**Capítulo DEZOITO**

Streams Paralelos

**Objetivos do Exame**

Usar Streams paralelos incluindo redução, decomposição, processos de junção, pipelines e desempenho.

**O que é um Stream Paralelo?**

Até agora, todos os exemplos desta seção usaram streams sequenciais, onde cada elemento é processado um por um.

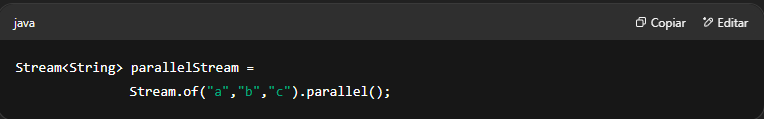
Em contraste, streams paralelos dividem o stream em várias partes. Cada parte é processada por uma thread diferente ao mesmo tempo (em paralelo).

Nos bastidores, streams paralelos usam o **Fork/Join Framework** (que revisaremos em um capítulo posterior).

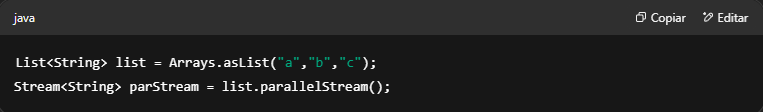
Isso significa que, por padrão, o número de threads disponíveis para processar streams paralelos é igual ao número de núcleos disponíveis do processador da sua máquina.

A vantagem de usar streams paralelos sobre o Fork/Join Framework é que eles são mais fáceis de usar.

Para criar um stream paralelo, basta usar o método parallel():

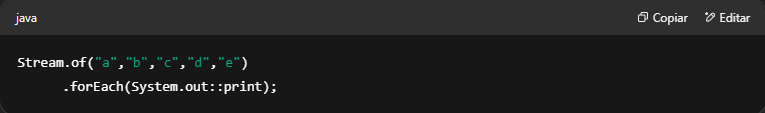


Para criar um stream paralelo a partir de uma Collection, use o método parallelStream():



**Como Funcionam os Streams Paralelos?**

Vamos começar com o exemplo mais simples:



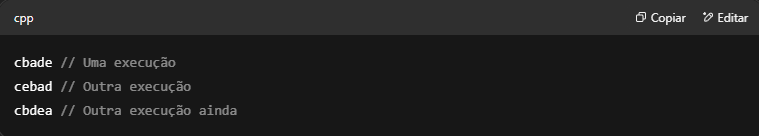
Imprimir uma lista de elementos com um stream sequencial exibirá o resultado esperado:



Contudo, ao usar um stream paralelo:



A saída pode ser diferente a cada execução:

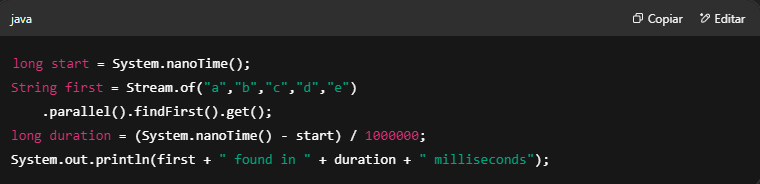


A razão é a decomposição do stream em partes e seu processamento por threads diferentes, como mencionamos antes.

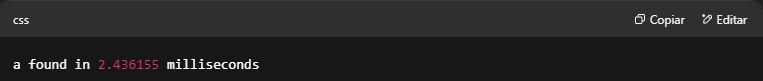
Portanto, streams paralelos são mais apropriados para operações onde a ordem de processamento não importa e que não precisam manter estado (elas são sem estado e independentes).

Um exemplo para ver essa diferença é o uso de findFirst() vs. findAny().

