01

Threads manipulando listas

Transcrição

Bem vindo de volta ao treinamento! No último capítulo, aprendemos o que pode acontecer quando dois *threads* acessam um mesmo objeto ao mesmo tempo. Como exemplo, utilizamos uma situação em que mais de um convidado tenta acessar um banheiro - o que deu errado quando não sincronizamos o acesso.

Agora, utilizaremos um exemplo um pouco mais real.

Nos métodos da classe `Banheiro`, criamos apenas um `System.out.println()`, que funciona didaticamente. Agora, faremos com que os dois *threads* manipulem uma lista, que é um objeto mais sofisticado.

Dentro do projeto "*threads*", criaremos um pacote `br.com.alura.lista`. Nele, criaremos a classe `Lista`, que terá a seguinte implementação:

```
public class Lista {
    private String[] elementos = new String[100];
    private int indice = 0;

    public void adicionaElementos(String elemento) {
        this.elementos[indice] = elemento;
        this.indice++;
    }

    public int tamanho() {
        return this.elementos.length;
    }

    public String pegaElemento(int posicao) {
        return this.elementos[posicao];
    }
}
```

Com ela, estamos simulando uma classe "lista" do java (`java.until.List`) na qual temos uma quantidade fixa de elementos (`100`), um índice que identifica a posição na lista e três métodos:

- o método `adicionaElementos()`, que recebe uma `String`, adiciona o elemento no índice atual e o incrementa;
- o método `tamanho()`, que devolve o tamanho total da lista;
- e o método `pegaElemento()`, que retorna um elemento com base na posição dele.

A seguir, criaremos os *threads* que manipularão essa lista. Para isso, no mesmo pacote, faremos uma classe `Principal` gerando um método `main()`. Aumentaremos um pouco a complexidade, em relação ao exemplo anterior, criando `10` *threads* por meio de um laço `for () {}`. Dentro dele, serão criados os *threads* com uma `TarefaAdicionarElemento()`, recebendo `lista` como parâmetro:

```
package br.com.alura.lista;

public class Principal {
```

```
    public static void main(String[] args) {

        Lista lista = new Lista();
        for (int i = 0; i < 10; i++) {

            new Thread(new TarefaAdicionarElemento(lista));

        }

    }

}
```

Para que esse código compile, teremos que criar a tarefa implementando a interface `Runnable` e gerando o construtor:

```
package br.com.alura.lista;

public class TarefaAdicionarElemento implements Runnable {

    private Lista lista;

    public TarefaAdicionarElemento(Lista lista) {

    }

    @Override
    public void run() {

    }

}
```

Aqui, criaremos outro laço, também de `0` até `9`, adicionando um elemento com `lista.adicionaElementos()`. Nessa função, vamos concatenar o nosso índice `i`:

```
@Override
public void run() {

    for (int i = 0; i < 10; i++) {
        lista.adicionaElementos(" " + i);
    }
}
```

Recapitulando: teremos uma lista com capacidade de `100` elementos, e `10` *threads* que adicionam `10` elementos cada. Para acompanharmos a execução de cada `Thread()`, adicionaremos também o parâmetro `i` à função `TarefaAdicionarElemento()`:

```
package br.com.alura.lista;

public class Principal {
    public static void main(String[] args) {
```

```
        Lista lista = new Lista();
        for (int i = 0; i < 10; i++) {

            new Thread(new TarefaAdicionarElemento(lista, i));

        }

    }

}
```

Dessa forma, teremos que ajustar nosso construtor:

```
public TarefaAdicionarElemento(Lista lista, int numeroDoThread) {
    this.lista = lista;
    this.numeroDoThread = numeroDoThread;
}
```

Por fim, resolveremos nossa concatenação, tornando as saídas mais claras:

```
@Override
public void run() {

    for (int i = 0; i < 10; i++) {
        lista.adicionaElementos("Thread " + numeroDoThread + " - " + i);
    }
}
```

Ainda precisamos inicializar o `Thread()`. Para isso, criaremos uma nova variável `thread` e chamaremos `thread.start()`:

```
package br.com.alura.lista;

public class Principal {
    public static void main(String[] args) {

        Lista lista = new Lista();
        for (int i = 0; i < 10; i++) {

            Thread thread = new Thread(new TarefaAdicionarElemento(lista, i));
            thread.start();
        }

    }

}
```

