Vláknové programování
část IV

Lukáš Hejmánek, Petr Holub
{xhejtman,hopet}@ics.muni.cz

Laboratoř pokročilých síťových technologií

PV192
2013–03–12

Přehled přednášky

Úlohy a vlákna

Executors, Thread Pools a Futures

Ukončování a přerušování

Úlohy a vlákna

- Úloha vs. vlákno
  - úloha – co se vykonává (`Runnable, Callable`)
  - vlákno – kdo úlohu vykonává (Executor/Future/TPE/…)
- Oddělení úloh od vláken
  - úloha nesmí předpokládat nic o chování vlákna, které ji vykonává
  - Politika ukončení vs. politika přerušení

(příklady povětšinou převzaty z JCiP, Goetz)

Executors, Thread Pools

- Koncept vykonavatelů kódu: Executors
  - vykonávají se objekty implementující Runnable
  - různé typy Executors
- ExecutorService přidává
  - schopnost zastavit vykonávání
  - schopnost vykonávat Callable<V>, nikoli pouze Runnable()
  - vracet objekty representované jako Future
- ThreadPoolExecutor
  - všeobecně použitelný executor, jednoduché API
  - minimální i maximální počet vláken
  - recyklace vláken
  - likvidace nepoužívaných vláken

Runnable vs. Callable

- Interface Runnable
  - implementuje úlohu
  - lze použít s konstruktorem třídy Thread
    - konceptuálně čistější přístup: nerozšiřujeme třídu, kterou vlastně rozšiřovat nechceme
  - použití i v hlavním vlákně

```
public class PrikladRunnable {
    static class RunnableVlakno implements Runnable {
        public void run() {
            System.out.println("Tu je vlakno.");
        }
    }

    public static void main(String[] args) {
        System.out.print("Startuji vlakno: ");
        new Thread(new RunnableVlakno()).start();
        System.out.println("hotovo.");
        System.out.println("Spoustim primo v hlavnim vlakne: ");
        new RunnableVlakno().run();
    }
}
```

Runnable vs. Callable

- Interface Callable<V>
  - na rozdíl od Runnable může vracet výsledek (typu V) a vyhodit výjimku

```
1  import java.util.concurrent.Callable;

3  public class PrikladCallable {
       static class CallableVlakno implements Callable<String> {
5          public String call() throws Exception {
               return "Retezec z Callable";
7          }
       }
9
       public static void main(String[] args) {
11         try {
               String s = new CallableVlakno().call();
13             System.out.println(s);
           } catch (Exception e) {
15             System.out.println("Chytil jsem vyjimku");
           }
17     }
   }
```

Executors

- Typy Executorů
  - SingleThreadExecutor
    - sekvenční vykonávání úloh
    - pokud vlákno selže, pokračuje se vykonáváním následujícího
  - ScheduledThreadPool
    - zpožděné či opakované vykonávání vláken
  - FixedThreadPool
    - používá pevný počet vláken
  - CachedThreadPool
    - vytváří nová vlákna dle potřeby
    - opakovaně používá existující uvolněná vlákna
  - ScheduledExecutorService
    - implmentace spouštění s definovaným zpožděním a opakovaného spouštění
    - **`http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/ScheduledExecutorService.html`**
  - Executors factory
    - implementace vlastních typů Executorů
    - **`http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/Executors.html`**

