ENTENDENDO O FRAMEWORK FORK/JOIN DE JAVA

Como funciona o Java Framework Fork /Join?

Paralelismo é a execução simultânea de duas ou mais tarefas.

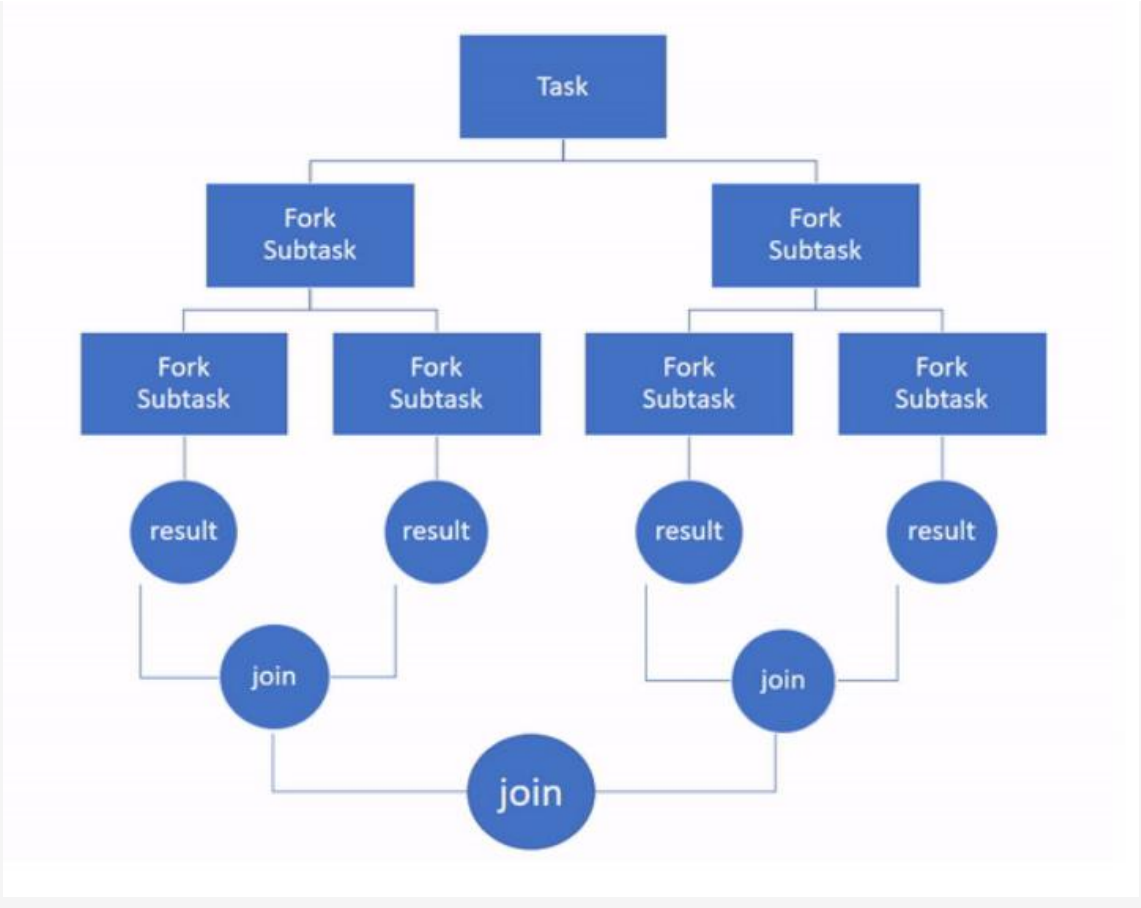
Para nossos propósitos, basta entender que, em alguns problemas, a simultaneidade faz parte do problema e, em outros, a simultaneidade não faz parte do problema, mas coisas como paralelismo podem ser parte da solução (acelerando o tempo de processamento, por exemplo).

No entanto, ao mesmo tempo, o paralelismo pode trazer problemas por conta própria.

O Framework Fork/Join foi projetada para acelerar a execução de tarefas que podem ser divididas em outras subtarefas menores, executando-as em paralelo e combinando seus resultados para obter uma única.

Por esse motivo, as subtarefas devem ser independentes umas das outras e as operações devem ser sem estado, fazendo com que esse framework não seja a melhor solução para todos os problemas.

Aplicando um princípio de divisão e conquista, o Framework, recursivamente, divide a tarefa em subtarefas menores até que um determinado limite seja atingido. Esta é a parte Fork.



Para executar as tarefas em paralelo, o Framework usa um pool de threads, que por default (por padrão), é igual ao número de processadores (núcleos) disponíveis para a Java Virtual Machine (JVM).

Cada thread tem sua própria fila de destino duplo (**deque**) para armazenar as tarefas que serão executadas.

Um **deque** é um tipo de fila que suporta a adição ou remoção de elementos da frente (cabeça) ou traseira (cauda) de uma fila. Isso permite duas coisas:

Um thread pode executar apenas uma tarefa de cada vez (a tarefa na cabeça de seu **deque**).

Um work-stealing algorithm (algoritmo de roubo de trabalho) é implementado para balancear (equilibrar) a carga de trabalho das threads.

Com esse algoritmo, as threads que ficam sem tarefas para processar podem roubar (pegar) tarefas de outras threads que ainda estão ocupadas (removendo tarefas da parte final de seu **deque**).

