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

==========