Implementando uma demonstração

Agora vamos rever um exemplo para entender melhor essa estrutura:

https://www.pluralsight.com/guides/introduction-to-the-fork-join-framework

Vamos implementar algo simples, como encontrar a soma de todos os elementos em uma lista.

Essa lista pode ser dividida em várias sub-listas para somar os elementos de cada uma. Então, podemos encontrar soma de todos esses valores.

Vamos implementar isso como **`RecursiveAction`**, primeiro. No entanto, como essa classe não retorna os resultados parciais, apenas os imprimiremos.

Primeiro, vamos criar uma classe que se estenda da **`RecursiveAction`**:

```
1   public class SumAction extends RecursiveAction {
2
3   }
```

Em seguida, vamos escolher um valor que indique se a tarefa é processada sequencialmente ou em paralelo.

O caso mais básico é quando temos uma lista de dois valores. No entanto, ter subtarefas que são muito pequenas pode ter um impacto negativo no desempenho, pois o excesso de criação cria um custo indireto significativo por meio da pilha recursiva.

Por esse motivo, temos que escolher um valor que represente o número de elementos que podem ser processados

sequencialmente sem nenhum problema. Um valor nem pequeno nem grande demais.

Para este exemplo simples, digamos que uma lista de cinco elementos seja o limite certo:

```
1  public class SumAction extends RecursiveAction {
2    private static final int SEQUENTIAL_THRESHOLD = 5;
3  }
```

Como o método **`compute()`** não aceita parâmetros, você precisa passar para o construtor da classe os dados para trabalhar e salvá-los como uma variável de instância:

```
1  public class SumAction extends RecursiveAction {
2    // ...
3
4    private List<Long> data;
5
6    public SumAction(List<Long> data) {
7      this.data = data;
8    }
9  }
```

Para cada chamada recursiva, podemos criar uma sublista sem ter que criar uma nova lista toda vez (lembre-se de que o método **`sublist`** retorna uma subvisualização da lista original e não uma cópia). Se estivéssemos trabalhando com matrizes, provavelmente seria melhor transmitir toda a matriz e o índice inicial e final em vez de criar cópias menores da matriz original a cada vez.

Então, o método **`compute()`** se parece com isto:

```
public class SumAction extends RecursiveAction {
  // ...
```

```
  @Override
  protected void compute() {
    if (data.size() <= SEQUENTIAL_THRESHOLD) { // base case
      long sum = computeSumDirectly();
      System.out.format("Sum of %s: %d\n", data.toString(), sum);
    } else { // recursive case
      // Calcuate new range
      int mid = data.size() / 2;
      SumAction firstSubtask =
          new SumAction(data.subList(0, mid));
      SumAction secondSubtask =
          new SumAction(data.subList(mid, data.size()));

      firstSubtask.fork(); // queue the first task
      secondSubtask.compute(); // compute the second task
      firstSubtask.join(); // wait for the first task result

      // Or simply call
      //invokeAll(firstSubtask, secondSubtask);
    }
  }
}
```

Se o tamanho da lista for igual ou menor que o limite, a soma é calculada diretamente e o resultado é impresso.

Caso contrário, a lista é dividida em duas tarefas **`SumAction`**. Em seguida, a primeira tarefa é bifurcada enquanto o resultado da segunda é calculada (essa é a chamada recursiva até que a

condição do caso base seja atendida) e, depois disso, aguardamos o resultado da primeira tarefa.

O método para calcular a soma pode ser tão simples quanto:

```
public class SumAction extends RecursiveAction {
  // ...

  private long computeSumDirectly() {
    long sum = 0;
    for (Long l: data) {
      sum += l;
    }
    return sum;
  }
}
```

Finalmente, vamos adicionar um método principal para executar a classe:

```
public class SumAction extends RecursiveAction {
  // ...

  public static void main(String[] args) {
    Random random = new Random();

    List<Long> data = random
        .longs(10, 1, 5)
        .boxed()
        .collect(toList());

    ForkJoinPool pool = new ForkJoinPool();
    SumAction task = new SumAction(data);
    pool.invoke(task);
  }
}
```

Nesse método, uma lista de 10 números aleatórios de 1 a 4 (o terceiro parâmetro do método **`longs`** representa o limite exclusivo do intervalo) é gerada e passada para uma instância **`SumAction`**, que por sua vez é passada para uma nova instância **`ForkJoinPool`** para ser executado.

