de fluxo herdam da interface BaseStream, que herda da interface AutoCloseable.

AutoCloseable

|

+--BaseStream

|

+--IntStream

|

+--LongStream

|

+--DoubleStream

|

+--Stream<T>

Se os fluxos usarem coleções como fonte de dados, e as coletas não precisam ser fechadas.

Se um fluxo for baseado em uma fonte de dados que pode ser fechada, como um canal de E/S de arquivo, Podemos criar o fluxo usando uma instrução try-with-resources para fechá-lo automaticamente.

BaseStream

A interface BaseStream define todos os métodos comuns a todos os tipos de fluxos.

* A operação de terminal iterator<T> iterator() retorna um iterador para o fluxo.
* A operação intermediária sequential() retorna um fluxo sequencial. Se o fluxo já for sequencial, ele retornará a si mesmo. Ele converte um fluxo paralelo em um fluxo sequencial.
* A operação intermediária parallel() retorna um fluxo paralelo. Se o fluxo já é paralelo, ele retorna a si mesmo. Ele converte um fluxo paralelo em um fluxo sequencial.
* boolean isParallel() retorna true se o fluxo for paralelo, false caso contrário.  
  Chamando esse método depois de invocar um terminal O método de operação de fluxo pode produzir resultados imprevisíveis.
* A operação intermediária unordered() retorna uma versão não ordenada do fluxo. Se o fluxo já estiver desordenado, ele retornará a si mesmo.

Riacho

Stream<T> interface representa um fluxo do elemento tipo T.

Um Stream<Student> representa um fluxo de Studentobjects.

A interface Stream<T> contém métodos como filter(), map(), reduce(), collect(), max(), min(), etc.

Ao trabalhar com tipos primitivos, podemos usar três interfaces de fluxo especializadas chamadas , e . IntStreamLongStreamDoubleStream

Essas interfaces fornecem métodos para lidar com valores primitivos.

Para outros tipos primitivos, como float, short, byte, ainda podemos Use as três interfaces de fluxo especializadas.

No código a seguir, vamos usar o fluxo para calcular uma soma de os quadrados de todos os inteiros ímpares na lista.

Usaremos as etapas a seguir para fazer o cálculo.

Criar um fluxo

O método stream() na interface Collection retorna um fluxo sequencial. Dessa forma, a Coleção atua como a fonte de dados.

O código a seguir cria um List<Integer> e obtém um Stream<Integer> da lista:

List<Integer> numbersList = Arrays.asList(1, 2, 3, 4, 5);

Stream<Integer> numbersStream = numbersList.stream();

Filtrar um fluxo

Stream<T> filter() usa a para manter elementos se o Predicado especificado retornar true para esse elemento. Predicate<T>

A instrução a seguir obtém um fluxo de apenas inteiros ímpares:

Stream<Integer> oddNumbersStream= numbersStream.filter(n ->n % 2 == 1);

Mapear um fluxo

Stream<T> map() usa uma função para mapear cada elemento no fluxo e criar um novo fluxo.

A instrução a seguir mapeia um fluxo para seus quadrados:

Stream<**Integer**> aStream = stream.map(n -> n \* n);

Reduzir um fluxo

reduce(T identity, BinaryOperator<T> accumulator) reduz o fluxo a um único valor.

É preciso um valor inicial e um acumulador que é um como Argumentos. BinaryOperator<T>

reduce(T identity, BinaryOperator<T> accumulator) executa uma redução nos elementos deste fluxo, usando o forneceu valor inicial e uma função de acumulação associativa, e retorna o valor reduzido.

Isso equivale a:

T result = identity;

**for** (T element : **this** stream)

result = accumulator.apply(result, element)

**return** result;

O código a seguir soma todos os inteiros no fluxo.

int sum = aStream.reduce(0, (n1, n2) -> n1 + n2);

O método executa a soma de dois inteiros. Integer.sum()

Podemos reescrever o código usando uma referência de método.

int sum = aStream.reduce(0, Integer::sum);

Junto

O código a seguir vincula cada etapa.

**import** java.util.Arrays;

**import** java.util.List;

//from w w w .ja va 2 s .c om

**public** **class** Main {

**public** **static** **void** main(String[] args) {

List<**Integer**> numbers = Arrays.asList(1, 2, 3, 4, 5);

**int** sum = numbers.stream()

.filter(n -> n % 2 == 1)

.map(n -> n \* n)

.reduce(0, **Integer**::sum);

System.out.println(sum);

}

}

O código acima gera o seguinte resultado.



Fluxos ordenados vs. fluxos não ordenados

Um fluxo pode ser ordenado ou não ordenado.

Um fluxo ordenado mantém a ordem de seus elementos.

A API Streams pode converter um fluxo ordenado, que pode representar uma fonte de dados ordenada, como uma lista ou um conjunto classificado, em um fluxo não ordenado.

Também podemos converter um fluxo não ordenado em um ordenado aplicando uma operação intermediária de classificação.