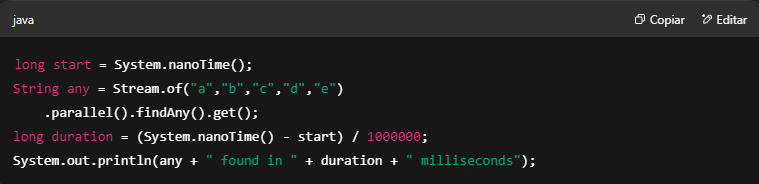
No capítulo anterior, mencionamos que o método findFirst() retorna o primeiro elemento de um stream. Mas ao usar streams paralelos e ele ser decomposto em múltiplas partes, este método precisa "saber" qual elemento é o primeiro:



Saída:



Por isso, se a ordem não importa, é melhor usar findAny() com streams paralelos:



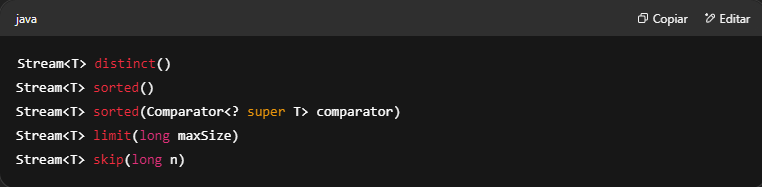
Saída:



Como um stream paralelo é processado, bem, em paralelo, é razoável acreditar que ele será processado mais rápido que um stream sequencial. Mas como você pode ver com findFirst(), isso nem sempre é o caso.

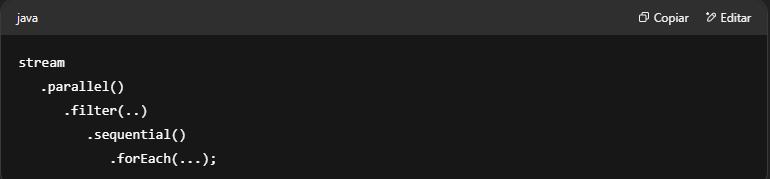
**Operações com Estado**

Operações com estado, como:

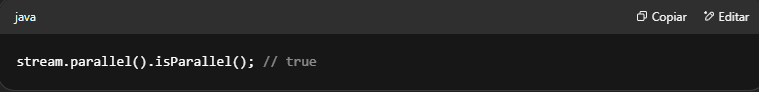


Incorporam estado de elementos processados anteriormente e podem precisar percorrer o stream inteiro para produzir um resultado, então não são adequadas para streams paralelos.

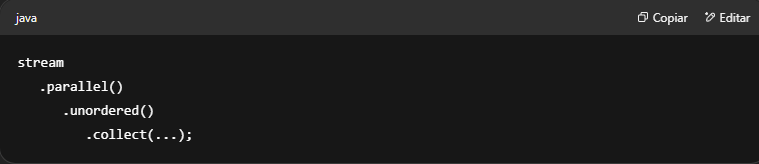
Aliás, você pode transformar um stream paralelo em um sequencial com o método sequential():



Verifique se um stream é paralelo com isParallel():



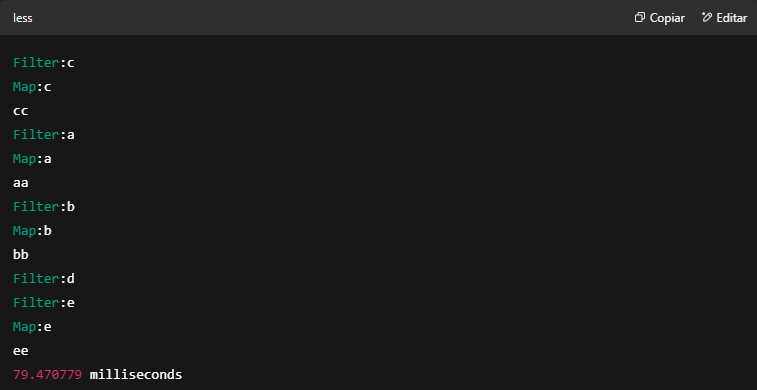
E transforme um stream ordenado em não ordenado (ou garanta que o stream seja não ordenado) com unordered():