Se executarmos o programa, isso pode ser uma saída possível:

```
Sum of [1, 4, 4, 2, 3]: 14
Sum of [4, 1, 2, 1, 1]: 9
```

Entretanto, dividir a tarefa nem sempre resulta em subtarefas distribuídas uniformemente. Por exemplo, se tentarmos com uma lista de onze elementos, isso pode ser uma saída possível:

```
Sum of [1, 2, 2]: 5
Sum of [3, 3, 2]: 8
Sum of [2, 4, 1, 3, 3]: 13
```

==========

Agora, vamos criar uma versão dessa classe que se estende de **`RecursiveTask`** e retorna a soma de todos os elementos:

```
public class SumTask extends RecursiveTask<Long> {

}
```

Podemos copiar as variáveis da instância, o construtor e o método **`computeSumDirectly()`**:

```
public class SumTask extends RecursiveTask<Long> {
  private static final int SEQUENTIAL_THRESHOLD = 5;

  private List<Long> data;

  public SumTask(List<Long> data) {
    this.data = data;
  }

  // ...

  private long computeSumDirectly() {
    long sum = 0;
    for (Long l: data) {
      sum += l;
    }
    return sum;
  }
}
```

Alterando o método **`compute()`** um pouco para retornar o valor da soma:

```
public class SumTask extends RecursiveTask<Long> {
  // ...

  @Override
  protected Long compute() {
    if (data.size() <= SEQUENTIAL_THRESHOLD) { // base case
      long sum = computeSumDirectly();
      System.out.format("Sum of %s: %d\n", data.toString(), sum);
      return sum;
    } else { // recursive case
      // Calcuate new range
      int mid = data.size() / 2;
      SumTask firstSubtask =
          new SumTask(data.subList(0, mid));
      SumTask secondSubtask =
          new SumTask(data.subList(mid, data.size()));
```

```
      // queue the first task
      firstSubtask.fork();

      // Return the sum of all subtasks
      return secondSubtask.compute()
              +
              firstSubtask.join();
    }
  }

  // ...
}
```

Em seu método main (), precisamos apenas imprimir o valor retornado da tarefa:

```java
public class SumTask extends RecursiveTask<Long> {
  // ...

  public static void main(String[] args) {
    Random random = new Random();

    List<Long> data = random
        .longs(10, 1, 5)
        .boxed()
        .collect(toList());

    ForkJoinPool pool = new ForkJoinPool();
    SumTask task = new SumTask(data);
    System.out.println("Sum: " + pool.invoke(task));
  }
}
```

Quando executamos o programa, o seguinte pode ser uma saída possível:

```
Sum of [4, 3, 1, 1, 1]: 10
Sum of [1, 1, 1, 2, 1]: 6
Sum: 16
```

Demonstração Completa

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

}

==========

Resolvendo com RecursiveAction

```
public class SumAction extends RecursiveAction {
  private static final int SEQUENTIAL_THRESHOLD = 5;
  private List<Long> data;
  public SumAction(List<Long> data) {
    this.data = data;
  }
  @Override
  protected void compute() {
    if (data.size() <= SEQUENTIAL_THRESHOLD) { // base case
      long sum = computeSumDirectly();
      System.out.format("Sum of %s: %d\n", data.toString(),
sum);
    } else { // recursive case
      // Calcuate new range
      int mid = data.size() / 2;
      SumAction firstSubtask =
          new SumAction(data.subList(0, mid));
      SumAction secondSubtask =
          new SumAction(data.subList(mid, data.size()));
      firstSubtask.fork(); // queue the first task
      secondSubtask.compute(); // compute the second task
      firstSubtask.join(); // wait for the first task
result

      // Or simply call
```

```
      //invokeAll(firstSubtask, secondSubtask);
    }
  }
  private long computeSumDirectly() {
    long sum = 0;
    for (Long l: data) {
      sum += l;
    }
    return sum;
  }
  public static void main(String[] args) {
    Random random = new Random();
    List<Long> data = random
        .longs(10, 1, 5)
        .boxed()
        .collect(toList());
    ForkJoinPool pool = new ForkJoinPool();
    SumAction task = new SumAction(data);
    pool.invoke(task);
  }
}
```

Resultados Possíveis:

