Programozási technológia 2.

Szálkezelés – 2.

Java Concurrent API,

Párhuzamosság Swing környezetben

Dr. Szendrei Rudolf

ELTE Informatikai Kar

2018.

Java Concurrent API

Java Concurrent API

- Az eddig látott szálkezelési lehetőségek a Java kezdetektől elérhető alacsony szintű konstrukciói (**`wait`**, **`notify`**, **`synchronized`**) .
- Összetettebb feladatok elvégzéséhez magasabb szintű elemekre van szükség.
- Java 1.5 óta elérhető a Concurrency Utilities, mellyel
  - javítható a program teljesítménye,
  - javítható a forráskód olvashatósága,
  - könnyebb a szálkezelési problémák megelőzése,
  - az alkalmazás könnyebben karbantarthatóvá válik.

Java Concurrent API

Java Concurrent API részei

- Task Scheduling Framework
- Callable's and Future's
- Concurrent Collections
- Atomic variables
- Locks

Java Concurrent API

Executor / ExecutorService / Executors

- Executor framework támogatja:
  - taskok szabványos módon való létrehozását,
  - ütemezését,
  - végrehajtását.

- A framework részei:
  - **`Executor`**: interfész, amely támogatja a taskok elindítását.
  - **`ExecutorService`**: az **`Executor`** interfészt bővíti ki, a taskok életciklusának menedzselési lehetőségével.
  - **`Executors`**: gyártó osztály a különböző **`ExecutorService`** implementációk létrehozására.

Java Concurrent API

Executor Interface

- Az alacsony szintű elemekkel létrehozott szálak esetén szoros kapcsolat van a **`Thread`**, és szál végrehajtandó feladata között.
- Nagyobb alkalmazásokban előnyös a szálak létrehozását elkülöníteni az alkalmazás több részétől, melyet az **`Executor`** interfész segítségével tehetünk meg.
- A taskok végrehajtása a használt **`Executor`** implementációjától függ.

- Például:

```
Executor executor = getSomeKindofExecutor();
executor.execute(new RunnableTask1());
```

Java Concurrent API

Executor / ExecutorService

- Az **`ExecutorService`** az életciklust kezelő metódusokkal egészíti ki az **`Executor`**-t.

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

public interface ExecutorService extends Executor {
    void shutdown();
    List<Runnable> shutdownNow();
    boolean isShutdown();
    boolean isTerminated();
    boolean awaitTermination(long timeout, TimeUnit unit);
    ...
}
```

Java Concurrent API

Callable

- Probléma: (alacsony szintű API-t használva)
  Új szál indítása után nincs lehetőség arra, hogy a szál értéket adjon vissza befejeződésekor közös változók, illetve megfelelő szinkronizáció nélkül.
- Ennek alacsony szintű megoldása a kódot bonyolulttá és nehezen érthetővé teszi.
- Erre ad megoldást a **`Callable`** interfész
  - **`run`** metódus helyett itt **`call()`**-t kell megvalósítani
- Az indító szál egy **`Future`** objektumon keresztül tud hozzáférni az eredményhez. (**`get()`** metódus)
  - Ha az eredmény készen van (a szál lefutott és a **`call()`** visszatért), megadja az eredményt.
  - Ha nem: a hívó szál blokkolva lesz.

Java Concurrent API

ExecutorService

- A taskok futtatásra alkalmas **`execute`** metódust kiegészíti egy hasonló, de sokoldalúbb **`submit`** metódussal, aminek
  - Paramétere: **`Runnable`** vagy **`Callable<T>`**, utóbbi lehetőséget ad arra, hogy a task eredménnyel térjen vissza.
  - Értéke: egy **`Future`** objektum, amin keresztül majd elérhető a **`Callable`** által visszaadott érték, és kezelhető annak életciklusa.

- **`ScheduledExecutorService`**:
  segítségével a **`Callable`** objektumok végrehajtása időzíthető, késleltetett indítással, vagy ismétlődő végrehajtással

Java Concurrent API

Példa