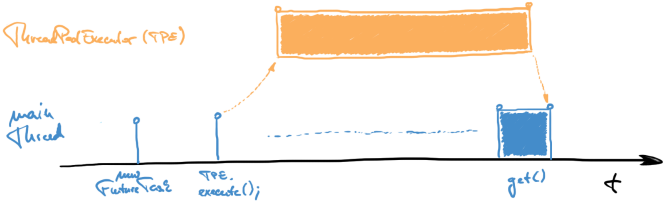
Executors

```
import java.util.concurrent.*;
import java.util.Random;

public class TPE {
    public static void main(String[] args) {
        final Random random = new Random();
        // forkbomba: ;-)
        // ExecutorService executor = Executors.newCachedThreadPool();
        ExecutorService executor = Executors.newFixedThreadPool(
                Runtime.getRuntime().availableProcessors()-1);
        for (int i = 0; i < 100; i++) {
            executor.execute(new Runnable() {
                public void run() {
                    int max = random.nextInt();
                    for(int j = 0; j < max; j++) { j += 2; j--; }
                    System.out.println("Dobehlo vlakno s max = " + max);
                }
            });
        }
        try {
            Thread.sleep(10000);
            executor.shutdown();
            executor.awaitTermination(1000, TimeUnit.SECONDS);
        } catch (InterruptedException e) {
        }
    }
}
```

Futures

- Princip:
  - někdy v budoucnu bude volající potřebovat výsledek výpočtu *X*
  - v době, kdy si volající řekne o výsledek výpočtu *X*: *(a)* výsledek je okamžitě vrácen, pokud je již k dispozici, nebo *(b)* volající se zablokuje, výsledek se dopočítá a vrátí, volající se odblokuje
- H. Baker, C. Hewitt, "The Incremental Garbage Collection of Processes". *Proceedings of the Symposium on Artificial Intelligence Programming Languages, SIGPLAN Notices 12.* August 1977.
  podobný koncept
  D. Friedman. "CONS should not evaluate its arguments". S. Michaelson and R. Milner, editors, *Automata, Languages and Programming*, pages 257-284. Edinburgh University Press, Edinburgh. Also available as *Indiana University Department of Computer Science Technical Report TR44.* 1976



Futures a ThreadPoolExecutor

```
import java.util.concurrent.*;

public class Futures {
    public static class StringCallable implements Callable {
        public String call() throws Exception {
            System.out.println("FT: Pocitam.");
            Thread.sleep(5000);
            System.out.println("FT: Vypocet hotov.");
            return "12345";
        }
    }
    public static void main(String[] args) {
        ThreadPoolExecutor tpe = new ThreadPoolExecutor(2, 8, 60L,
                TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>());
        FutureTask ft = new FutureTask(new StringCallable());
        System.out.println("main: Poustim vypocet.");
        tpe.execute(ft);
        // alternativa: Future ft = tpe.submit(new StringCallable());
        try {
            System.out.println("main: Chci vysledek.");
            String s = (String) ft.get();
            System.out.println("main: Mam vysledek: " + s);
            tpe.shutdown();
            tpe.awaitTermination(1, TimeUnit.MINUTES);
        } catch (InterruptedException e) {}
        catch (ExecutionException e) {}
    }
}
```

Futures vs. CompletionService

- Problém: máme řadu odložených úloh (Future) a potřebujeme je v pořadí dokončení, nikoli zaslání
  1. opakované procházení seznamu a používání `get(0, TimeUnit.SECONDS);`
  2. použijeme `CompletionService`
- `CompletionService`
  - kombinuje `Executor` a `BlockingQueue`
  - `submit()` – vkládáme úlohy pomocí
  - `take()` a `poll()` – vybíráme dokončené úlohy
  - při prázdné frontě dokončných úloh se `take()` blokuje, `poll()` vrací `null`

Futures vs. CompletionService

```
ArrayList<FileData> stahniSoubory(ArrayList<String> list) {
    ArrayList<FileData> ald = new ArrayList<FileData>();
    CompletionService<FileData> completionService =
            new ExecutorCompletionService<FileData>(
                    new ThreadPoolExecutor(1, 10, 60, TimeUnit.SECONDS,
                            new LinkedBlockingQueue<Runnable>()));
    for (final String s : list) {
        completionService.submit(new Callable<FileData>() {
            public FileData call() throws Exception {
                FileData fd = new FileData();
                fd.s = s; fd.data = getFile(s);
                return fd;
            }
        });
    }
    try {
        for (int i = 0, size = list.size(); i < size; i++) {
            Future<FileData> f = completionService.take();
            ald.add(f.get());
        }
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    } catch (ExecutionException e) { launderThrowable(e.getCause()); }
    return ald;
```

