

## Formas de Escalonamento e Gerenciamento de um Pool de Threads em Java - Questões 1, 2 e 3 da Prova 1

Ver os enunciados dos exercícios no texto que segue, em rosa. Cada questão vale até 0,5.

Exemplificando o uso das interfaces:

**ExecutorService,**

**ScheduledExecutorService**

**ScheduledThreadPoolExecutor**

do pacote `java.util.concurrent.*` para diferentes formas de escalonamento.

### Interfaces e Classes usadas:

interface	<code>Runnable,</code>
interface	<code>Executor,</code>
interface	<code>ExecutorService,</code> para gerenciar threads em um pool de threads.
classe	<code>Executors</code> (com 's' no final é uma classe, sem 's' no final é interface).
interface	<code>ScheduledExecutorService,</code>
interface	<code>ScheduledThreadPoolExecutor.</code>

O escalonamento de threads pode ser explicado em analogia ao número de pessoas que cabem num pedacinho num lago, em geral duas pessoas, (o pool de threads) que tem de atender a um número maior de pessoas, que podem estar numa fila (diversas outras threads) e que disputam, dentro do pool, o pedacinho (equivalente ao processador), que processa um passeio pelas águas de um lago. Uma pessoa que administre os usos dos pedacinhos pelas pessoas, seria o **Scheduler**. Veja o exemplo, que pode ser executado.

Neste exemplo, **WorkThreads** são threads que são executadas como escalonadas em um pool de threads.

O pool de threads define quantas threads são escalonadas pelo processador, podendo existir um número maior de *threads requisitando execução*, do que o tamanho do pool de threads definido para escalonar.

=====

```
public class WorkerThread implements Runnable
{
    private String threadName = null;

    public WorkerThread(String threadName)
    {
        this.threadName = threadName;
    }

    public void run()
    {
        System.out.println(this.threadName + " started...");
        try
        {
            Thread.sleep(5000);
        }
        catch (InterruptedException e)
        {
            e.printStackTrace();
        }
        System.out.println(this.threadName + " ended...");
    }
}
```

**EXERCÍCIO 1 - É recomendado, como exercício, que você estude o caso seguinte e execute um programa (pode ser escolhido na Internet) para se certificar do funcionamento da interface denominada `ExecutorService` .**

=====

Pacotes a serem usados no que segue, para testar o uso de `ExecutorService` (Caso 1), `ScheduledExecutorService` (Caso 2) ou `ScheduledThreadPoolExecutor` (Caso 3):

```
import java.util.concurrent.Executors;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledThreadPoolExecutor;
```

```
import java.util.concurrent.TimeUnit;
```

```
=====
public class Main
{
    public static void main(String args[] )
    {
        /* define o tamanho do pool */

        int PoolSize = 2;

    /**
     * Caso 1 - Esta classe mostra o uso de ExecutorService
     */

        /* Um pool de threads de tamanho PoolSize é criado com ExecutorService*/
        ExecuteService te = Executors.newFixedThreadPool(PoolSize);

        /* Um pool de Threads de tamanho PoolSize fixado é criado com ExecutorService.
         * Threads são utilizadas pelo objeto te ( threadExecutor) para executar os Runnablees
         * (ou seja, o código no método run() das threads, que serão executadas nas threads
         * (WorkThreads) criadas pelo ExecutorService).
         *
         * Se o método execute for chamado e todas as threads em ExecutorService estiverem
         * em uso (caso em que existem mais threads requisitando execução do que threads
         * no pool), a thread será colocada numa fila e atribuída no lugar da primeira thread
         * que terminar.
         */

        te.execute(new WorkThread( "WorkThread-executando-imediatamente-em-
            timesliced-default-do processador" ) ); // cria a thread e a inicia para a execução
            tornando a WorkThread ficar no estado executável (estado de pronto)

        /* As instruções Java, em vermelho,funcionam em conjunto.*/

    O método execute toma um objeto de java.lang.Runnable (uma thread)
    e a executa assincronamente.*/

    /** -----
     */

```

**EXERCÍCIO 2 - É recomendado, como exercício, que você estude o caso seguinte e execute um programa (pode ser escolhido na Internet) para se certificar do funcionamento da interface chamada **ScheduledExecutorService** .**

```
/**
```

```
 * Caso 2 - Esta classe mostra o uso de ScheduledExecutorService
```

```
 */
```

public interface [ScheduledExecutorService](#) extends [ExecutorService](#)

Um [ExecutorService](#) que pode escalonar comandos para rodar após um dado atraso ou para executar periodicamente.

O método [scheduleAtFixedRate](#) cria tarefas com vários atrasos e retorna um objeto (tarefa) que pode ser usado para cancelar ou checar uma execução. Os métodos [scheduleAtFixedRate](#), cria e executa tarefas que rodam periodicamente até serem canceladas.

[scheduleAtFixedRate\(\)](#) : Este permite programar tarefas que serão executadas primeiro, após um atraso especificado e, em seguida, serão executadas novamente com base no período especificado. Se você definir o atraso inicial de cinco segundos e, em seguida, o período subsequente de cinco segundos, em seguida, sua tarefa será executada primeiro, cinco segundos após a primeira submissão e, em seguida, irá executar periodicamente a cada cinco segundos.

