02

Listas sincronizadas

Transcrição

O aluno atento poderia reclamar que fizemos uma implementação "caseira" da lista. No mundo Java já existem implementações muito mais sofisticadas e normalmente se usa um `ArrayList` para guardar elementos. Que tal testarmos o `ArrayList`?

Vamos substituir a nossa lista pela interface `java.util.List` e a implementação `java.util.ArrayList`. E ao invés de utilizar o método `pegaElemento`, utilizaremos o método `get` de `List`:

```
public class Principal {

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

        List<String> lista = new ArrayList<String>();

        for (int i = 0; i < 10; i++) {
            new Thread(new TarefaAdicionarElemento(lista, i)).start();
        }

        Thread.sleep(2000);

        for (int i = 0; i < lista.size(); i++) {
            System.out.println(i + " - " + lista.get(i)); //utilizando get(i)
        }
    }
}
```

Além disso, é preciso modificar a tarefa `TarefaAdicionarElementos`. Vamos receber a lista de Strings e chamar método `add(...)`:

```
public class TarefaAdicionarElemento implements Runnable {

    private List<String> lista;
    private int numeroDoThread;

    public TarefaAdicionarElemento(List<String> lista, int numeroDoThread) {
        this.lista = lista;
        this.numeroDoThread = numeroDoThread;
    }

    @Override
    public void run() {
        for (int i = 0; i < 100; i++) {
            lista.add("Thread " + numeroDoThread + " - " + i);
        }
    }
}
```

Tudo pronto para testarmos, então vamos rodar a classe `Principal`. Ao executar percebemos que aparecem também elementos nulos e até podem aparecer exceções. A classe é `ArrayList` não é *thread safe*!

Coleções thread safe

Então como podemos trabalhar de maneira *thread safe* com listas? Há duas formas de resolver o problema. A primeira é utilizar um método auxiliar da classe `java.util.Collections`:

```
List<String> lista = Collections.synchronizedList(new ArrayList<String>());
```

O método devolve uma lista já sincronizada! Ao testar e rodar o código, os nossos problemas com elementos nulos devem desaparecer.

A segunda forma é utilizar uma coleção que já foi implementada de maneira *thread safe*. A classe `Arraylist` possui um "irmão", que se dá muito bem com vários threads. Esse "irmão" se chama de `Vector`. Então podemos testar e criar uma instância da classe Vector:

```
List<String> lista = new Vector<String>();
```

Repare que a classe `Vector` implementa a interface `java.util.List`. Tudo deve estar funcionando, sem nenhum elemento nulo!

Vamos verificar a classe `Vector` e analisar o código do método `add(..)`:

```
public synchronized boolean add(E e) {
    modCount++;
    ensureCapacityHelper(elementCount + 1);
    elementData[elementCount++] = e;
    return true;
}
```

Não há surpresa. O método `add(..)` foi sincronizado, diferentemente do método `add(..)` da classe `ArrayList`:

```
public boolean add(E e) { //sem synchronized
    ensureCapacityInternal(size + 1);
    elementData[size++] = e;
    return true;
}
```

Ótimo, vimos mais um exemplo que usa um bloco sincronizado. No próximo capítulo vamos voltar ao nosso **banheiro** e abordar tópicos importantes, como `wait` e `notify`.