Futures vs. CompletionService

```
    public static RuntimeException launderThrowable(Throwable t) {
2       if (t instanceof RuntimeException)
            return (RuntimeException) t;
4       else if (t instanceof Error)
            throw (Error) t;
6       else
            throw new IllegalStateException("Not unchecked", t);
8   }
```

Ukončování a přerušování pro pokročilé

- Kooperativní ukončování úloh a přerušování vláken
  - příznakem proměnné
  - přerušením – interrupt
  - `Thread.stop` – deprecated
- Důvody ukončení úloh
  - uživatelem vyvolané ukončení úlohy (GUI, JMX)
  - časově omezené úlohy
  - události uvnitř – několik úloh hledá řešení paralelně, jedna ho najde
  - externí chyby
  - ukončení aplikace

Ukončování a přerušování pro pokročilé

- Politika ukončování (cancellation policy)
  - vývojářem specifikováno pro každou úlohu (JavaDoc)
  - jak? – jak se vyvolává ukončení?
  - kdy? – kdy je možné vlákno ukončit?
  - co? – co bude třeba udělat před ukončením?
- Ukončování příznakem a/nebo přerušením?

Přerušení – interrupt

- Mechanismus zasílání zprávy mezi vlákny
  - sémanticky definováno jen jako signalizace mezi vlákny
  - nastavení příznaku

```
1  public class Thread {
       public void interrupt() {...}
3      public boolean isInterrupted() {...}
       public static boolean interrupted() {...}
5  }
```

- Pozor na metodu interrupted()
  - vrátí a *vymaže* stav příznaku
- Zpracování přerušení
  - vyhození výjimky `InterruptedException`
  - předání příznaku dále
  - polknutí příznaku 
- Typické metody na `InterruptedException`
  - `wait, sleep, join`
  - blokující operace na omezených frontách (`BlockingQueue x.put`)

Přerušení – interrupt

- Politiky přerušení
  - specifikováno vývojářem pro každé vlákno
  - standardní chování: ukliď, dej vědět vlastníkovi (TPE) a zmiz
  - nestandardní chování: není vhodné pro normální úlohy
  - vlákno může potřebovat předat stav `interrupted` svému TPE
  - úloha by neměla předpokládat nic o politice vlákna, v němž běží
    - předat stav dál
    - buď `throw new InterruptedException();`
    - nebo `Thread.currentThread().interrupt();`
      např. pokud je úloha `Runnable`
  - vlákno/TPE může následně `interrupted` příznak potřebovat
  - specifikace: kdy?, jak?, další předání?

Přerušení – interrupt

- Kombinace blokujících operací s politikou přerušení a úlohy s ukončením až na konci

```
public Task getNextTask(BlockingQueue<Task> queue) {
    boolean interrupted = false;
    try {
        while (true) {
            try {
                return queue.take();
            } catch (InterruptedException e) {
                interrupted = true;
            }
        }
    } finally {
        if (interrupted) Thread.currentThread().interrupt();
    }
}
```

  - nesmíme příznak `interrupted` nastavit před voláním `take()`, protože by volání hned skončilo

Omezený běh – Futures

- `Future` má metodu `cancel(boolean mayInterruptIfRunnig)`
  - `mayInterruptIfRunnig = true` znamená, že se má běžící úloha přerušit
  - `mayInterruptIfRunnig = false` znamená, že se pouze nemá spustit, pokud ještě neběží
  - vrací, zda se ukončení povedlo
- Kdy můžeme použít `mayInterruptIfRunnig = true`?
  - pokud známe politiku přerušení vlákna
  - pro standardní implementace `Executor` to je známé a bezpečné