---

Exemplificando uso de [ScheduledExecutorService](#) com [scheduleAtFixedRate\(\)](#) :

```
ScheduledExecutorService ste = Executors.newScheduledThreadPool\(PoolSize\);
```

```
/* Um pool de Threads de tamanho PoolSize é criado com ScheduledExecutorService.  
 * Threads são utilizadas pelo objeto ste ( scheduledThreadExecutor )  
 * para executar os Runnable (ou seja, os códigos nos métodos run() de classes  
 * que implementam a interface Runnable para implementação de threads em Java,  
 * Os códigos do método run() serão executados nas threads criadas pelo  
 * ScheduledExecutorService. Se o método scheduleAtFixedRate  
 * for chamado e todas as threads em ScheduledExecutorService estiverem em uso  
 * (caso em que existem mais threads requisitando execução do que threads no pool),  
 * o Runnable será colocado numa fila e atribuído à primeira thread que terminar.  
 */
```

```
ste.scheduleAtFixedRate (new WorkerThread ("WorkerThread-Executando-  
scheduled-At-Fixed-Rate"), 0, 5, TimeUnit.MILLISECONDS)
```

```
/*
```

```
 * Esta instrução executará uma thread requerendo execução, continuamente de 5 em  
 * 5 milissegundos, com um atraso inicial de 0 milissegundos (ou seja, sem nenhum  
 * atraso definido), para a primeira WorkerThread iniciar o ciclo de execução. Neste
```

\* caso, se a primeira `WorkThread` é completada ou não, a segunda `WorkThread`  
 \* iniciará exatamente após 5 segundos, portanto, chamada de escalonamento  
 \* em taxa fixa (`scheduleAtFixedRate`).

\* Isto continua até que 'n' threads sejam executadas no todo.  
 \* Este caso corresponde a usar **time-sliced** com um tempo definido diferente do  
 \* **tempo default do schedule**. Caso o atraso não seja preciso, o valor do  
 \* **parâmetro** deve ser zero.

\*/

```
ste.scheduleAtFixedRate (new WorkerThread ("WorkerThread-Executando-
scheduled-At-Fixed-Rate"), 10, 5, TimeUnit.MILLISECONDS);
```

/\*  
 \* Esta instrução executará uma thread requerendo execução, continuamente de 5 em  
 \* 5 milisegundos, **com um atraso inicial de 10 milisegundos**, para a primeira  
 \* `WorkerThread` iniciar o ciclo de execução. Neste caso, se a primeira `WorkThread` é  
 \* completada ou não, a segunda `WorkThread` iniciará exatamente após 5 segundos,  
 \* portanto, chamada de escalonamento em taxa fixa (`schedule at FixedRate`).

\* Isto continua até que 'n' threads sejam executadas no todo  
 \* Este caso corresponde a usar **time-sliced** com um tempo definido **diferente do**  
 \* **tempo default do processador**. Caso o atraso não seja preciso, o valor do  
 \* **parâmetro** deve ser zero, como no caso anterior.

\*/

=====

Há situações pelas quais podemos ter uma mesma tarefa (thread) repetidamente executada.  
 Veja o link [Schedule Periodic Tasks](#), que mostra o uso de `ScheduledExecutorService` :

<http://www.javapractices.com/topic/TopicAction.do?id=54>

=====

**EXERCÍCIO 3 - É recomendado, como exercício, que você estude o caso seguinte e execute um programa (pode ser escolhido na Internet) para se certificar do funcionamento de uma outra interface chamada `ScheduledThreadPoolExecutor` .**

**Você precisará do escalonamento futuro de threads:**

Usando o pacote `java.util.concurrent` para escalonamento futuro:

```
import java.util.concurrent.ScheduledFuture;
```

## interface `ScheduledFuture<V>`

onde **V** é o tipo de resultado retornado para este futuro.

**`ScheduledFuture<V>`** - Uma ação retardada que pode ser cancelada. Normalmente, um futuro agendado é o resultado da programação de uma tarefa com um **`ScheduledExecutorService`** ou **`ScheduledThreadPoolExecutor`**, uma outra interface no nível de **`ExecutorService`** e **`ScheduledExecutorService`**.

```
public interface ScheduledFuture<V> extends Delayed, Future<V>
```

```
/* -----  
*/
```

Uma outra forma executar tarefas repetidamente é usar **`ScheduledThreadPoolExecutor`** do pacote **`java.util.concurrent`**.\* :

```
/**  
* Caso 3 - Esta classe mostra o uso de ScheduledThreadPoolExecutor  
*/
```

Você pode ver o original em : <https://code latte.wordpress.com/2013/11/13/49/>

Há três maneiras pelas quais podemos ter uma tarefa (thread) repetidamente executada. Uma delas é por usar **`ScheduledThreadPoolExecutor`** do pacote **`java.util.concurrent`**.