Essa abordagem torna o processamento mais eficiente aumentando o rendimento quando há muitas tarefas a serem processadas ou quando uma tarefa é subdividida em muitas subtarefas.

Entendendo as classes do framework

O framework Fork/Join tem duas classes principais, **`ForkJoinPool`** e **`ForkJoinTask`**.

O **`ForkJoinPool`** é uma implementação da interface **`ExecutorService`**. Em geral, os executores fornecem uma maneira mais fácil de gerenciar tarefas simultâneas. A principal característica desta implementação é o

algoritmo de roubo de trabalho (work-stealing algorithm) acima mencionado.

Há uma instância comum do ForkJoinPool disponível para todos os aplicativos que você pode obter com o método estático **`commonPool()`**:

```
1    ForkJoinPool commonPool = ForkJoinPool.commonPool();
```

O pool comum é usado por qualquer tarefa que não seja submetida explicitamente a um pool específico, como os usados por **streams** paralelos.

De acordo com a documentação da classe, o uso do pool comum normalmente reduz o uso de recursos, porque as threads num pool são recuperadas durante os períodos em que não são utilizadas e *reinstated* após o uso subsequente.

Você também pode criar sua própria instância de **`ForkJoinPool`** usando um desses construtores:

```
ForkJoinPool()
ForkJoinPool(int parallelism)
ForkJoinPool(int parallelism,
             ForkJoinPool.ForkJoinWorkerThreadFactory factory,
             Thread.UncaughtExceptionHandler handler,
             boolean asyncMode
)
```

Há outro construtor com muitos outros parâmetros para configurar, mas na maioria das vezes você usará um dos itens acima.

A primeira versão é a maneira recomendada porque cria uma instância com um número de threads igual ao número retornado pelo método:

`Runtime.getRuntime().AvailableProcessors()`,

usando padrões para todos os outros parâmetros.

Assim como um **`ExecutorService`** executa uma implementação do **Runnable** ou do **`Callable`**, a classe **`ForkJoinPool`** chama uma tarefa do tipo **`ForkJoinTask`**, que você deve implementar estendendo uma de suas duas subclasses:

- **RecursiveAction**, que representa tarefas que não produzem um valor de retorno, como um **`Runnable`**.
- **`RecursiveTask`**, que representa tarefas que geram valores de retorno, como um **`Callable`**.

Essas classes contêm o método **`compute()`**, que será responsável por resolver o problema diretamente ou pela execução da tarefa em paralelo. Na maioria das vezes, esse método é implementado de acordo com o seguinte pseudo-código:

```
if (problem is small)
  directly solve problem
else {
  split problem into independent parts
  fork new subtasks to solve each part
  join all subtasks
  compose result from subresults
}
```

As subclasses de **`ForkJoinTask`** também contêm os seguintes métodos:

- **`fork()`**, que permite que uma **`ForkJoinTask`** seja agendada para execução assíncrona (iniciando uma nova subtarefa de uma existente).

- **`join()`**, que retorna o resultado da computação quando isso é feito, permitindo que uma tarefa aguarde a conclusão de outra.

Primeiro, você tem que decidir quando o problema é pequeno o suficiente para resolver diretamente. Isso funciona como o caso base. Uma grande tarefa é dividida em 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 no *deque* da thread corrente, em seguida, chama o método **`compute()`** na segunda subtarefa para processá-la recursivamente.

Finalmente, para obter o resultado da primeira subtarefa, você chama o método **`join()`** nesta primeira subtarefa. Esse deve ser o último passo, porque **`join()`** bloqueará o próximo programa de ser processado até que o resultado seja retornado.

Assim, a ordem na qual você chama os métodos é importante. Se você não chamar **`fork()`** antes de **`join()`**, não haverá nenhum resultado para recuperar. Se você chamar **`join()`** antes de **`compute()`**, o programa funcionará como se fosse executado em uma thread e você estará perdendo tempo.

Se você seguir a ordem correta, enquanto a segunda subtarefa estiver recursivamente calculando o valor, a primeira poderá ser roubada por outro thread para processá-lo. Desta forma, quando **`join()`** é finalmente chamado, o resultado está pronto ou você não precisa esperar muito tempo para obtê-lo.

Você também pode chamar o método **`invokeAll (ForkJoinTask <?> ... tasks)`** para bifurcar (fazer fork) e unir (fazer join) a tarefa na ordem correta.

Finalmente, para enviar uma tarefa para o pool de threads, você pode usar a tarefa

`execute(ForkJoinTask <?>)`

da seguinte maneira:

```
forkJoinPool.execute(recursiveAction);
recursiveAction.join();

// Or
forkJoinPool.execute(recursiveTask);
Object result = recursiveTask.join();
```

Como alternativa, use o método

`submit(ForkJoinTask)` (que difere apenas do método execute no qual ele retorna a tarefa enviada):

```
forkJoinPool.execute(recursiveAction).join();
// Or if a value is returned
Object result = forkJoinPool.execute(recursiveTask).join();
```

No entanto, você normalmente usará **`invoke(ForkJoinTask)`**, que executa a tarefa determinada, retornando seu resultado após a conclusão:

```
forkJoinPool.invoke(recursiveAction);
// Or if a value is returned
Object result = forkJoinPool.invoke(recursiveTask);
```

Agora vamos rever um exemplo para entender melhor esse Framework.