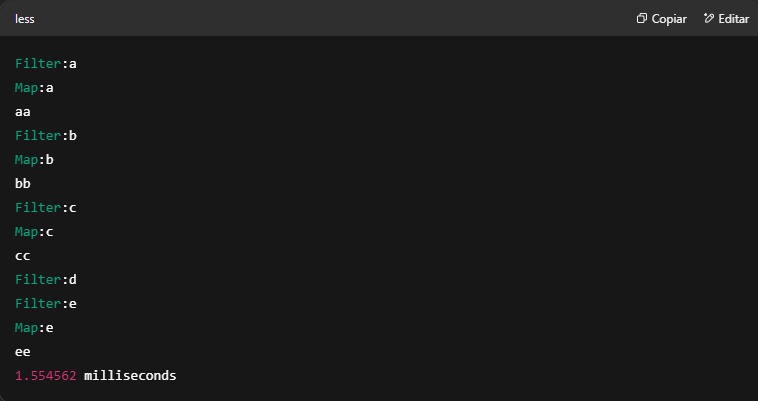
Mas não acredite que, ao executar primeiro as operações com estado e depois transformar o stream em paralelo, o desempenho será melhor em todos os casos, ou pior, toda a operação pode ser executada em paralelo, como no exemplo a seguir:



Pode-se pensar que o stream é ordenado e filtrado sequencialmente, mas a saída mostra outra coisa:

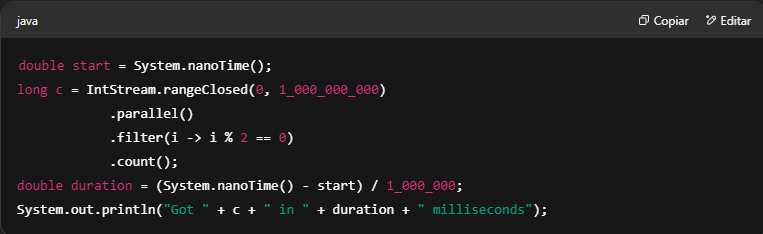


Compare com a versão sequencial (basta comentar parallel()):

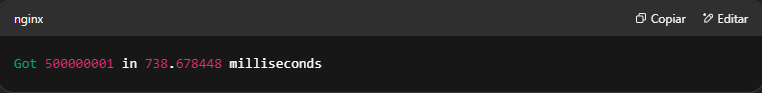


Neste caso, a versão sequencial teve melhor desempenho.

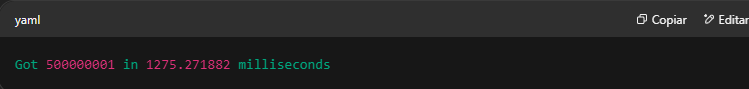
Mas se temos uma operação independente ou sem estado, onde a ordem não importa, por exemplo, contar o número de números pares em um intervalo grande, a versão paralela terá melhor desempenho:



Saída da versão paralela:



Saída da versão sequencial:



**Em resumo**

Streams paralelos nem sempre têm desempenho melhor que streams sequenciais.  
Isso, o fato de que streams paralelos processam resultados de forma independente e que a ordem não pode ser garantida são os pontos mais importantes que você precisa saber.

Mas na prática, como saber quando usar streams sequenciais ou paralelos para obter melhor desempenho?

Aqui estão algumas regras:

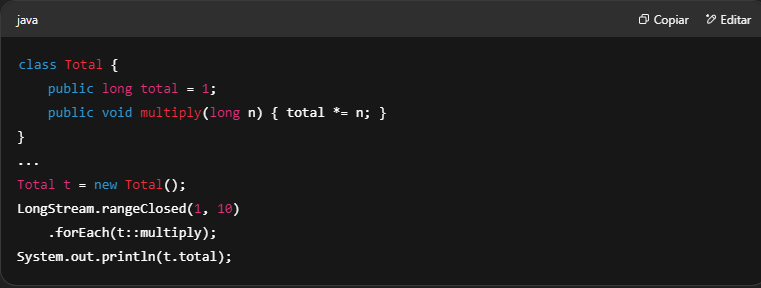
* Para um conjunto pequeno de dados, streams sequenciais são quase sempre a melhor escolha devido à sobrecarga do paralelismo.
* Ao usar streams paralelos, evite operações com estado (como sorted()) e baseadas em ordem (como findFirst()).
* Operações que são computacionalmente caras (considerando toda a operação no pipeline), geralmente têm melhor desempenho usando um stream paralelo.
* Em caso de dúvida, verifique o desempenho com um benchmark apropriado.