```
class CallableExample implements Callable<String>{
   public String call() {
      /* Do some work and create a result */
      String result = "The work is ended";
      return result;
   }
}
...
ExecutorService es = Executors.newSingleThreadExecutor();
Future<String>  f  = es.submit(new CallableExample());
/* Do some work in parallel */
try {
    String callableResult = f.get();
} catch (InterruptedException ie) { /* Handle */ }
```

Java Concurrent API

Thread Pool

- A legtöbb **`Executor`** implementáció egy Thread Pool-t használ, amely worker thread-ekből áll.
- Ezek a szálak több task végrehajtására alkalmasak.
- Mivel a **`Thread`** objektum létrehozása és felszabadítása jelentős költséggel jár, ezért ennek a konstrukciónak a segítségével spórolhatunk az erőforrásokon (kevesebbszer kell szálakat létrehozni azok újrahasznosítása miatt).

Java Concurrent API

Példa: FixedThreadpool

- Rögzített számú taskot futtat egyszerre.
- Előnye: a teljesítmény csökkenés folyamatos
  - Tegyük fel, hogy egy web-szerver minden HTTP kérés esetén egy új szálat indít a kérés kiszolgálására.
  - Az alkalmazás leáll, ha több kérést kap, mint amennyit azonnal ki tud szolgálni (vagyis a szükséges szálak száma meghaladja a rendszer kapacitását).
  - A szálak számának limitálásával azonban az alkalmazás lassabban fog ugyan válaszolni a kérésekre ilyenkor, de továbbra is működőképes marad.

Java Concurrent API

Thread Pool létrehozása

- Az **`Executors`** gyártó osztály tartalmazza a szükséges metódusokat
  - **`newFixedThreadPool`**: Thread tároló fix mérettel

```
ExecutorService executor =
Executors.newFixedThreadPool(10);
```

  - **`newCachedThreadPool`**: kiterjeszthető, pl.: sok rövid életű task esetén.
  - **`newSingleThreadExecutor`**: Olyan Executor, amely egyetlen taskot hajt végre egyszerre.

Java Concurrent API

Konkurrens gyűjtemények

- Wrapper implementációk
- Minden gyűjteményhez tartozik egy szálbiztos (szinkronizált) implementáció.
- A Collections osztály segítségével hozhatóak létre.
  Pl.: synchronizedList
- A bejárást mindig szinkronizált blokkon belül kell elvégezni.

```
Collection<Type> c =
Collections.synchronizedCollection(myCollection);

synchronized(c) {

   for (Type e : c){ foo(e); }

}
```

- BlockingQueue

Java Concurrent API

Atomi változók

- java.util.concurrent.atomic csomag
  - Egyszerű osztályok, amelyek a változók zárolás mentes, szálbiztos használatát teszik lehetővé.

```
AtomicInteger balance = new AtomicInteger(0);
public int deposit(integer amount) {
    return balance.addAndGet(amount);
}
```

Java Concurrent API

Counter példa

```
class AtomicCounter {
   private AtomicInteger c = new AtomicInteger(0);

   public void increment() { c.incrementAndGet(); }
   public void decrement() { c.decrementAndGet(); }
   public int value() {      return c.get(); }
}
```

Java Concurrent API

Lock

- A szinkronizáció a reentrant lock-ok egy egyszerű fajtája, amely könnyen használható, de erősen korlátozott lehetőségekkel bír.
- A **`Lock`** interfész sokkal kifinomultabb zárolási lehetőségeket biztosít.
- A **`Lock`** objektumok hasonlóan működnek, mint a szinkronizációnál használt monitor lock-ok (objektum szintű zár)
- Tulajdonságai
  - Egyszerre csak egy szál birtokolhatja a zárat
  - Támogatja a **`wait/notify`** használatát egy hozzá tartozó **`Condition`** objektumon keresztül.
  - A zárat nem lehet automatikusan elengedni (**`try/finally`**)

Java Concurrent API

Lock

- Zárolás két módja:
  - tryLock: visszatér, ha a lock nem elérhető (megadható timeout)
  - lockInterruptibly: visszatér, ha egy másik szál megszakítást küld, mielőtt a lock-ot megkaptuk volna
- A Lock objektumokkal megoldható a holtpont probléma

Java Concurrent API

Lock implementációk

- ReentrantLock
  - A zárat birtokló szál többször is hívhatja a lock() metódust, anélkül, hogy blokkolna
- ReadWriteLock
  - Két zárat használ, egyet-egyet az írás, illetve az olvasás biztosítására
  - Több szál birtokolhatja a olvasási zárat, ha egyetlen szál sem birtokolja az írási zárat
  - Ha egy szál birtokolja az írási zárat, egyetlen más szál sem kaphatja meg az olvasási zárat
    - rwl.readLock().lock();
    - rwl.writeLock().lock();

