UTILIZAREA FIRELOR DE EXECUȚIE ÎN JAVAFX

Cervac PETRU, student C-151

Universitatea Tehnică a Moldovei

Abstract: *Firul de execuție este cea mai mică secvență de instrucțiuni ce poate fi administrată independent de către sistemul de operare. Utilizarea firelor de execuție permite unui program să execute o parte din instrucțiunile sale în paralel. Aplicațiile contemporane, în special cele ce posedă o interfață grafică sunt nevoite să execute concomitent mai multe operații, precum reacționarea la acțiunile utilizatorului, procesarea datelor, redesenarea elementelor de pe interfață. O aplicație contemporană nu poate fi concepută fără utilizarea firelor de execuție. Sarcinile de bază ale unei biblioteci grafice contemporane includ și susținerea unor mecanisme de administrare a firelor de execuție.*

Cuvinte cheie: *Java, JavaFX, Multithreading, fir de execuție.*

1. Introducere

Ce este Multithreading?

Multithreading – proprietatea platformei (Sistemului de operarare, Mașinii Virtuale etc.) sau aplicației, care constă în faptul că procesul generat de sistemul de operare poate fi constituit din mai multe fire de execuție, ce se îndeplinesc ***paralel***, fără o ordine predefinită în timp.

Cum arată Multithreading in Java?

Multithreading-ul în **Java** a apărut încă în prima versiune a limbajului, în anul **1995**. Firele de execuție pot fi create și administrare prin intermediul moștenirii clasei ***java.lang.Thread*** sau implementarea interfeței ***java.lang.Runnable***. În **Java 5** a apărut packetul ***java.util.concurent*** care include cîteva framework-uri și clase ce oferă funcționalități utile care sunt altfel greu de implementat precum: **Concurrent Collections, Queues, Synchronizers, Executors, Locks, Atomics**.

2. Ce este JavaFX?

JavaFX este un set de packete grafice și media care permite utilizatorilor să proiceteze, să creeze, să testeze diverse aplicații client care funcționează în mod consistent pe diverse platforme. Scris ca **JavaFX**, codul unei aplicații **JavaFX** poate accesa orice alt **API** din orice bibliotecă **Java**. O aplicație **JavaFX** poate fi personalizată utilizînd **Cascading Style Sheets (CSS)** separînd astefel aspectul și stilul de implementare, astfel încît dezvoltatorii să se poată concentra asupra codării. **JavaFX** permite deasemena separarea componentei vizuale de cea comportamentală prin intermediul fișierelor **FXML**, fișiere de tip **XML** în care dezvoltatorul poate descri componentele **UI** ale aplicației.

3. JavaFX Multithreading

Graful scenei **JavaFX**, ce reprezintă **GUI** al unei aplicații **JavaFX**, nu este thread-safe și poate fi accesat și modificat doar de **JavaFX Application Thread**. Implementarea taskurilor de lungă durată, poate face aplicație **JavaFX** să nu răspundă. Pentru a evita astfel de situații, se recomandă de a utiliza unul sau mai multe fire de execuție de background, iar thread-ul aplicației **JavaFX** să se ocupe doar de procesarea evenimentelor utilizatorului.

Creînd o sarcină de fundal, într-un anumit moment avem nevoie să interacționăm cu threadul principal al aplicației **JavaFX**, fie cu un rezultat, fie cu progresul sarcinii de fundal. În acest scop, **JavaFX API** propune packetul ***javafx.concurent***, care oferă un set de instrumente pentru interacționarea sarcinelor de fundal cu **UI** al aplicației și asigură interacțiunea cu firul corect. Packetul ***javafx.concurent*** utilizează setul de librării pentru manipularea proceselor concurente, oferit de packetul ***java.util.concurent*** și le aplică pentru firul **JavaFX Application**.

Packetul ***javafx.concurent*** constă din interfața ***Worker*** și două clase de bază, ***Task*** și ***Service***, fiecare implementînd interfața ***Worker***. Interfața ***Worker*** oferă **API**-ul pentru a simplifica interacțiunea taskului de fundal cu **UI** al aplicației. Clasa ***Task*** este o implementare completă a clasei ***java.util.concurent.FutureTask***. Clasa ***Task*** permite utilizatorilor să implementeze task-uri asincrone într-o aplicație **JavaFX**. Clasa ***Service*** este concepută pentru executarea obiectelor de tip **Task**.

Clasele din pachetul javafx.concurent nu sunt singurele posibilități de creare a firelor de execuție de fundal. În JavaFx există un șir larg de clase care permit crearea unor task-uri specifice precum Transition, Animation, etc.

