**Capítulo VINTE E OITO**

**Framework Fork/Join**

**Objetivos do Exame**

Usar o framework paralelo Fork/Join.

**O Framework Fork/Join**

O framework Fork/Join é projetado para trabalhar com tarefas grandes que podem ser divididas em tarefas menores.

Isso é feito por meio de recursão, onde você continua dividindo a tarefa até encontrar o caso base, uma tarefa tão simples que pode ser resolvida diretamente, e então combina todos os resultados parciais para calcular o resultado final.

Dividir o problema é conhecido como *FORKING* e combinar os resultados é conhecido como *JOINING*.

A principal classe do framework Fork/Join é java.util.concurrent.ForkJoinPool, que na verdade é uma subclasse de ExecutorService.

Criamos uma instância de ForkJoinPool principalmente por meio de dois construtores:



A primeira versão é a forma recomendada. Ela cria uma instância com um número de threads igual ao número de processadores da sua máquina (usando o método Runtime.getRuntime().availableProcessors()).

A outra versão permite definir o número de threads que serão utilizadas.

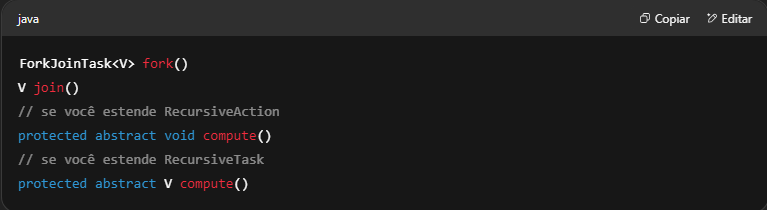
Assim como um ExecutorService executa uma tarefa representada por um Runnable ou um Callable, no framework Fork/Join uma tarefa geralmente é representada por uma subclasse de:

* RecursiveAction, que é o equivalente ao Runnable no sentido de que **NÃO** retorna um valor.
* RecursiveTask<V>, que é o equivalente ao Callable no sentido de que **retorna** um valor.

Ambas estendem da classe abstrata java.util.concurrent.ForkJoinTask.

No entanto, ao contrário das threads de trabalho que um ExecutorService usa, as threads de um ForkJoinPool usam um algoritmo de *roubo de trabalho* (*work-stealing*), o que significa que quando uma thread está livre, ela **ROUBA** o trabalho pendente de outras threads que ainda estão ocupadas com outro trabalho.

Para implementar isso, três métodos da sua classe baseada em ForkJoinTask são importantes para o framework:



E cada thread no ForkJoinPool tem uma fila dessas tarefas.

No início, você tem uma tarefa grande. Essa tarefa é dividida em (geralmente) duas tarefas menores recursivamente até que o caso base seja alcançado.

Cada vez que uma tarefa é dividida, você chama o método fork() para colocar a primeira subtarefa na fila da thread atual, e então chama o método compute() na segunda subtarefa (para calcular recursivamente o resultado).

Dessa forma, a primeira subtarefa ficará esperando na fila para ser processada ou roubada por uma thread ociosa para repetir o processo. A segunda subtarefa será processada imediatamente (também repetindo o processo).

Claro, você tem que dividir a tarefa vezes suficientes para manter todas as threads ocupadas (de preferência em um número de tarefas maior que o número de threads para garantir isso).

Tudo bem, vamos revisar isso. A primeira subtarefa está esperando na fila para ser processada, e a segunda está sendo processada imediatamente. Então quando ou como você obtém o resultado da primeira subtarefa?

Para obter o resultado da primeira subtarefa, você chama o método join() nessa primeira subtarefa.

Isso deve ser o último passo porque join() bloqueará o programa até que o resultado seja retornado.

Isso significa que a **ORDEM** na qual você chama os métodos é **IMPORTANTE**.

Se você não chamar fork() antes de join(), não haverá resultado para obter.

Se você chamar join() antes de compute(), o programa se comportará como se fosse executado em uma única thread e você estará perdendo tempo, porque enquanto a segunda subtarefa está calculando recursivamente o valor, a primeira pode ser roubada por outra thread para processá-la. Dessa forma, quando join() for finalmente chamado, o resultado já estará pronto ou você não terá que esperar muito para obtê-lo.

