CONCURENȚA ÎN LIMBAJUL OBIECT ORINTAT FANTOM

Autor: Truhin Alexandr
Conducător ştiinţific: lector superior Ciorbă Dumitru
Universitatea Tehnică a Moldovei
Email: bmbalex@gmail.com, dumitru.ciorba@ati.utm.md

Abstract: *Fantom este un limbaj obiect orientat cu caracteristici a limbajelor funcționale. Acest limbaj reprezintă interes sporit deoarece oferă o portabilitate înaltă şi deoarece are implementată concurența în care sunt prevăzute şi soluționate din timp cazuri marginale din concurența clasică. De asemenea acest limbaj este unul de nivel înalt, ceea ce înseamnă că multe mecaisme din interior sunt ascunse de la programator, sarcina căruia se transformă din lupta cu erorile de nivel jos,în proiectarea şi implementarea rapidă şi eficientă a soluțiilor. În acest articol sunt przentate mecanismele de bază de concurență din acest limbaj, şi este analizată aplicabilitatea metodeleor de sincronizare a firelor de execuție din concurența clasică.*

Cuvinte cheie: *Fantom,limbaj obiect orientat, limbaj funcțional, concurența, modelul actor.*

1. Limbajul obiect orientat Fantom

Fantom este un limbaj obiect orientat care poate fi rulat pe JRE (Java Runtime Environment), .NET CLR (Common Language Runtime) şi JavaScript ceea ce îi oferă portabilitatea pe majoritatea sistemelor moderne [1]. Limbajul posedă caracteristici ale limbajelor funcționale, de asemenea este concurent. Concurența se asigură utilizînd modelul actorului. Ideologia limbajului se bazează pe crearea unui limbaj de nivel înalt care poate fi folosit în producție.

Limbajul suportă cît tipizarea statică, atît şi cea dinamică. Pentru a determina care tipizare se va folosi, obiectele se accesează prin operatori diferiți. Moştenirea se face prin Mixins care sunt asemeni interfețelor din Java sau C# doar că care mai posedă şi implementarea metodelor la moştenire [2] [3].

2. Concurența în Fantom

Marea parte a limbajelor de programare utilizate în ziua de azi utilizează modelul în care toate firele din cadrul unui proces utilizează acelaşi spațiu de memorie. Aceasta poate aduce la probleme în cazul cînd nu este blocată porțiunea de memorie critică sau nu este respectată sincronizarea între fire, şi aceleaşi porțiuni de memorie pot fi accesate şi modificate simultan de diferite fire de execuție. Dar şi blocarea memoriei utilizată incorect poate aduce la interblocarea firelor. Toate aceste manipulări sunt considerate elemente a programării de nivel jos, ce îngreunează dezvoltarea soluțiilor masive şi a soluțiilor bazate pe componente compozabile dezvoltate separat [3].

Abordarea acestor probleme de către limbajul Fantom se bazează pe următoarele momente cheie [4]:

- Imuabilitatea (inflexibilitatea) este încorporată în limbaj (partajarea datelor imuabile este implementată foarte eficient, cu siguranță față de inflexibilitatea datelor);
- Imposibilitatea de a partaja date mutabile între fire de execuție;
- Comunicarea între fire este posibilă prin intermediul transferurilor de mesaje.

3. Imuabilitatea datelor

Datele (de ex. un obiect) sunt imuabile în cazul cînd este garantat că după crearea lor, starea lor va rămîne neschimbată. Aceasta permite partajarea datelor între fire de execuție fără riscul de a avea un race condition (condiție de cursă). Datele imuabile sunt partajate pe mai multe fire utilizînd referințele, astfel eficiența partajării este maximă. Imuabilitatea datelor se asigură prin două mecanisme:

- Oricare obiect care este instanța clasei *const class;*
- Rezultatul aplicării metodei *Obj.toImmutable* pentru entitățile *List*, *Map* sau *Func*.