4. Interfața Worker

Worker este un obiect care efectuează diverse operații în unul sau mai multe fire de fundal și a cărui stare este vizibilă și disponibilă pentru aplicațiile **JavaFX**. De asemenea este utilizabil din principalul fir de execuție al aplicației **JavaFX**. Această interfață este implementată în principal de către **Task** și **Service**, oferind un **API** comun celor două clase. Un **Worker** poate să fie reutilizabil. Spre exemplu, un **Task** nu poate fi reutilizat, pe când un **Service** poate.

Un obiect de tip **Worker** are următorul ciclu de viață:

Fiecare **Worker** începe cu starea **READY**.

Cînd un **Worker** este programat de a începe, starea acestuia trece în **SHEDULED**.

Atunci cînd obiectul **Worker** își începe realizarea starea sa devine **RUNNING**.

Cînd un **Worker** își finisează execuția cu succes, starea acestuia trece în **SUCCEEDED**, iar proprietatea ***value*** este setată ca rezultatul obiectului **Worker.**

În caz contrar, dacă orice excepție este aruncată, starea **Worker**-ului trece în **FAILED**.

În orice moment de timp lucrului **Worker**-ului poate fi înrerupt, apelînd metoda cancel(), ce setează obiectul în starea **CANCELLED**.

Starea executării unui **Worker** poate fi verificată accesînd trei proprietăți diferite: ***totalWork***, ***workDone*** și ***progress***.

5. Clasa Task

Obiectele de tip **Task** sunt utulizate pentru implementarea logicii sarcinii ce trebuie îndeplinită de către un fir de execuție de fundal. Pentru a crea un **Task** este necesar de moșteni clasa **Task** și supraîncărca metoda ***call()*** care va executa sarcina de fundal și va întoace rezultatul.

Metoda ***call()*** este apelată de către thread-ul de fundal. Această metodă poate interacționa doar cu obiectele care sunt pentru citire și scriere. Manipularea cu graful scenei active din metoda ***call()*** va arunca excepții de runtime. Firul principal al aplicație **JavaFX** poate interacționa cu un obiect de tip **Task** prin intermediul proprietăților publice ale acesui obiect. Dezvoltatorul poate utiliza metotodele ***updateProgress(), updateMessage(), updateTitle()*** pentru a comunica firului principal despre schimbarea starei sarcinii.

Clasa **Task** moștenește clasa ***java.utils.concurent.FutureTask***, ce implementează interfața ***Runnable***. Din această cauză, un obiect de tip **Task** poate fi utilizat împreună cu **Executor API** pentru programarea concurentă din **Java** și deasemenea poate fi transmis unui obiect de tip **Thread** ca paramentru.

Un obiect de tip Task poate fi startat prin două modalități:

- prin startarea unui Thread cu transmiterea task-ului ca parametru

```
Thread th = new Thread(task);
th.setDaemon(true);
th.start( );
```

- prin utilizarea **ExecutorService API**

```
ExecutorService.submit(task);
```

Un obiect de tip **Task** nu poate fi reutilizabil. Pentru reutilizarea unui **Worker**, este necesară implementarea unui **Service**.

Nu există nici o metodă sigură de a opri un thread în progres. Totuși un task trebuie să fie stopat atunci cînd a fost apelată metoda ***cancel()***. Un obiect de tip **Task** trebuie să verifice periodic dacă task-ul a fost anulat, prin verificare metodei ***isCancelled()***. Mai jos este prezentat un exemplu de program care verifică periodic dacă taskul a fost anulat

```
import javafx.concurrent.Task;

Task<Integer> task = new Task<Integer>() {
   @Override protected Integer call() throws Exception {
      int it;
      for (it= 0; it < 100000; it++) {
         if (isCancelled()) break;
         System.out.println("Iteration " + iterations);
      }
      return iterations;
   }
};
```

În cazul în care taskul utilizează metode de blocare precum Thread.sleep() o excepție de tip InterruptedException este aruncată, ceea ce poate fi perceput ca un semnal pentru anularea taskului. În acest caz, taskul trebuy să arate în felul următor:

```
import javafx.concurrent.Task;

Task<Integer> task = new Task<Integer>() {
   @Override protected Integer call() throws Exception {
      int it;
      for (it = 0; it < 1000; it++) {
         if (isCancelled()) break;
         try {
            Thread.sleep(100);
         } catch (InterruptedException interrupted) {
            if (isCancelled()) {
               updateMessage("Cancelled");
```

```
                break;
            }
        }
    }
    return iterations;
  }
};
```

6. Clasa Service

Clasa Service a fost concepută pentr a executa un Task pe unul sau mai multe Thread-uri de fundal. Metodele şi stările unui Service trebuie să fie apelate doar de către firul de execuţie al aplicaţiei JavaFX. Scopul acestei clase este să ajute dezvoltatorii să implanteze interacţiunea corect între firele de execuţie de fundal şi firul de execu1ţie al aplicaţiei JavaFX. Programatorul primeşte controlul complet asupra startării, revocării şi restartării unui Service.