Mas lembre-se, o framework Fork/Join não é para toda tarefa. Você pode usá-lo para qualquer tarefa que possa ser resolvida (ou algoritmo que possa ser implementado) recursivamente, mas é melhor para tarefas que podem ser divididas em subtarefas menores **E** que podem ser computadas independentemente (ou seja, a ordem não importa).

Então vamos escolher um exemplo simples: encontrar o valor mínimo de um array. Este array pode ser dividido em vários subarrays e localizar o mínimo de cada um deles. Depois, podemos encontrar o valor mínimo entre esses valores.

**Exemplo com RecursiveAction**

Vamos codificar esse exemplo com um RecursiveAction primeiro, para ver como esse fork/join funciona. Lembre-se de que essa classe **não** retorna um resultado, então vamos apenas imprimir os resultados parciais.

Outra coisa: o cenário mais básico que podemos ter (o caso base) é quando só temos que comparar dois valores. No entanto, ter subtarefas muito pequenas não terá um bom desempenho.

Por esse motivo, ao trabalhar com fork/join, geralmente você divide os elementos em conjuntos de certo tamanho (que podem ser manipulados por uma única thread), para os quais você resolve o problema sequencialmente.

Para este exemplo, vamos processar cinco números por thread:

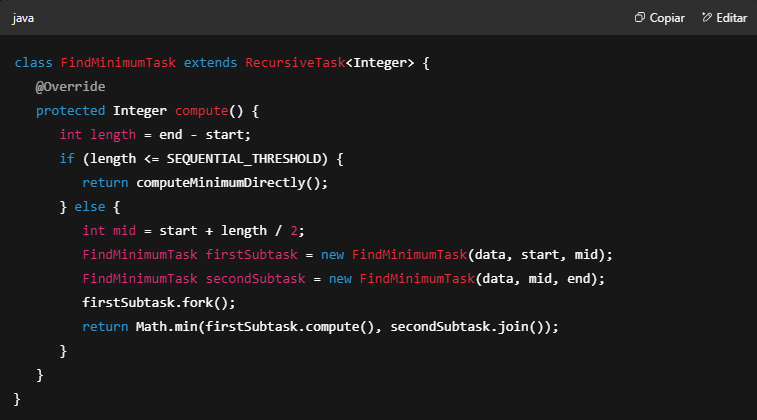


**Execução com main()**



**Versão com RecursiveTask**

Agora, vamos mudar esse exemplo para usar um RecursiveTask para que possamos retornar o valor mínimo de todos.



**Pontos-Chave**

* O framework Fork/Join é projetado para trabalhar com tarefas grandes que podem ser divididas em tarefas menores.
* Isso é feito por recursão até atingir um caso base simples, e depois combinando os resultados parciais.
* Dividir o problema = *FORKING*, combinar os resultados = *JOINING*.
* A classe principal é ForkJoinPool.
* Tarefas podem ser representadas por RecursiveAction (sem retorno) ou RecursiveTask<V> (com retorno).
* As threads usam o algoritmo de *roubo de trabalho*.
* A ordem dos métodos fork(), compute() e join() é crucial.

**Autoavaliação**

**1. Qual das seguintes afirmações é verdadeira?**  
A. RecursiveAction é uma subclasse de ForkJoinPool.  
B. Ao trabalhar com o framework Fork/Join, por padrão, uma thread por CPU é criada.  
C. Você precisa encerrar um ForkJoinPool explicitamente.  
D. fork() bloqueia o programa até que o resultado esteja pronto.

**2. Qual das seguintes é a ordem correta para chamar os métodos de uma ForkJoinTask?**  
A. compute(), fork(), join()  
B. fork(), compute(), join()  
C. join(), fork(), compute()  
D. fork(), join(), compute()

**3. Ao usar um RecursiveTask, qual das seguintes afirmações é verdadeira?**  
A. Você pode usar o método invokeAll() em vez dos métodos fork(), join() e compute().  
B. Você pode usar ExecutorService diretamente com essa classe.  
C. Uma ação é disparada quando a tarefa é concluída.  
D. ForkJoinTask.invoke() retorna o mesmo tipo que o tipo genérico de RecursiveTask.

**4. Dado:**



**O que não está certo sobre esta implementação do framework Fork/Join?**  
A. Está tudo certo, é uma implementação perfeita do framework Fork/Join.  
B. A ordem dos métodos fork(), join(), compute() não está correta.  
C. Esta implementação é muito ineficiente, as subtarefas serão muito pequenas.  
D. Não compila.