Metode avansate de programare

Informatică Româna, 2020-2021, Curs 9,

Partea a doua: java.util.concurrent

Referinte pe care se bazeaza acest curs

- Oracle tutorials
- **http://winterbe.com/posts/2015/04/07/java8-concurrency-tutorial-thread-executor-examples/**
- **http://www.javacodegeeks.com/2015/09/java-concurrency-essentials.html**
- http://tutorials.jenkov.com/java-util-concurrent/executorservice.html
- http://stacktips.com/tutorials/java/countdownlatch-and-java-concurrency-example
- http://blogs.msmvps.com/peterritchie/2007/04/26/thread-sleep-is-a-sign-of-a-poorly-designed-program/

Paralelism si concurenta in Java incepand cu JDK 5.0

- *Versiunea JDK 5.0* a fost un pas major în programarea concurentă, astfel mașina virtuală Java a fost îmbunătățită semnificativ pentru a permite claselor să profite de suportul pentru concurență oferit la nivel hardware.
- Pachetul *java.util.concurent* aduce un set bine testat si foarte performant de funcții și structuri de date pentru concurență care ajuta programatorul, cu un effort redus de programare, sa proiecteze aplicatii concurente care:
  - au performanta ridicata,
  - sunt de incredere
  - si usor de intretinut

Paralelism si concurenta in Java incepand cu JDK 5.0

Îmbunătățirile aduse din perspectiva suportului pentru concurență sunt structurate în 3 categorii:

- **Modficări la nivelul mașinii virtuale java:** Procesoarele moderne oferă suport hardware pentru concurență, de obicei în forma unor instrucțiuni *compare-and-swap* (CAS)...
- **Clase utilitare de nivel scăzut – lacăte și variabile atomice:** De exemplu clasa *ReentrantLock* oferă functionalitate asemănătoare cu soluția *synchronized*, dar cu un control mai bun asupra blocării (timed locks, lock polling, etc.) și o mai bună scalabilitate.
- **Clase utilitare la nivel înalt:** Clase care implementează: mutexuri, semafoare, lacăte, bariere, thread pools și colecții thread-safe. Acestea sunt oferite dezvoltatorilor de aplicații pentru a construi diverse soluții.

Thread Pools

- Un mecanism clasic pentru managementul unui grup mare de task-uri este combinarea unei cozi de lucru (work *queue)* cu un set de threaduri (*thread pool)*.
- Work *queue* este o coadă de taskuri ce trebuie procesate.
- Un *thread pool* este o colecție de thread-uri care extrag sarcini (task-uri) din coada și le execută.
- Când un *worker thread* termină o sarcină, se întoarce la coadă pentru a vedea daca mai există sarcini de executat, iar dacă da, extrage sarcina din coada și o execută.

Thread Pools și Framework-ul Executor

- Atunci când este necesar să fie rulate mai multe sarcini complexe în paralel, și să se aștepte finalizarea tuturor pentru ca mai apoi să se returneze o valoare, devine destul de dificilă conceperea unui cod bun care să le sincronizeze.
- Java introduce Executor, o interfață ce permite crearea ***seturilor de thread-uri, sincronizarea și execuția lor.***

```
public interface Executor {
    void execute (Runnable command);
}
```

- Politica de execuție a task-urilor depinde de implementarea de *Executor* aleasă.
  - `Executors.newCachedThreadPool() : ThreadPoolExecutor`
  - `Executors.newFixedThreadPool(int n) : ThreadPoolExecutor`
  - `Executors.newSingleThreadExecutor()`
- Clasa ThreadPoolExecutor (implements ExecutorService) poate fi intens customizată în funcție de necesitați.

Thread Pools și Framework-ul Executor

- Un set de thread-uri poate fi reprezentat printr-o instanță a clasei ExecutorService. Acesta poate fi de mai multe tipuri:
  - Single Thread Executor – un set care conține un singur thread; codul se va executa secvențial
  - Fixed Thread Pool – un set care conține un număr fix de thread-uri; dacă un thread nu este disponibil pentru un task, acesta se pune într-o coadă și așteaptă finalizarea unui alt task
  - Cached Thread Pool – un set care creează atâtea thread-uri cîte sunt necesare pentru executarea unui task în paralel
  - Scheduled Thread Pool – un set creat pentru planificarea task-urilor viitoare
  - Single Thread Scheduled Pool – un set care conține un singur thread utilizat în planificarea task-urilor viitoare.

Crearea unui ExecutorService

- Folosind clasa Executors (Ce sablon de proiectare intalnim aici?)

```
ExecutorService executorService1 =
        Executors.newSingleThreadExecutor();

ExecutorService executorService2 =
        Executors.newFixedThreadPool(10);

ExecutorService executorService3 =
        Executors.newScheduledThreadPool(10);
```

Utilizarea unui ExecutorService

- Exista cateva modalitati prin care putem folosi un ExecutorService:

execute(Runnable)

submit(Runnable)

submit(Callable)

invokeAny(...)

invokeAll(...)