**Reduzindo Streams Paralelos**

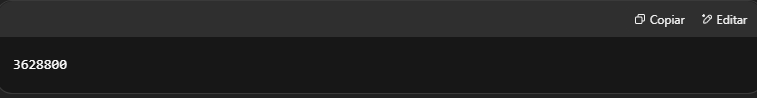
Em ambientes concorrentes, atribuições são ruins.

Isso porque, se você mutar o estado de variáveis (especialmente se elas forem compartilhadas por mais de uma thread), poderá ter muitos problemas para evitar estados inválidos.

Considere este exemplo, que implementa o fatorial de 10 de um modo bastante particular:



Aqui, estamos usando uma variável para reunir o resultado do fatorial. A saída ao executar este trecho de código é:



Contudo, ao tornarmos o stream paralelo:



Às vezes obtemos o resultado correto, e outras vezes não.

O problema é causado pelas múltiplas threads acessando a variável total simultaneamente. Sim, podemos sincronizar o acesso a essa variável (como veremos em um capítulo posterior), mas isso meio que anula o propósito do paralelismo (eu disse que atribuições são ruins em ambientes concorrentes).

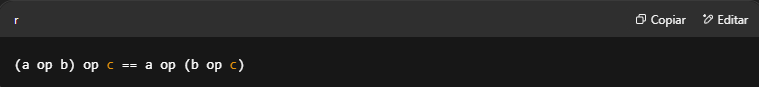
É aí que reduce() é útil.

Se você se lembra do capítulo anterior, reduce() combina os elementos de um stream em um único.

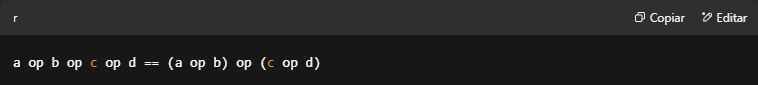
Com streams paralelos, este método cria valores intermediários e depois os combina, evitando o problema de "ordem" ao mesmo tempo que permite que os streams sejam processados em paralelo ao eliminar o estado compartilhado e mantê-lo dentro do processo de redução.

O único requisito é que a operação de redução aplicada deve ser associativa.

Isso significa que a operação op deve seguir esta igualdade:

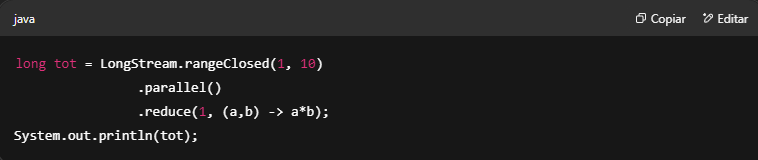


Ou:



Assim podemos avaliar (a op b) e (c op d) em paralelo.

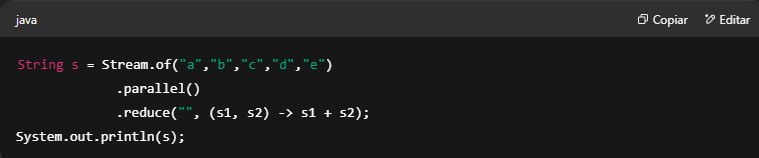
Voltando ao nosso exemplo, podemos implementá-lo usando parallel() e reduce() desta maneira:



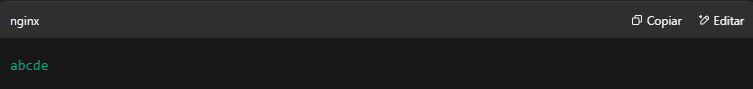
Ao executar este trecho de código, ele produz o resultado correto toda vez (3628800).

E se cronometrarmos a execução do primeiro trecho (47.216181 milissegundos) e deste último (3.094717 milissegundos), podemos ver uma grande melhoria de desempenho. Claro, esses valores (e os outros apresentados neste capítulo) dependem da potência da máquina, mas você deverá obter resultados semelhantes.