Omezený běh – Futures

```
public class FutureCancel {
    ThreadPoolExecutor taskExec = new ThreadPoolExecutor(1,10,60,
            TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>());
    public void timedRun (Runnable r, long timeout, TimeUnit unit)
            throws InterruptedException {
        Future<?> task = taskExec.submit(r);
        try {
            task.get(120, TimeUnit.SECONDS);
        } catch (ExecutionException e) {
            throw new RuntimeException(e.getMessage());
        } catch (TimeoutException e) {
            // uloha bude ukoncena nize
        }
        finally {
            // neskodne, pokud ukloha skoncila,
            // jinak interrupt
            task.cancel(true);
        }
    }
```

Nepřerušitelná blokování

- Existují blokování, která nereagují na `interrupt`
- Příklady:
  - synchronní soketové I/O v java.io
    - *problém:* metody `read` a `write` na `InputStream` a `OutputStream` nereagují na `interrupt`
    - *řešení:* zavřít socket, visící čtení/zápis vyhodí `SocketException`
  - čekání na získání monitoru (intrinsic lock)
    - *problém:* vlákno čekající na monitor (`synchornized`) nereaguje na `interrupt`
    - *řešení:* neexistuje „násilné“ řešení pro monitory, musí se dočkat
    - *obejití:* explicitní zámky `Lock` podporují metodu `lockInterruptibly`

Nepřerušitelná blokování

- Další vychytávky:
  - synchronní I/O v java.nio
    - přerušení vyhází u všech zablokovaných vláken **`ClosedByInterruptException`**, pokud je kanál typu **`InterruptibleChannel`**
    - zavření vyhází u všech zablokovaných vláken **`AsynchronousCloseException`**, pokud je kanál typu **`InterruptibleChannel`**
  - asynchronní I/O při použití Selector
    - **`Selector.select`** vyhodí výjimku **`ClosedSelectorException`**, pokud obdrží **`interrupt`**

Nepřerušitelná blokování

- Využití `ThreadPoolExecutor.newTaskFor(callable)`
  - dostupné od Java 6
  - vrací `RunnableFuture` pro danou úlohu
  - přepsání `newTaskFor` umožňuje vlastní tvorbu `RunnableFuture` a tudíž přepsat metodu `cancel()`
    - uzavření synchronních socketů pro java.io
    - statistiky, debugování, atd.
  - lze napsat tak, že si Callable/Runnable dodá vlastní implementaci `cancel()`
    `http://www.javaconcurrencyinpractice.com/listings/SocketUsingTask.java`

Zastavování vláknových služeb

- Problém dlouho běžících vláken
  - vlákna v exekutorech často běží déle, než tvůrce executorů
- Vlákno by měl zastavovat jeho „vlastník“
  - vlastník vláken není definován formálně
  - bere se ten, kdo ho vytvořil
  - vlastnictví není transitivní (jako u objektů – princip zapouzdření)
  - vlastník by měl poskytovat metody na řízení životního cyklu
  - požadavek na ukončení by měl být signalizován vlastníkovi

Zastavování vláknových služeb

```
public class LogWriter {
    private final BlockingQueue<String> queue;
    private final LoggerThread logger;
    private volatile boolean shutdownRequested = false;

    public LogWriter() throws FileNotFoundException {
        this.queue = new LinkedBlockingQueue<String>();
        this.logger = new LoggerThread(new PrintWriter("mujSoubor"));
        logger.start();
    }

    private class LoggerThread extends Thread {
        private final PrintWriter writer;

        private LoggerThread(PrintWriter writer) {
            super("Logger Thread");
            this.writer = writer;
        }

        public void run() {
            try {
                while (true)
                    writer.println(queue.take());
            } catch (InterruptedException ignored) {
            } finally {
                writer.close();
            }
        }
    }
```

Zastavování vláknových služeb

```
1   public void stop() {
        shutdownRequested = true;
3       logger.interrupt();
    }
5
    public void log (String msg) throws InterruptedException {
7           queue.put (msg);
    }
```

- Potřeba ukončovat konzumenty i producenty
  - konzument: `run()`
  - producent: `log(String msg)`

Zastavování vláknových služeb

```
    public void logLepe (String msg) throws InterruptedException {
2       if (!shutdownRequested)
            queue.put (msg);
4       else
            throw new IllegalStateException("logger se ukoncuje");
6   }
```

- Ukončení producenta
  - jakpak zjistíme jeho vlákno?
  - nijak ;-)
  - už je to správně?