JavaFX:

class Task implements java.lang.Runnable – o unitate logica a carei stare este OBSERVABILA si disponibila aplicatiilor JavaFX

Interfata Runnable

executor.execute(Runnable)

Executor_ex1

```
ExecutorService executorService =
        Executors.newSingleThreadExecutor();
executorService.execute(new Runnable() {
    public void run() {
        System.out.println("Asynchronous task");
    }
});

System.out.println(Thread.currentThread().getName());
ExecutorService executor = Executors.newFixedThreadPool(5);
executor.execute(() -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello " + threadName);

});
executor.execute(() -> {
    String threadName = Thread.currentThread().getName();
    System.out.println("Hello " + threadName);

});

System.out.println(Thread.currentThread().getName());
executorService.shutdown();
executor.shutdown();
```

There is no way of obtaining the result of the executed Runnable, if necessary. We have to use a Callable for that…

ExecutorService never stops. We have to shut down the executor!!!

See also:
executor.awaitTermination(..), executor.shutdownNow();

Interfata Callable

Executor_ex2

- `executor.submit(Callable)`

```
Future<String> future = executor.submit(new Callable<String>(){
    public String call() throws Exception {
        Thread.sleep(5000);
        System.out.println("Asynchronous Callable");
        return "Callable Result";
    }
});

String result=future.get(); // asteptam sa obtinem rezultatul
System.out.println(result);
executor.shutdown();
```

Rezultatul Callable poate fi obținut prin intermediul obiectului Future întors.

invokeAll(Collection<Callable>))

- Executorii suportă trimiterea simultană a mai multor Callable prin intermediul metodei invokeAll (…) și returnează o listă de Future.

```
ExecutorService executor = Executors.newFixedThreadPool(5);
List<Callable<String>> callables = Arrays.asList(
        () -> "task1",
        () -> "task2",
        () -> "task3");
List<Future<String>> results=executor.invokeAll(callables);
results.stream()
        .map(future -> {
            try {
                return future.get();
            } catch (Exception e) {
                throw new IllegalStateException();
            }
        })
        .forEach(System.out::println);
```

invokeAny(Collection<Callable>))

Executor_ex4

```
Callable<String> callable(String result, long sleepSeconds) {
    return () -> {
        TimeUnit.SECONDS.sleep(sleepSeconds);
        return result;
    };
}

ExecutorService executor = Executors.newWorkStealingPool();

List<Callable<String>> callables = Arrays.asList(
        callable("task1", 2),
        callable("task2", 1),
        callable("task3", 3));

String result = executor.invokeAny(callables);
System.out.println(result);

// => task2
```

- Cel mai rapid callable

ScheduledExecutorService

- Metoda call ar trebui sa se execute dupa 5 secunde

```
ScheduledExecutorService scheduledExecutorService =
        Executors.newScheduledThreadPool(2);
ScheduledFuture scheduledFuture =
        scheduledExecutorService.schedule(new Callable() {
            public Object call() throws Exception {
                System.out.println("Executed!");
                return "Called!";
            }
        }, 5,TimeUnit.SECONDS);

scheduledExecutorService.shutdown();
```

Clase de sincronizare

- Exemple de clase de sincronizare: Semaphore, CyclicBarrier, CountdownLatch, și Exchanger
- *Semaphore:* doar un anumit nr de thread-ri pot avea simultan acces concurent la o resursă.
  - Implementează un semafor clasic, care are un număr dat de permisii ce pot fi cerute și eliberate.
  - Este folosit pentru a restricționa numărul de thread-uri ce pot avea simultan acces concurent la o resursă.
  - Înainte să obțină o resursă un thread trebuie să obțină permisiunea de la semafor – adică resursa este disponibilă.
  - Apoi, când termină de utilizat resursa respectivă, thread-ul se întoarce la semafor pentru a semnala că aceasta este din nou disponibilă.

Clase de sincronizare - cont

- Mutex - un caz special de semafor, cu o singură permisie (permite acces exclusiv)
- CyclicBarrier - oferă un ajutor de sincornizare: permite unui set de thread-uri să aștepte ca întreg setul de thread-uri să ajungă la o barieră comună.
- CountdownLatch - oarecum similar cu *CyclicBarrier* prin faptul că permite coordonarea unui grup de thread-uri. Diferența e ca atunci când un thread ajunge la barieră, nu se blochează ci doar decrementează valoarea inițiala a lacătului. Este util când o problemă este divizată între mai multe thread-uri, fiecare făcând o parte. Când un thread termină de rezolvat decrementează contorul.