Também podemos aplicar redução ao exemplo apresentado no início deste capítulo para processar a string em paralelo mantendo a ordem:

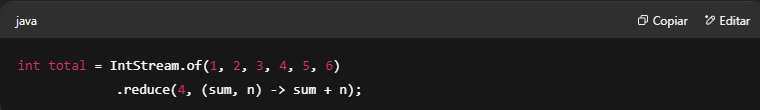


Saída:



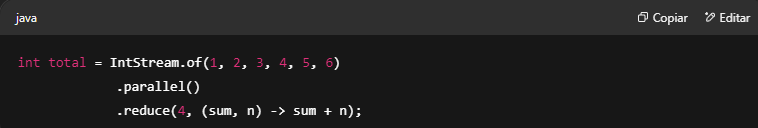
Estes são exemplos simples, mas redução é difícil de acertar às vezes, então é melhor evitar estado mutável compartilhado e usar operações sem estado e independentes para garantir que streams paralelos produzam os melhores resultados.

Por exemplo, lembra deste exemplo do capítulo anterior?



Neste exemplo, o primeiro parâmetro não é realmente uma identidade, já que 4 + n não é igual a n.

Quando tornei o stream paralelo, em vez de obter o resultado esperado (25), obtive 45:



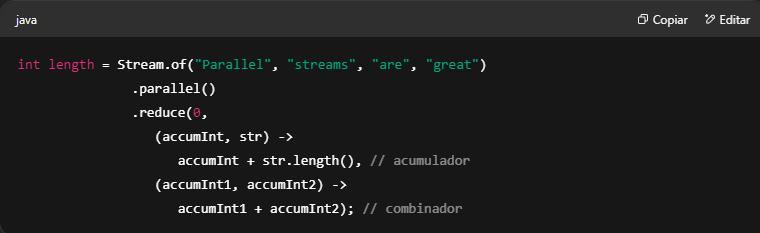
Por quê?

Porque o stream é dividido em partes, e a função acumuladora é aplicada a cada uma de forma independente, o que significa que 4 é adicionado não apenas ao primeiro elemento, mas ao primeiro elemento de **cada** parte.

**Redução com Três Argumentos**

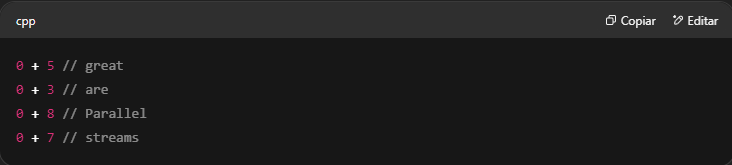
Como também vimos no capítulo anterior, a versão de reduce() que recebe três argumentos é particularmente útil em streams paralelos.

Por exemplo, se pegarmos o exemplo do capítulo anterior que soma o comprimento de algumas strings e o tornarmos paralelo:



Podemos ver o que acontece para obter o resultado (23):

1. A função acumuladora é aplicada a cada elemento sem uma ordem específica. Por exemplo:



Os resultados da função acumuladora são combinados:



Quando estamos retornando um valor reduzido do tipo U a partir de elementos do tipo T, e em um stream paralelo os elementos são divididos em N resultados intermediários do tipo U, faz sentido ter uma função que saiba como combinar esses valores U em um único resultado.

Por isso, se não estivermos usando tipos diferentes, a função acumuladora **é** a mesma que a função combinadora.

**Finalmente**

Assim como reduce(), podemos usar com segurança collect() com streams paralelos se seguirmos os mesmos requisitos de **associatividade** e **identidade**, como, por exemplo: para qualquer resultado parcialmente acumulado, combiná-lo com um recipiente de resultado vazio deve produzir um resultado equivalente.

Ou, se estivermos agrupando com a classe Collectors e a ordenação não for importante, podemos usar o método groupingByConcurrent(), a versão concorrente de groupingBy().