Utilizând clasa Service, programatorul poate vizualiza starea curentă a sarcinii de fundal şi să stopeze execuţia acesteia. La fel, programatorul poate să reseteze sau să repornească execuţia. La implementarea unei subclase a clasei Service, programatorul trebuie să fie atent să transmită parametrii necesari obiectului de tip Task ca proprietăţi ai subclasei.

Un Service poate fi executat utilizând una din metodele următoare:

- Utilizând un obiect de tip Executor, dacă este specificat
- Utilizând un fir de execuţie Daemon
- Utilizând un Executor propriu la fel ca ThreadPoolExecutor.

Mai jos este prezentat un exemplu de creare a unui Service:

```
Import javafx.concurrent.Service

public static class FirstLineService extends Service<String> {
    private StringProperty url = new SimpleStringProperty(this, "url");
    public final void setUrl(String value) { url.set(value); }
    public final String getUrl() { return url.get(); }
    public final StringProperty urlProperty() { return url; }

    protected Task createTask() {
       final String _url = getUrl();
       return new Task<String>() {
          protected String call() throws Exception {
             URL u = new URL(_url);
             BufferedReader in = new BufferedReader(new InputStreamReader(u.openStream()));
             String result = in.readLine();
             in.close();
             return result;
          }
       };
    }
 }
```

7. Modificarea Grafului Scenei

Uneori apare necesitatea ca un fir de execuţie să modifice direct graful scenei aplicaţiei JavaFX. Modificarea directă duce la generarea excepţiilor de tip runtime. Pentru acest lucru este necesară utilizarea metodei javafx.application.Platform.runLater(Runnable runnable) ce va executa obiectul de tip Runnable transmis ca parametru la firul aplicaţiei JavaFX într-un timp nedefinit.

```
Task task = new Task<Void>() {
  @Override
  public Void call() throws Exception {
    int i = 0;
    while (true) {
      final int finalI = i++;
      Platform.runLater(() -> label.setText("" + finalI));
      Thread.sleep(1000);
    }
  }
```

```
};
Thread th = new Thread(task);
th.setDaemon(true);
th.start();
```

Concluzie

Pachetul ***javafx.concurent*** oferă un set de instrumente pentru administrarea taskurilor care vor fi executate cu ajutorul firelor de execuție de fundal. Pachetul ***javafx.concurent*** nu este o alternativă pentru firele de execuție obișnuite, ci o extindere a acestora. Pachetul ***javafx.concurent*** oferă doar un mecanism de nivel înalt pentru crearea și administrarea task-urilor, lăsând crearea și inițializarea clasei **Thread**. Pentru executarea unui task, cu ajutorul clasei **Task**, sau **Service**, oricum este necesară crearea unui exemplar al clasei **Thread**.

Graful scenei aplicației **JavaFX** poate fi modificat doar de către firul de execuție principal. Deși prin intemediul metodei ***Platform.runLater(Runnable r)*** un fir de execuție poate schimba nodurile aplicație, acest lucru intră în discordanță cu scopul care a fost conceput de pachetul ***javafx.concurent***. Shimbarea concomitentă de către mai multe fire de execuție de fundal, a unui și aceluiași nod, poate să nu aducă la efectul dorit. Schimbarea grafului scenei aplicației trebuie să rămână prerogativa thread-ului principal.

Firele de execuție sunt un mecanizm puternic care permite dezvoltarea aplicațiilor care utilizează la maxim posibilitățile computatoarelor moderne. Crearea aplicațiilor cu interfață grafică este domeniul în care firele de execuție își pot demonstra posibilitățile. O aplicație cu interfață grafică modernă nu poate fi închipuită fără utilizarea firelor de execuție. În prezent, în API-ul programului Java se introduc schimbări pentru a simplifica dezvoltarea aplicațiilor utilizând firele de execuție.

Bibliografie

1. https://docs.oracle.com/javafx/2/threads/jfxpub-threads.htm
2. https://docs.oracle.com/javase/8/javafx/api/javafx/concurrent/package-summary.html
3. https://docs.oracle.com/javase/8/javafx/api/javafx/concurrent/Worker.html
4. Java: The Complete Reference, Tenth Edition by Herbert Schildt, 2014