**import** java.util.Arrays;

**import** java.util.List;

/\* ww w . j ava 2s . c o m\*/

**public** **class** Main {

**public** **static** **void** main(String[] args) {

List<**Integer**> numbers = Arrays.asList(3,7,9,3,1,2,1, 2, 3, 4, 5);

numbers.stream()

.filter(n -> n % 2 == 1)

.sorted()

.forEach(System.out::println);

}

}

O código acima gera o seguinte resultado.

Texto

Descrição gerada automaticamente com confiança média

Java Streams - Java Opcional

Java 8 introduziu uma classe java.util.Optional<T> para lidar com NullPointerException graciosamente.

Um opcional é um wrapper para um valor não nulo que pode ou não conter um valor não nulo.

Métodos que podem retornar null devem retornar um Opcional em vez de nulo.

O isPresent() de Optional<T> retorna true se contiver um valor não nulo, false caso contrário.

O método get() retorna o valor não nulo se ele contém um valor não nulo e lança um NoSuchElementException de outra forma.

Quando um método retorna um opcional, você deve verificar se ele contém um valor não nulo antes de solicitá-lo para o valor.

Se o método get() for chamado antes de se certificar de que ele contém um valor não nulo, um NoSuchElementException é descartado em vez de um NullPointerException.

Criar objeto opcional

A classe Optional<T> fornece três estáticas de fábrica métodos para criar objetos opcionais.

* <T> Optional<T> empty()  
  Retorna um Optional vazio.  
  O Optional<T> retornado desse método não contém um valor não nulo.
* <T> Optional<T> of(T value)  
  Retorna um Optional que contém o valor especificado como o valor não nulo.  
  Se o valor especificado for null, ele lançará um NullPointerException.
* <T> Optional<T> ofNullable(T value)  
  Retorna um Optional contendo o valor especificado se o valor for non-null.  
  Se o valor especificado for null, ele retornará um Opcional vazio.

O código a seguir mostra como criar objetos opcionais:

**import** java.util.Optional;

//w w w .java 2 s .c o m

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Optional<String> empty = Optional.empty();

System.out.println(empty);

Optional<String> str = Optional.of(**"java2s.com"**);

System.out.println(str);

String nullableString = **""**;

Optional<String> str2 = Optional.of(nullableString);

System.out.println(str2);

}

}

O código acima gera o seguinte resultado.

Uma imagem contendo Texto

Descrição gerada automaticamente

Exemplo 2

O código a seguir imprime o valor em um opcional se ele contiver um valor não nulo:

**import** java.util.Optional;

/\* w w w . j av a 2 s .c om\*/

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Optional<String> str = Optional.of(**"java2s.com"**);

**if** (str.isPresent()) {

String value = str.get();

System.out.println(**"Optional contains "** + value);

} **else** {

System.out.println(**"Optional is empty."**);

}

}

}

O código acima gera o seguinte resultado.



Opcional ifPresent

ifPresent(Consumer<? super T> action) método da classe Optional executa uma ação no valor contido no Optional.

Se o opcional estiver vazio, esse método não fará nada.

O código a seguir imprime o conteúdo de Opcional.

**import** java.util.Optional;

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Optional<String> str = Optional.of(**"java2s.com"**);

str.ifPresent(value -> System.out.println(**"Optional contains "** + value));

}

}

O código acima gera o seguinte resultado.



Se o opcional estivesse vazio, o código não imprimiria nada.

Valor opcional

A seguir estão quatro métodos para obter o valor de um opcional:

* T get()  
  Retorna o valor contido no Opcional. Se o Opcional estiver vazio, ele lançará um NoSuchElementException.
* T orElse(T defaultValue)  
  Retorna o valor contido no Opcional.  
  Se o Optional estiver vazio, ele retornará o defaultValue especificado.
* T orElseGet(Supplier<? estende T> defaultSupplier)  
  Devolve o valor contido no Opcional.  
  Se o Opcional estiver vazio, ele retornará o valor retornado do defaultSupplier especificado.
* <X estende Throwable> T ou ElseThrow(Supplier<? estende X> exceptionSupplier) lança X estende Throwable  
  Retorna o valor contido no Opcional.  
  Se o Opcional estiver vazio, ele lançará a exceção retornada do exceptionSupplier especificado.

OptionalInt, OptionalLong e OptionalDouble lidam com valores primitivos opcionais.

* O método getAsInt() da classe OptionalInt retorna o valor int.
* O método getAsLong() da classe OptionalLong retorna o valor long.
* O método getAsDouble() da classe OptionalDouble retorna o valor duplo.

Os getters para classes opcionais primitivas também lançar um NoSuchElementException quando eles estão vazios.

O código a seguir mostra como usar o OptionalInt classe:

**import** java.util.OptionalInt;

/\*from w ww. jav a 2s.c o m\*/

**public** **class** Main {

**public** **static** **void** main(String[] args) {

OptionalInt number = OptionalInt.of(2);

**if** (number.isPresent()) {

**int** value = number.getAsInt();

System.out.println(**"Number is "** + value);

} **else** {

System.out.println(**"Number is absent."**);

}

}

}

O código acima gera o seguinte resultado.