Sum of [1, 4, 4, 2, 3]: 14

Sum of [4, 1, 2, 1, 1]: 9

==========

Resolvendo com RecursiveTask

```
public class SumTask extends RecursiveTask<Long> {
 private static final int SEQUENTIAL_THRESHOLD = 5;
```

```
  private List<Long> data;
  public SumTask(List<Long> data) {
    this.data = data;
  }
  private long computeSumDirectly() {
    long sum = 0;
    for (Long l: data) {
      sum += l;
    }
    return sum;
  }
  @Override
  protected Long compute() {
    if (data.size() <= SEQUENTIAL_THRESHOLD) { // base case
      long sum = computeSumDirectly();
      System.out.format("Sum of %s: %d\n", data.toString(),
sum);
      return sum;
    } else { // recursive case
      // Calcuate new range
      int mid = data.size() / 2;
      SumTask firstSubtask =
          new SumTask(data.subList(0, mid));
      SumTask secondSubtask =
          new SumTask(data.subList(mid, data.size()));

      // queue the first task
      firstSubtask.fork();
      // Return the sum of all subtasks
      return secondSubtask.compute()
             +
```

```
            firstSubtask.join();
        }
    }

    public static void main(String[] args) {
        Random random = new Random();
        List<Long> data = random
            .longs(10, 1, 5)
            .boxed()
            .collect(toList());
        ForkJoinPool pool = new ForkJoinPool();
        SumTask task = new SumTask(data);
        System.out.println("Sum: " + pool.invoke(task));
    }
}
```

Resultados Possíveis:

Sum of [4, 3, 1, 1, 1]: 10

Sum of [1, 1, 1, 2, 1]: 6

Sum: 16

```
package <nome_do_seu_pacote>.fork;
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.RecursiveTask;
import java.util.concurrent.ThreadLocalRandom;
```

Teste de Desempenho

Você deve executar um teste informal para ver o desempenho da classe **`SumTask`** versus uma implementação sequencial, em uma lista de dez milhões de elementos:

```
1  public class ComparePerformance {
2
3    public static void main(String[] args) {
4      Random random = new Random();
5
6      List<Long> data = random
7          .longs(10_000_000, 1, 100)
8          .boxed()
9          .collect(toList());
10
11     testForkJoin(data);
12     //testSequentially(data);
13   }
14
15   private static void testForkJoin(List<Long> data) {
16     final long start = System.currentTimeMillis();
17
18     ForkJoinPool pool = new ForkJoinPool();
19     SumTask task = new SumTask(data);
20     pool.invoke(task);
21
22     System.out.println("Executed with fork/join in (ms): "
23   }
24
25   private static void testSequentially(List<Long> data) {
26     final long start = System.currentTimeMillis();
27
28     long sum = 0;
29     for (Long l: data) {
30       sum += l;
31     }
32
33     System.out.println("Executed sequentially in (ms): " +
34   }
35 }
```

Sabemos que a validade de um teste como este é questionável (usando **`System.currentTimeMillis()`** para medir o tempo de execução). Dependendo do hardware que você está testando, você pode obter resultados diferentes.

No entanto, não precisamos nos preocupar muito com os números atuais porque como eles se comparam uns com os outros é muito mais interessante !

1. **Execute o programa dez vezes para cada teste para obter um tempo médio de execução.**

O **primeiro teste**, você pode ver o Framework Fork/Join, usando um limite *threshold* de 1000 elementos, depois de um threshold de 100.000 e, em seguida, um threshold de 1 milhão de elementos.

O **segundo teste**, você pode ver a implementação sequencial.

Aqui estão os resultados:

	ForkJoin 1,000 TH	ForkJoin 100,000 TH	ForkJoin 1,000,000 TH	Sequentially
Time #01 (in ms)	36	26	34	15
Time #02 (in ms)	107	31	36	16
Time #03 (in ms)	38	25	28	16
Time #04 (in ms)	34	31	33	31
Time #05 (in ms)	32	31	29	15
Time #06 (in ms)	140	27	30	16
Time #07 (in ms)	35	26	30	32
Time #08 (in ms)	36	28	36	16
Time #09 (in ms)	34	27	29	16
Time #10 (in ms)	37	32	33	15
Average	**52.9**	**28.4**	**31.8**	**18.8**

Isso deu alguns resultados inesperados.

Primeiro, você pode ver que em uma lista de dez milhões, 1000 (mil) elementos são, na verdade, um threshold baixo, o que é devido à sobrecarga de criar muitas pequenas subtarefas.

Por outro lado, se você chegar a um milhão, conseguirá melhores tempos em média, mas o ponto ideal é mais de 100.000 (cem mil).

No entanto, a implementação mais eficiente foi a sequencial, cerca de 30% mais rápida do que a implementação Fork/Join com um limite de cem mil.