Na verdade, poderíamos realizar essa inicialização em uma única linha, evitando a criação de uma variável que só seria válida dentro do laço:

```
package br.com.alura.lista;

public class Principal {
    public static void main(String[] args) {

        Lista lista = new Lista();
        for (int i = 0; i < 10; i++) {

            new Thread(new TarefaAdicionarElemento(lista, i)).start()
        }

    }

}
```

Terminaremos a construção da nossa classe adicionando um `sleep()` de `2` segundos e criando um laço para imprimir os elementos na tela:

```
package br.com.alura.lista;

public class Principal {
    public static void main(String[] args) throws InterruptedException '{

        Lista lista = new Lista();
        for (int i = 0; i < 10; i++) {

            new Thread(new TarefaAdicionarElemento(lista, i)).start()
        }

        Thread.sleep(2000);

        for(int i = 0; i <lista.tamanho(); i++) {
            System.out.println(lista.pegaElemento(i));
        }

    }

}
```

Clicando em *Run Java Application*, nossa aplicação será executada normalmente. Como retorno, teremos os *threads* e os respectivos elementos que eles adicionam à lista. Tudo certo, correto?

Agora, aumentaremos o tamanho da nossa lista para `1000` e pediremos para cada *thread* adicionar `100` elementos:

```
public class Lista {
    private String[] elementos = new String[1000];
    private int indice = 0;
/...


@Override
public void run() {
```

```
        for (int i = 0; i < 100; i++) {
            lista.adicionaElementos(" " + i);
        }
    }
```

Ao executarmos esse código, teremos, além das adições desejadas, três instâncias de `null`. Para entendermos melhor o que ocorreu, imprimiremos também a posição:

```
package br.com.alura.lista;

public class Principal {
    public static void main(String[] args) throws InterruptedException '{

        Lista lista = new Lista();
        for (int i = 0; i < 10; i++) {

            new Thread(new TarefaAdicionarElemento(lista, i)).start()
        }

        Thread.sleep(2000);

        for(int i = 0; i <lista.tamanho(); i++) {
            System.out.println(i + " - " + lista.pegaElemento(i));
        }

    }

}
```

Executando novamente o código, continuaremos recebendo diversos nulos ao longo do retorno impresso, mas principalmente ao final dele:

```
996 - null

997 - null

998 - null

999 - null

(...)
```

Isso não faz sentido, certo? Como temos `10` *threads* adicionando `100` elementos cada um, nossa lista de `1000` elementos deveria ser corretamente preenchida. Tal situação ocorre pois vários *threads* acabam trabalhando sobre o mesmo objeto - o mesmo problema que tivemos no exemplo anterior, quando dois convidados tentavam acessar o mesmo banheiro.

Ou seja, um *thread* está tentando acessar uma posição que ainda não foi incrementada depois do acesso do *thread* anterior, e os dois acabam utilizando o mesmo índice, de modo que o novo precisa sobrescrevê-lo. Porém, ambos os *threads* acabam incrementando a posição, gerando uma posição nula (`null`) no processo.

617 - Thread 6 - 29

618 - null

619 - Thread 7 - 17

Precisamos, então, que as linhas `this.elementos[indice] = elemento` e `this.indice++` sejam executadas de uma só vez. Já sabemos como consertar isso, não é? Utilizaremos o método `synchrozed()` sobre este bloco, e a chave será a `lista` :

```
public class Lista {
    private String[] elementos = new String[100];
    private int indice = 0;

    public void adicionaElementos(String elemento) {
        synchronized (this) {
            this.elementos[indice] = elemento;
            this.indice++;
        }
    }

    public int tamanho() {
        return this.elementos.length;
    }

    public String pegaElemento(int posicao) {
        return this.elementos[posicao];
    }
}
```

Executando novamente nosso código, não receberemos mais nenhum elemento `null` , o que significa que nossos *threads* foram executados corretamente. Ainda existe outra maneira de utilizarmos o `synchronized` : na assinatura do método!

```
public class Lista {
    private String[] elementos = new String[100];
    private int indice = 0;

    public synchronized void adicionaElementos(String elemento) {
            this.elementos[indice] = elemento;
            this.indice++;
    }

    public int tamanho() {
        return this.elementos.length;
    }

    public String pegaElemento(int posicao) {
        return this.elementos[posicao];
    }
}
```

Como todas as operações estavam compreendidas pelo método `synchronized()`, estamos apenas criando um atalho. Continuando nossos estudos, começaremos a utilizar a lista padrão do Java e verificaremos se ela já é sincronizada automaticamente. Até lá!