Semaphore

```
ExecutorService executor = Executors.newFixedThreadPool(10);
Semaphore semaphore = new Semaphore(5);
Runnable longRunningTask = () -> {
    boolean permit = false;
    try {
        permit = semaphore.tryAcquire(1, TimeUnit.SECONDS);
        if (permit) {
            System.out.println("Semaphore acquired");
            sleep(5);
        } else {System.out.println("Could not acquire semaphore");}
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    } finally {
        if (permit) {semaphore.release();
        }}
};
IntStream.range(0, 10).forEach(i -> executor.submit(longRunningTask));
executor.shutdown();
```

Mutex

- ReentrantLock

```
ReentrantLock lock = new ReentrantLock();
int count = 0;

void increment() {
    lock.lock();
    try {
        count++;
    } finally {
        lock.unlock();
    }
}
```

CountdownLatch

```
class Waiter implements Runnable{
    CountDownLatch latch = null;
    public Waiter(CountDownLatch latch) {
        this.latch = latch;
    }
    public void run() {
        try {
            latch.await();
            //DoSomething
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Waiter
Released");
    }
}
```

```
 CountDownLatch latch = new CountDownLatch(3);

 Waiter waiter  = new Waiter(latch);
 Decrementer decrementer = new Decrementer(latch);

 new Thread(waiter).start();
 new Thread(decrementer).start();
```

```
class Decrementer implements Runnable {
    CountDownLatch latch = null;
    public Decrementer(CountDownLatch latch) {
        this.latch = latch;
    }
    public void run() {
        try {
            Thread.sleep(1000);
            this.latch.countDown();

            Thread.sleep(1000);
            this.latch.countDown();

            Thread.sleep(1000);
            this.latch.countDown();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Colecții Thread-Safe

- Framework-ul Collections introdus în JDK 1.2 este un framework flexibil pentru reprezentarea colecțiilor de obiecte, folosind interfețele de bază Map, List, Set.
- Cateva dintre implementări sunt Thread-Safe (*Hashtable, Vector*), celelalte pot fi făcute thread-safe cu ajutorul colecțiilor, și anume *Collections.synchronizedMap(), Collections.synchronizedList() si Collections.synchronizedSet().*
- Pachetul *java.util.concurrent* adăugă câteva noi colecții concurente: *ConcurrentHashMap*, *CopyOnWriteArrayList* și *CopyOnWriteArraySet*. Scopul acestor clase este să îmbunatățească performanța și scalabilitatea oferită de tipurile de colecții de bază.

Colecții Thread-Safe

- JDK 5.0 oferă de asemenea două noi structuri și interfețe pentru utilizarea cozilor : *Queue* și *BlockingQueue*.
- Iteratorii s-au schimbat de asemenea în JDK 5.0. Daca până la veriunea 5.0 nu se permitea modificarea unei colecții în timpul iterației, iteratorii introduși în JDK 5.0 oferă o vedere consistenta asupra colecției, chiar dacă aceasta se schimbă în timpul iterarii.
- *CopyOnWriteArrayList* și *CopyOnWriteArraySet* sunt versiuni îmbunătățite ale structurilor Vector și ArrayList. Îmbunătațirile sunt aduse în special la nivelul iterației. Astfel, dacă în timpul parcurgerii unui Vector sau a unui ArrayList colecția este modificată, se va arunca o excepție, iar noile clase rezolvă această problemă.

Colecții Thread-Safe

- JDK 5.0 oferă de asemenea două noi structuri și interfețe pentru utilizarea cozilor : *Queue* și *BlockingQueue*.
- Iteratorii s-au schimbat de asemenea în JDK 5.0. Daca până la veriunea 5.0 nu se permitea modificarea unei colecții în timpul iterației, iteratorii introduși în JDK 5.0 oferă o vedere consistenta asupra colecției, chiar dacă aceasta se schimbă în timpul iterarii.
- Queue: Există două implementări principale, care determină ordinea în care elementele unei cozi sunt accesate: ConcurrentLinkedQueue (acces FIFO) și PriorityQueue (acces pe bază de priorități).

Colecții Thread-Safe

- Cozi *Cozi cu blocare (BlockingQueue)*
  - Acest tip de cozi sunt folosite atunci când se dorește blocarea unui thread, în situația în care anumite operații pe o coadă nu pot fi executate. Un exemplu ar fi cazul în care consumatorii extrag mai greu din coadă informația decat ea este plasată în coadă de producatori.
  - Prin folosirea *BlockingQueue* se blochează automat producătorii până când se eliberează un element din coadă. Implementările interfeței BlockingQueue sunt: *LinkedBlockingQueue*, *PriorityBlockingQueue*, *ArrayBlockingQueue si SynchronousQueue*.