É porque a tarefa é tão simples que não precisa ser executada em paralelo?

Provavelmente. Embora uma tarefa de **soma** possa ser considerada para o algoritmo de dividir e conquistar (o tipo de algoritmo que funciona perfeitamente com o Framework Fork/Join), se a tarefa for pequena, provavelmente é melhor fazê-lo sequencialmente; no final, o custo de dividir e enfileirar as tarefas aumenta para exacerbar o tempo de execução.

Por outro lado, o método **`computeSumDirectly`** na implementação Fork/Join possui o mesmo loop da implementação sequencial para somar todos os elementos.

==========

2. **Mas, o que acontece se mudarmos a implementação sequencial para algo como o seguinte:**

```
1    private static void testSequentiallyStream(List<Long> data)
2      final long start = System.currentTimeMillis();
3
4      data.stream().reduce(0L, Long::sum);
5
6      System.out.println("Executed with a sequential stream in
7    }
```

O código parece mais elegante, mas neste caso, a execução média subiu para 241,5 ms, provavelmente devido à sobrecarga de criação do fluxo.

E quando usarmos um parallel stream (fluxo paralelo) (que usa o Framework Fork/Join sob o uso do **pool comum**):

```
private static void testParallelStream(List<Long> data) {
  final long start = System.currentTimeMillis();

  data.parallelStream().reduce(0L, Long::sum);

  System.out.println("Executed with a sequential stream in (ms): "
}
```

O tempo médio de execução que obtenho é de 181,6 ms.

Então a implementação também é importante.

Mas de volta à comparação original, é por causa do processador usado que só tem quatro núcleos e não permite um alto nível de paralelismo?

Provavelmente, isso também parece um bom motivo.

Mais núcleos - talvez dezesseis deles - poderiam funcionar melhor para o paralelismo, já que tarefas menores podem ser executadas em “pipeline” rapidamente.

No entanto, o que podemos negar é que dois aspectos muito importantes para obter um bom desempenho ao usar o Framework Fork/Join são:

- **Determinar o threshold adequado para uma tarefa Fork/Join.**

- **Determinar o nível correto de paralelismo.**

Em outras palavras, decidir se usar o Framwork Fork/Join é experimentar parâmetros diferentes.

David Hovemeyer says in his lecture about Fork/join parallelism:

> *One of the main things to consider when implementing an algorithm using fork/join parallelism is choosing the threshold which determines whether a task will execute a sequential computation rather than forking parallel sub-tasks.*
>
> *If the threshold is too large, then the program might not create enough tasks to fully take advantage of the available processors/cores.*
>
> *If the threshold is too small, then the overhead of task creation and management could become significant.*
>
> *In general, some experimentation will be necessary to find an appropriate threshold value.*

Claramente o paralelismo tem vantagens e desvantagens, e não é um bom ajuste para todas as situações.

Conclusão

Você aprendeu sobre o Framework Fork/Join em Java, que é projetado para trabalhar com grandes tarefas que podem ser recursivamente divididas em tarefas menores (a parte do Fork) para processamento e os resultados podem ser combinados no final (a parte de junção Join).

A classe principal do Framework Fork/Join é **`ForkJoinPool`**, uma subclasse de **`ExecutorService`** que implementa um algoritmo de furto de trabalho, que balanceia as filas **deque** para a execução de tarefas no pool threads. Esse algoritmo permite que uma thread furte o trabalho pendente de threads mais ocupados.

Mas, o mais importante para aproveitar esse Framework é determinar o ***threshold*** correto para executar uma tarefa em sequência ou em paralelo e **obter o nível correto de paralelismo**.

Infelizmente, não há uma fórmula concreta para identificar esse valor. Embora possamos cavar mais fundo e calcular um limite assintótico no tempo de execução para processamento paralelo, a maneira mais prática de descobrir quando usar Fork/Join é experimentar. Dependendo da tarefa e do hardware usado, haverá uma variedade de resultados que fornecerão informações sobre se o paralelismo é o caminho a ser seguido.

Neste guia, foi executado um teste de desempenho informal. No entanto, isso foi projetado para demonstrar os conceitos por trás do *framework*.

Podemos encontrar referências mais profissionais (*benchmarks*) em:

Fork/Join Framework vs. Parallel Streams vs. ExecutorService: The Ultimate Fork/Join Benchmark

se este guia despertou seu interesse.

PS: **Benchmarking** consiste no processo de busca das melhores práticas neste contexto e que conduzem ao melhor desempenho.