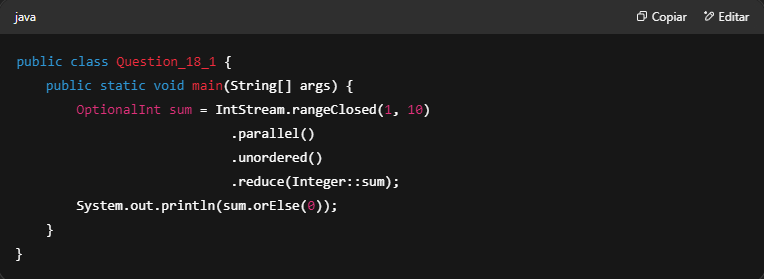
**Pontos-Chave**

* Streams paralelos dividem o stream em múltiplas partes. Cada parte é processada por uma thread diferente ao mesmo tempo (em paralelo).
* Para criar um stream paralelo a partir de outro stream, use o método parallel().
* Para criar um stream paralelo a partir de uma Collection, use o método parallelStream().
* Streams paralelos são mais apropriados para operações onde a ordem de processamento não importa e que não precisam manter estado (elas são sem estado e independentes).
* Você pode transformar um stream paralelo em sequencial com o método sequential().
* Você pode verificar se um stream é paralelo com isParallel().
* Você pode tornar um stream ordenado não ordenado (ou garantir que o stream seja não ordenado) com unordered().
* Streams paralelos **nem sempre** têm desempenho melhor do que streams sequenciais.
* Com streams paralelos, reduce() cria valores intermediários e depois os combina, evitando o problema de "ordem" e ainda permitindo que os streams sejam processados em paralelo ao eliminar o estado compartilhado (mutável) e mantê-lo dentro do processo de redução.
* O único requisito é que a operação de redução aplicada seja associativa:



**Autoavaliação**

**1. Dado:**



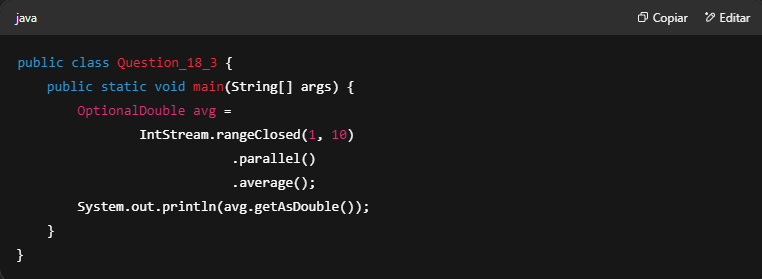
Qual é o resultado?

A. 0  
B. 55 é impresso o tempo todo  
C. Às vezes 55 é impresso  
D. Uma exceção ocorre em tempo de execução

**2. Qual das afirmações a seguir é verdadeira?**

A. Você pode chamar o método parallel() em uma Collection para criar um stream paralelo.  
B. Operações que são computacionalmente caras geralmente têm melhor desempenho usando um stream sequencial.  
C. filter() é um método sem estado.  
D. Streams paralelos sempre têm desempenho melhor que streams sequenciais.

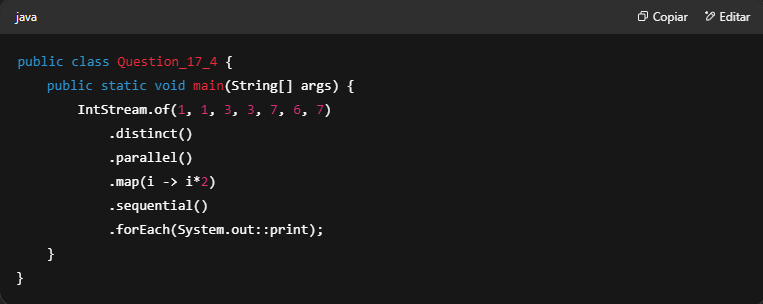
**3. Dado:**



Qual é o resultado?

A. 5.5 é impresso o tempo todo  
B. Às vezes 5.5 é impresso  
C. Falha na compilação  
D. Uma exceção ocorre em tempo de execução

**4. Dado:**



Qual é o resultado?

A. Pode imprimir 142612 às vezes  
B. 1133767  
C. Pode imprimir 3131677 às vezes  
D. 261412