```
class ReadWriteMap {
    final Map<String, Data> m = new TreeMap<String, Data>();
    final ReentrantReadWriteLock rwl = new ReentrantReadWriteLock();
    final Lock r = rwl.readLock();
    final Lock w = rwl.writeLock();
    public Data get(String key) {
        r.lock();
        try { return m.get(key) }
        finally { r.unlock(); }
    }
    public Data put(String key, Data value) {
        W.lock();
        try { return m.put(key, value); }
        finally { w.unlock(); }
    }
    public void clear() {
        w.lock();
        try { m.clear(); }
        finally { w.unlock(); }
    }
}
```

Java Concurrent API

Fork / Join

- A framework az **`ExecutorService`** egy implementációja. Segíti a rendelkezésre álló erőforrások (több processzor) kihasználását.
- Rekurzívan kisebb részekre bontható feladatok megoldását segíti.
- A többi **`ExecutorService`** implementációhoz hasonlóan több worker-thread között osztja szét a taskokat.
- Különbség: Work-stealing algoritmus
  - Egy worker-thread, amely végzett a feladatával, átvehet taskokat a még elfoglalt szálaktól.

Java Concurrent API

Fork / Join

- Alap használata:

```
if (my portion of the work is small enough)
    do the work directly
else
    split my work into two pieces
    invoke the two pieces and wait for the results
```

- Első lépésként olyan kódot kell írni, amely a feladat felosztását elvégzi.
- Példa: Kép elhomályosítása
  - A forrás kép integer tömbként adott, amelynek minden eleme egy pixel szín értékét adja meg
  - A cél képet egy hasonló tömb reprezentálja
  - Az elhomályosítás úgy történik, hogy minden pixelt átlagolni kell a körülötte lévőkkel (piros, zöld, kék komponensek átlagolása)

Párhuzamosság Swing környezetben

Párhuzamosság Swing környezetben

- A szálkezelés a grafikus alkalmazásokban is fontos.
- Cél egy olyan felhasználói felület készítése, amely soha nem fagy, mindig válaszol a felhasználói interakciókra, bármit is csináljon éppen.
- A Swing 3 féle szállal dolgozik:
  - Kezdeti szálak: (initail Threads) — az alkalmazást futtató kezdeti szállal
  - Eseménykezelő szál: (Event dispatch thread) (röviden EDT) — az eseménykezelő kódját és swing-el kapcsolatos interakciókat futtatja
  - Háttér szálak: (worker-threads) — időigényes műveletek háttérben futtatására
- A szálakat nem szükséges explicit létrehozni, ezeket a Swing kezeli helyettünk.

Párhuzamosság Swing környezetben

Kezdeti szálak

- Minden alkalmazáshoz tartozik néhány szál, ahonnan az alkalmazás elindul (általában ez a main thread).
- Swing alkalmazásokban a kezdeti szál nem lát el sok feladatot
- Legfontosabb feladatai
  - A GUI-t inicializáló **`Runnable`** objektum létrehozása
  - A **`Runnable`** objektum ütemezése az EDT-re
- A GUI elindítása után az alkalmazást többnyire az interfész eseményei vezérlik.
- Az események rövid taskok végrehajtását váltják ki az EDT-n.

Párhuzamosság Swing környezetben

Kezdeti szálak, Eseménykezelő szál (EDT)

- Az alkalmazás egyéb taskokat is tud az EDT-re vagy háttér szálra ütemezni.
- A kezdeti szálak a GUI létrehozásának ütemezése
  - **`SwingUtilites.invokeLater`**: Ütemezi a taskot és visszatér
  - **`SwingUtilities.invokeAndWait`**: Ütemezi a taskot és megvárja, hogy befejeződjön
- Általában a GUI létrehozásának beütemezése az utolsó dolog, amit a main szál végez.
- **Minden Swing componenst használó kódnak az EDT-en kell futnia!**

