Formas de Escalonamento de Threads em Java

Exemplificando o uso das classes

ExecutorService,

ScheduledExecutorService

ScheduledThreadPoolExecutor

para diferentes formas de escalonamento.

Interfaces e Classes usadas:

interface Runnable,
interface Executor,
interface ExecutorService, para gerenciar threads em um pool de threads.
classe Executors (com *s* no final é uma classe).

O escalonamento de threads pode ser explicado em analogia ao número de pessoas que cabem num pedalinho 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 e que disputam o pedalinho (equivalente ao processador), que processa um passeio pelas águas de um lago. Uma pessoa que administre os uso dos pedalinhos 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...");
    }
}

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 Runnables
        *   (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)


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

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 Runnables (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 atribuido à primeira thread que terminar.
*/

/*
   * 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.

*/
```

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

        /* -------------------------------------------------------------------------------------------------------
         */

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

Como usar um ScheduledThreadPoolExecutor

Você pode ver o original em : **https://codelatte.wordpress.com/2013/11/13/49/**

Há três maneiras pelas quais você pode ter uma tarefa (thread) repetidamente executada; Duas delas estão usando **timers** e um delas é por usar **ScheduledThreadPoolExecutor** do pacote **java.util.concurrent**. Agora é explicado como usar o executor agendado.

Como sempre, você deve 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 principais.

```
ScheduledThreadPoolExecutor  sch = (ScheduledThreadPoolExecutor)
Executors.newScheduledThreadPool(5);
```

Precisamos lançar o valor retornado para **ScheduledThreadPoolExecutor** porque estaremos usando métodos que não estão definidos na interface ExecutorService. Esse é um capricho do sistema de herança de Java; Se você armazenar a referência de objeto em sua variável de interface, você pode apenas chamar esses métodos que são definidos pela interface.

Existem três métodos que analisaremos:

1. schedule (): Este permite que você programe um Callable ou um Runnable para execução one-shot 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.

3. scheduleWithFixedDelay (): Este permite que você crie 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 você criar uma tarefa com atraso inicial de cinco segundos, e o atraso subseqüente 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 eu quero que você entenda entre as operações de scheduleAtFixedRate () e scheduleWithFixedDelay (). Vou continuar com os nossos cinco segundos iniciais, cinco segundos subsequentes, o exemplo mencionado acima. Vamos começar com o atraso fixo porque é fácil de entender.

Suponha que você tenha 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.

Em scheduleWithFixedRate (), se você definir 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.
O exemplo que segue foi projetado para mostrar o mesmo. Eu recomendo que você comente as várias seções um por um para observar as mudanças na saída.

```
import java.text.DateFormat;
import java.util.Date;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.ScheduledThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

public class ScheduledExample {
    final static DateFormat fmt = DateFormat.getTimeInstance(DateFormat.LONG);

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

```java
        // Create a scheduled thread pool with 5 core threads
        ScheduledThreadPoolExecutor sch = (ScheduledThreadPoolExecutor)
                Executors.newScheduledThreadPool(5);

        // Create a task (thread) for one-shot execution using schedule()
        Runnable oneShotTask = new Runnable(){
            @Override
            public void run() {
                System.out.println("\t oneShotTask Execution Time: "
                            + fmt.format(new Date()));
            }
        };

        // Create another task (thread)
        Runnable delayTask = new Runnable() {
            @Override
            public void run() {
                try{
                    System.out.println("\t delayTask Execution Time: "
                            + fmt.format(new Date()));
                    Thread.sleep(10 * 1000);
                    System.out.println("\t delayTask End Time: "
                            + fmt.format(new Date()));
                }catch(Exception e){

                }
            }
        };

        // And yet another (thread)
        Runnable periodicTask = new Runnable(){
            @Override
            public void run() {
                try{
                    System.out.println("\t periodicTask Execution Time: "
                            + fmt.format(new Date()));
                    Thread.sleep(10 * 1000);
                    System.out.println("\t periodicTask End Time: "
                            + fmt.format(new Date()));
                }catch(Exception e){

                }
            }
        };

        System.out.println("Submission Time: " + fmt.format(new Date()));

//      ScheduledFuture<?> oneShotFuture = sch.schedule(oneShotTask, 5,
           TimeUnit.SECONDS);
//      ScheduledFuture<?> delayFuture = sch.scheduleWithFixedDelay(delayTask,
           5, 5, TimeUnit.SECONDS);
        ScheduledFuture<?> periodicFuture = sch.scheduleAtFixedRate(
           periodicTask, 5, 5, TimeUnit.SECONDS);
    }
}
```

java.util.concurrent

`import java.util.concurrent.ScheduledFuture;`

Interface ScheduledFuture<V>

`V` - The result type returned by this Future

ScheduledFuture<V> ???? Uma ação retardada que pode ser cancelada. Normalmente, um futuro agendado é o resultado da programação de uma tarefa com um **ScheduledExecutorService**/**ScheduledThreadPoolExecutor.**

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

==

Schedule periodic tasks

http://www.javapractices.com/topic/TopicAction.do?Id=54

==

How to stop a task in ScheduledThreadPoolExecutor once I think it's completed

Eu tenho um **ScheduledThreadPoolExecutor** com o qual posso escalonar uma tarefa para executar em uma taxa fixa. Eu quero que a tarefa seja executada com um atraso especificado para um máximo de 10 vezes até que ele "consegue". Depois disso, não quero que a tarefa seja repetida. Então, basicamente, eu vou precisar parar de executar a tarefa agendada quando eu quero que ela seja interrompida, mas sem desligar o ScheduledThreadPoolExecutor. Alguma idéia de como eu faria isso?

Veja este código e o seguinte:

```
public class ScheduledThreadPoolExecutorTest
{
```

```
  public static ScheduledThreadPoolExecutor executor = new
ScheduledThreadPoolExecutor(15);  // no multiple instances, just one to serve
all requests

  class MyTask implements Runnable
  {
    private int MAX_ATTEMPTS = 10;
    public void run()
    {
      if(++attempt <= MAX_ATTEMPTS)
      {
        doX();
        if(doXSucceeded)
        {
          //stop retrying the task anymore
        }
      }
      else
      {
        //couldn't succeed in MAX attempts, don't bother retrying anymore!
      }
    }
  }

  public void main(String[] args)
  {
    executor.scheduleAtFixedRate(new ScheduledThreadPoolExecutorTest().new
                                     MyTask(), 0, 5, TimeUnit.SECONDS);
  }
}
```

run this test, it prints 1 2 3 4 5 and stops

```
public class ScheduledThreadPoolExecutorTest {

    static ScheduledThreadPoolExecutor executor = new
           ScheduledThreadPoolExecutor(15);

    static ScheduledFuture<?> t;

    static class MyTask implements Runnable {
        private int attempt = 1;

        public void run() {
            System.out.print(attempt + " ");
            if (++attempt > 5) {
                t.cancel(false);
            }
        }
    }

    public static void main(String[] args) {
        t = executor.scheduleAtFixedRate(new MyTask(), 0, 1,
TimeUnit.SECONDS);
    }
}
```

===