Valor opcional de fluxo

Algumas operações de fluxo retornam opcional. Por exemplo, o método max() em todas as classes de fluxo retorna um objeto opcional. Ao usar o método max() em um fluxo vazio, o valor de retorno é um objeto opcional sem nada dentro.

Usando a classe Optional, podemos manipular o valor de retorno desses métodos normalmente.

O código a seguir mostra como usar Objeto opcional retornou o formulário max().

**import** java.util.OptionalInt;

**import** java.util.stream.IntStream;

//from w w w .j a v a 2 s. co m

**public** **class** Main {

**public** **static** **void** main(String[] args) {

OptionalInt maxOdd = IntStream.of(10, 20, 30).filter(n -> n % 2 == 1).max();

**if** (maxOdd.isPresent()) {

**int** value = maxOdd.getAsInt();

System.out.println(**"Maximum odd integer is "** + value);

} **else** {

System.out.println(**"Stream is empty."**);

}

}

}

O código acima gera o seguinte resultado.



# Java Streams - Criação de Stream Java

* [« Anterior](http://www.java2s.com/Tutorials/Java/Java_Stream/0035__Java_Optional.htm)
* [Seguinte »](http://www.java2s.com/Tutorials/Java/Java_Stream/0050__Java_Stream_From_Functions.htm)

Novos métodos foram adicionados às bibliotecas Java para retornar um fluxo.

Podemos criar fluxo das seguintes maneiras.

* Criar fluxos a partir de valores
* Criar fluxos a partir de fluxos vazios
* Criar fluxos a partir de funções
* Criar fluxos a partir de matrizes
* Criar fluxos a partir de coleções
* Criar fluxos a partir de arquivos
* Criar fluxos de outras fontes

Nas seções a seguir, aprenderemos como criar fluxos.

## Criar fluxos a partir de valores

Podemos usar a partir da interface Stream para criar um Stream sequencial a partir de um único valor e vários valores . of()

<T> Stream<T> of(T t)

<T> Stream<T> of(T...values)

A partir da assinatura do método, podemos ver que o primeiro método of() cria um fluxo a partir de um único valor enquanto o segundo método of() cria um fluxo de parâmetros de comprimento variados

O código a seguir cria um fluxo que contém um único valor.

**import** java.util.stream.Stream;

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Stream<String> stream = Stream.of(**"java2s.com"**);

stream.forEach(System.out::println);

}

}

O código acima gera o seguinte resultado.



## Exemplo 2

O código a seguir cria um fluxo com quatro cadeias de caracteres.

**import** java.util.stream.Stream;

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Stream<String> stream = Stream.of(**"XML"**, **"Java"**, **"CSS"**, **"SQL"**);

stream.forEach(System.out::println);

}

}

O código acima gera o seguinte resultado.



## Exemplo 3

O código a seguir cria um fluxo a partir de uma matriz de objetos.

**import** java.util.stream.Stream;

**public** **class** Main {

**public** **static** **void** main(String[] args) {

String[] names = { **"XML"**, **"Java"**, **"SQL"**, **"CSS"** };

Stream<String> stream = Stream.of(names);

stream.forEach(System.out::println);

}

}

The code above generates the following result.

Interface gráfica do usuário

Descrição gerada automaticamente com confiança média

## Stream Builder

We can use to create streams. Stream.Builder<T>

The following code creates a stream builder.

Stream.Builder<String> builder = Stream.builder();

**import** java.util.stream.Stream;

//from w ww. j a v a 2 s.c om

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Stream<String> stream = Stream.<String>builder()

.add(**"XML"**)

.add(**"Java"**)

.add(**"CSS"**)

.add(**"SQL"**)

.build();

stream.forEach(System.out::println);

}

}

The code above generates the following result.



## IntStream from range

We can use the following two methods from IntStream interfaces to create IntStream from a range of int values.

IntStream range(int start, int end)

IntStream rangeClosed(int start, int end).

They create an IntStream that contains ordered integers between the start and end.

The specified end is exclusive in the range() method and inclusive in the rangeClosed() method.

The following code uses both methods to create an IntStream having integers 1, 2, 3, 4, and 5 as their elements:

**import** java.util.stream.IntStream;

**public** **class** Main {

**public** **static** **void** main(String[] args) {

IntStream oneToFive = IntStream.range(1, 6);

**//IntStream oneToFive = IntStream.rangeClosed(1, 5);**

oneToFive.forEach(System.out::println);

}

}

Like the IntStream interface, the LongStream class also contains range() and rangeClosed() methods that takes arguments of type long and return a LongStream.

The code above generates the following result.

Texto

Descrição gerada automaticamente com confiança média

## Córregos vazios

Um fluxo vazio não tem elementos.

Podemos usar o método estático empty() da interface Stream para criar um fluxo sequencial vazio.

**import** java.util.stream.Stream;

**public** **class** Main {

**public** **static** **void** main(String[] args) {

Stream<String> stream = Stream.empty();

stream.forEach(System.out::println);

}

}

As interfaces IntStream, LongStream e DoubleStream também contêm um método estático empty() para criar um fluxo vazio de tipos primitivos.

O código a seguir cria um fluxo vazio de inteiros.

IntStream numbers = IntStream.empty();