```
SwingUtilities.invokeLater(new Runnable() {
    public void run() {
        createAndShowGUI();
    }
});
```

Párhuzamosság Swing környezetben

Eseménykezelő szál (EDT)

- A Swing eseménykezelő kódja egy speciális szálon fut (EDT).
- Swing komponenseket csak ezen a szálon futó kódból hozhatunk létre vagy használhatunk!
- A legtöbb Swing objektum nem szálbiztos
- Az EDT rövid taskok sorozataként fut:
  - Leggyakoribb az eseménykezelő metódusok hívása, pl. **`actionPerformed`**
  - Egyéb: az alkalmazás által ütemezett taskok, az **`invokeLater`** és **`invokeAndWait`** metódusok használatával
  - Az taskoknak rövidnek kell lenniük
- Kódban a **`SwingUtilities.isEventDispatchThread`** metódussal kérdezhetjük meg, hogy az az EDT-n fut-e.

Párhuzamosság Swing környezetben

Worker Threads – SwingWorker

- Hosszú futásidejű feladatok végrehajtására használható szálak
- Minden task-ot egy **`SwingWorker`** példány reprezentál
- Maga a **`SwingWorker`** egy absztrakt osztály, ezért ennek egy leszármazottját kell definiálniunk
- **`SwingWorker`** a Java SE 6 óta használható

Párhuzamosság Swing környezetben

Worker Threads – SwingWorker

- Kommunikációs módjai
  - A háttér szál elérhetővé tud tenni részeredményeket a futás befejeződése előtt a **`publish`** metódus hívásával
  - A részeredményeket a **`process`** metódus dolgozza fel, amely automatikusan az EDT-n fut
  - A **`done`** metódusa a háttér task lefutása után automatikusan meghívásra kerül az EDT-n
  - A **`SwingWorker`** implementálja a Future interfészt, így a háttér task adhat vissza értéket.
  - Az interfész lehetővé teszi még a task leállítását (cancel) és a task állapotának lekérdezését (befejezedőött, leállított).

Párhuzamosság Swing környezetben

```
SwingWorker worker = new SwingWorker<ImageIcon[], Void>() {
    public ImageIcon[] doInBackground() {
        final ImageIcon[] innerImgs = new ImageIcon[nimgs];
        for (int i = 0; i < nimgs; i++){
            innerImgs[i] = loadImage(i+1);
        }
        return innerImgs;
    }
    public void done() {
        try { imgs = get(); }       // vagy get(1000);
        catch (InterruptedException ignore) {}
        catch (java.util.concurrent.ExecutionException e) {
            ...
        }
    }
};
```

Párhuzamosság Swing környezetben

```
private class FlipTask extends SwingWorker<Void, Integer> {
    protected Void doInBackground() {
        Random random = new Random();
        while (!isCancelled()) {
             publish(random.nextInt(1000));
        }
        return null;
    }
    protected void process(List<Integer> r) {
         ...
    }
}
```

Párhuzamosság Swing környezetben

Worker Threads – Futás megszakítása

- A háttér task-nak támogatnia kell a megszakítását:
  - Befejezés interrupt hatására (ahogy a szálaknál), **`isInterrupted`** által visszaadot érték figyelésével
  - **`isCancelled()`** metódus hívása periódikusan. Igazat ad vissza, ha a worker **`cancel`** metódusát meghívták

Párhuzamosság Swing környezetben

Swing Timer

- Egy vagy több Action Event-et generál a megadott idő után
- Az általános Timer-el ellentétben, ez a GUI-val kapcsolatos időzítéses feladatok elvégzésére való
- Előre létrehozott időzítő szálat használ
- A GUI taskok automatikusan az EDT-n hajtódnak végre
- Felhasználási módok:
  - Task végrehajtása egyszer, késleltetés után
  - Ismétlődő feladatok végrehajtása.

Párhuzamosság Swing környezetben

Swing Timer használata

- Az időzítő létrehozásakor meg kell adni:
  - egy ActionListener-t, amely lefut a megadott időben
  - a végrehajtások közötti várakozási időt
- A `setRepeats(false)` hívással megadható, hogy az időzítő csak egyszer járjon le, ne periodikusan.
- Az időzítőt a `Start` metódussal lehet elindítani, és a `Stop`-al megállítani.