4. Obiectele partajate

Pentru siguranța transmiterii datelor între fire, în limbajul Fantom este permis partajarea a doar două tipuri de date: date imuabile şi date serializabile. Dacă pentru cazul cu date imuabile, partajarea se face prin referințe, iar în cazul în care datele sunt mutabile, unica posibilitate de a ocoli problemele implementărilor concurente descrie în punctul 2 este de a serializa aceste date şi de a transmite o copie a datelor serializate în alt fir de execuție. Obiectele serializate sunt convenabile în cazurile obiectelor cu o structură arborescentă de nivel mare, doar că viteza lor de transmitere este relativ medie.

5. Modelul actorilor

Modelul actorilor este un model matematic de calcul concurent în care actorii sunt primitivele universale în calculele digitale concurente: ca răspuns la un mesaj primit, actorul poate face decizii locale, crea alți actori, trimite mai multe mesaje şi să determine cum să răspundă la următorul mesaj [5].

În Fantom, actorii sunt nişte obiecte ”uşoare” care se execută asincron în fond. Actorii îşi încep execuția la primirea mesajelor. Controlul asupra actorilor îl are *ActorPool*. Actorii sunt descendenți a clasei *const class*, ceea ce înseamnă că ei sunt imuabili, şi este posibilă transmiterea referințelor lor. Mesajele pot fi transmise către actor prin unul din trei metode:

- send: adaugă mesajul în coada de execuții imediat;
- sendLater: adaugă mesajul în coada de execuții peste o anumită perioadă;
- sendWhenDone: adaugă mesajul în coada de execuții cînd are loc finisarea procesării altui mesaj.

După procesarea mesajului, este returnat un obiect de tip *Future* care conține resultatul procesării.

Aşteptarea rezultatului poate fi plasată în coada de aşteptare folosing funcția *isDone,* conținutul căreia va fi executat la returnarea rezultatului în urma procesării mesajului.

De asemenea rezultatul procesării mesajului poate fi primit prin metoda *get*. Această metodă este blocantă. Pentru a nu crea blocări de fire, metoda *get* nu trebuie utilizată în cazurile *actorilor* recursivi.

6. Sincronizarea firelor de execuție

În sistemele concurente sincronizarea se face prin mai multe metode cum ar fi [7]:

- Mutex şi Multiplex;
- Randez-vous şi Bariara;
- Semnalizare.

În cazul limbajului Fantom, nu avem nevoie şi nici nu putem implementa folosind instrumentele standarte sincronizările mutex şi multiplex. Aceasta se datorează faptului că nu avem date partajate mutabile.

Sincronizarea Randez-vous şi Bariera nu sunt posibile d-ce unicul semnal pe care îl putem primi de la un fir, este semnalul de finisare a procesării mesajului şi a eliberării firului.

Este posibilă sincronizarea prin semnalizare, deoarece instanța *get* este blocantă, astfel dacă trimitem unui actor un mesaj ce conține rezultatul procesării altui mesaj de către alt actor, acest actor nu va începe procesarea mesajului pînă nu va primi rezultatul procesării primului mesaj. În caz că putem vedea un fir de execuție clasic ca o succesiune de două fire de execuție, atunci e posibil implementarea şi celorlalte metode de sincronizare, punctul de sincronizare fiind trecerea de la un fir la altul.

Bibliografie

21. *Fantom Setup*, Fantom, Verificat 15.11.2011 - http://fantom.org/doc/docTools/Setup.html
22. J. Boyland, G. Castanga, *ECOOP '96, Object-oriented Programming: 10th European Conference*, 1996, p. 16-22.
23. *WhyFantom*, Fantom, Verificat 15.11.2011 - http://fantom.org/doc/docIntro/WhyFantom.html
24. *Concurrency*, Fantom, Verificat 15.11.2011 - http://fantom.org/doc/docLang/Concurrency.html
25. *Actor Model of Computation: Scalable Robust Information Systems*, Inconsistency Robustness 2011. Stanford University. August 16-18, 2011, Verificat 15.11.2011 http://knol.google.com/k/inconsistency-robustness/inconsistency-robustness-2011/1sx0o2as3axsf/1
26. *Actors*, Fantom, Verificat 15.11.2011 - http://fantom.org/doc/docLang/Actors.html
27. Ciorbă Dumitru, *Excluderea reciprocă, sincronizarea şi semafoarele*, Verificat 15.11.2011 - http://ciorba.name/?p=65