10

Para saber mais: Processando comandos com ordem natural

Em algumas situações, é necessário garantir a ordem dos elementos da nossa fila com base em outros critérios. No nosso projeto, apresentamos o comando através de uma simples `string`, mas poderíamos criar uma classe para guardar mais informações sobre o comando:

```
public class Comando {

    private String tipo;
    private int prioridade;
    private String params;

    //construtor e getters

}
```

Além de `prioridade`, mais informações poderiam ser úteis, como a data de criação ou quem é o cliente. A ideia é que o cliente não envie apenas `c1` ou `c2`, mas sim um comando mais completo como, por exemplo:

```
ADD-3?curso=threads2&dataCriacao=12/06/2016&nivel=avancado
```

O comando aqui é `ADD` com a prioridade `3`, além de enviar vários parâmetros.

Repare que isso já tem muito mais a ver com um protocolo de verdade, onde a primeira parte é o *tipo do comando*, seguido pelo *número da prioridade*, seguido pelos *parâmetros*:

```
TIPO-PRIORIDADE?PARAMS
```

A primeira ideia seria usar a nossa implementação da fila `ArrayBlockingQueue` e adicionar um comando depois do outro:

```
BlockingQueue<Comando> comandos = new ArrayBlockingQueue<>(5);

// new Comando(nome_do_comando, prioridade, params)
comandos.put(new Comando("ADD", 5, "curso=threads2&dataCriacao=12/06/2016&nivel=avancada"));
comandos.put(new Comando("UPDATE", 3, "curso=threads2&dataCriacao=13/06/2016"));
comandos.put(new Comando("REMOVE", 1, "id=3"));
comandos.put(new Comando("GET", 2, "id=4"));

//imprimindo comandos
Comando comando = null;
while((comando = comandos.take()) != null) {
    System.out.println(comando.getTipo() + " - " + comando.getPrioridade());
}
```

Com a saída:

```
ADD - 5
UPDATE - 3
REMOVE - 1
GET - 2
```

Ao tentar consumir esses comandos, podemos ver que o *primeiro que entrar será o primeiro a sair (First in, first out)*. O problema é que nesse cenário *entra em jogo a prioridade*, como então podemos utilizá-la para estabelecer uma ordem de consumo para os comandos?

Para esse tipo de situação, pode ser útil conhecer um outro tipo de implementação de `BlockingQueue` : a `PriorityBlockingQueue` . Para usar essa implementação, precisamos que a classe `Comando` implemente a interface `Comparable` e por isso implemente o método `compareTo` :

```
public class Comando implements Comparable<Comando> {
    private String tipo;
    private int prioridade;
    private String params;

    public Comando(String tipo, int prioridade, String params) {
        this.tipo = tipo;
        this.prioridade = prioridade;
        this.params = params;
    }

    // construtor e getters

    @Override
    public int compareTo(Comando outroComando) {
        return outroComando.prioridade - prioridade;
    }
}
```

E no teste trocaremos a implementação `ArrayBlockingQueue` por `PriorityBlockingQueue` (repare que ela não recebe um limite no construtor como a `ArrayBlockingQueue`).

O código final deve ficar assim:

```
BlockingQueue<Comando> comandos = new PriorityBlockingQueue<>();

comandos.put(new Comando("ADD", 5, "curso=threads2&dataCriacao=12/06/2016&nivel=avancada"));
comandos.put(new Comando("UPDATE", 3, "curso=threads2&dataCriacao=13/06/2016"));
comandos.put(new Comando("REMOVE", 1, "id=3"));
comandos.put(new Comando("GET", 2, "id=4"));

Comando comando = null;
while((comando = comandos.take()) != null) {
    System.out.println(comando.getTipo() + " - " + comando.getPrioridade());
}
```

E portanto na saída:

```
ADD - 5
UPDATE - 3
GET - 2
REMOVE - 1
```