Como sempre, devemos obter o **`ScheduledThreadPoolExecutor`**, usando um dos métodos estáticos da classe **`Executors`**. O código abaixo mostra como você pode obter o executor do pool de threads, agendado com cinco threads.

```
ScheduledThreadPoolExecutor sch = (ScheduledThreadPoolExecutor)  
Executors.newScheduledThreadPool(PoolSize);
```

Existem três métodos que analisaremos:

1. **`schedule()`** : Este permite que você programe uma thread **`Runnable`** (interface) para execução após um atraso especificado.

2. **`scheduleAtFixedRate()`** : Este permite programar tarefas que serão executadas primeiro, após um atraso especificado e, em seguida, serão executadas novamente com base no período especificado. Se você definir o atraso inicial de cinco segundos e, em seguida, o período subsequente de cinco segundos, em seguida, sua tarefa será

executada primeiro, cinco segundos após a primeira submissão e, em seguida, irá executar periodicamente a cada cinco segundos.

Em `scheduleWithFixedRate()`, se definirmos o período para cinco segundos, então isso significa que a cada cinco segundos sua tarefa será executada. Se sua tarefa leva trinta segundos para ser concluída, como pode ser reexecutada a cada cinco segundos? Bem, em tais casos, o *scheduler* agendará a tarefa para a execução e assim que a tarefa for feita com sua execução precedente, começará a executar outra vez imediatamente. Efetivamente, a taxa é reduzida. Sua tarefa, portanto, deve executar apenas duas vezes por minuto.

3. `scheduleWithFixedDelay()` : Este permite que criar tarefas que serão primeiro executadas após o atraso inicial, em seguida, com atraso dado entre o término de uma execução e início de outra execução. Portanto, se criarmos uma tarefa com atraso inicial de cinco segundos, e o atraso subsequente de cinco segundos, a tarefa será executada cinco segundos após a submissão. Quando a tarefa terminar a execução, o *scheduler* aguardará cinco segundos e, em seguida, executará a tarefa novamente.

Há uma diferença sutil que devemos entender entre as operações de `scheduleAtFixedRate()` e `scheduleWithFixedDelay()`. Vamos continuar com os nossos cinco milisegundos iniciais, cinco milisegundos subsequentes, no exemplo mencionado acima. Vamos começar com o atraso fixo porque é fácil de entender.

Suponha que tenhamos uma tarefa que faça algum trabalho na rede e demore trinta ou mais segundos para ser concluída. Em `scheduleWithFixedDelay()`, o *scheduler* aguardará a conclusão da tarefa e aguardará cinco segundos antes de executá-la novamente.

O exemplo que segue foi projetado para mostrar este caso. **É recomendado, como seu exercício**, que comente as várias seções uma-a-uma para observar as mudanças na saída.

```
1 import java.text.DateFormat;
2 import java.util.Date;
3 import java.util.concurrent.Executors;
4 import java.util.concurrent.ScheduledFuture;
5 import java.util.concurrent.ScheduledThreadPoolExecutor;
6 import java.util.concurrent.TimeUnit;
7
7 public class ScheduledExample {
```

```

8      final static DateFormat fmt = DateFormat.getTimeInstance(DateFormat.LONG);
9
10     public static void main(String[] args) {
11         // Create a scheduled thread pool with 5 core threads
12         ScheduledThreadPoolExecutor sch = (ScheduledThreadPoolExecutor)
13             Executors.newScheduledThreadPool(5);
14
15         // Create a task (thread) for one-shot execution using schedule()
16         Runnable oneShotTask = new Runnable() {
17             @Override
18             public void run() {
19                 System.out.println("\t oneShotTask Execution Time: "
20                     + fmt.format(new Date()));
21             }
22         };
23
24         // Create another task (thread)
25         Runnable delayTask = new Runnable() {
26             @Override
27             public void run() {
28                 try{
29                     System.out.println("\t delayTask Execution Time: "
30                         + fmt.format(new Date()));
31                     Thread.sleep(10 * 1000);
32                     System.out.println("\t delayTask End Time: "
33                         + fmt.format(new Date()));
34                 }catch(Exception e){
35
36                 }
37             }
38         };
39
40         // And yet another (thread)
41         Runnable periodicTask = new Runnable() {
42             @Override
43             public void run() {
44                 try{
45                     System.out.println("\t periodicTask Execution Time: "
46                         + fmt.format(new Date()));
47                     Thread.sleep(10 * 1000);
48                     System.out.println("\t periodicTask End Time: "
49                         + fmt.format(new Date()));
50                 }catch(Exception e){
51
52                 }
53             }
54         };
55
56         System.out.println("Submission Time: " + fmt.format(new Date()));
57
58         // ScheduledFuture<?> oneShotFuture = sch.schedule(oneShotTask, 5,
59             TimeUnit.SECONDS);
60         // ScheduledFuture<?> delayFuture = sch.scheduleWithFixedDelay(
61             delayTask, 5, 5, TimeUnit.SECONDS);
62         ScheduledFuture<?> periodicFuture = sch.scheduleAtFixedRate(
63             periodicTask, 5, 5, TimeUnit.SECONDS);
64     }
65 }

```



===== FIM =====