Zastavování vláknových služeb

```
1   public void logLepe (String msg) throws InterruptedException {
        if (!shutdownRequested)
3           queue.put (msg);
        else
5           throw new IllegalStateException("logger se ukoncuje");
    }
```

- … není!
- Race condition
  - složené testování podmínky a volání metody!
- Složené zamykání
  - testování a rezervace v jednom `synchronized` bloku
  - konzument testuje, že zpracoval všechny rezervace

Zastavování vláknových služeb

```
public class SafeLogWriter {
    private final BlockingQueue<String> queue;
    private final LoggerThread logger;
    @GuardedBy("this") private volatile boolean shutdownRequested
            = false;
    @GuardedBy("this") private int reservations;
```

...

```
        public void run() {
            try {
                while (true) {
                    synchronized (this) {
                        if (shutdownRequested && reservations == 0)
                            break;
                    }
                    String msg = queue.take();
                    synchronized (this) {--reservations;};
                    writer.println(msg);
                }
            } catch (InterruptedException ignored) {
            } finally {
                writer.close();
            }
        }
```

Zastavování vláknových služeb

```
    public void log (String msg) throws InterruptedException {
2       synchronized (this) {
            if (shutdownRequested)
4               throw new IllegalStateException("logger se ukoncuje");
            ++reservations;
6       }
        queue.put (msg);
8   }
```

Zastavování vláknových služeb

- **`ExecutorService`**
  - proč nepoužít, co je hotovo?
  - **`shutdown()`**
    - pohodové ukončení
    - dokončí se zařazené úlohy
  - **`shutdownNow()`**
    - vrací seznam úloh, které ještě nenastartovaly
    - problém, jak se dostat k seznamu úloh, které nastartovaly, ale byly ukončeny
  - nemá metodu, která by umožnila dokončit bežící úlohy a nové už nestartovala
  - zapouzdření do vlastního ukončování:

    ```
    exec.shutdown();
    exec.awaitTermination(timeout, unit);
    ```

  - využití i pro jednoduchá vlákna: **`newSingleThreadExecutor()`**

Zastavování vláknových služeb

```
public class TrackingExecutor extends AbstractExecutorService {
    private final ExecutorService exec;
    private final Set<Runnable> tasksCancelledAtShutdown =
            Collections.synchronizedSet(new HashSet<Runnable>());
```

...

```
    public List<Runnable> getCancelledTasks() {
        if (!exec.isTerminated())
            throw new IllegalStateException(/*...*/);
        return new ArrayList<Runnable>(tasksCancelledAtShutdown);
    }

    public void execute(final Runnable runnable) {
        exec.execute(new Runnable() {
            public void run() {
                try {
                    runnable.run();
                } finally {
                    if (isShutdown()
                            && Thread.currentThread().isInterrupted())
                        tasksCancelledAtShutdown.add(runnable);
                }
            }
        });
    }
```

Zastavování vláknových služeb

- Vzor – jedovaté sousto (poison pill)
  - ukončování systému producent – konzument
  - jedovaté sousto – jeden konkrétní typ zprávy
  - funguje pro známý počet producentů
    - konzument umře po požití N_{prod} otrávených soust
  - lze rozšířit i na více konzumentů
    - každý producent musí do fronty zapsat N_{konz} otrávených soust
    - problém s počtem zpráv $N_{prod} \cdot N_{konz}$

Ošetření abnormálního ukončení vlákna

- Zachytávání `RunTimeException`
  - normálně se nedělá, měla by vyústit v stacktrace
  - potřeba zpracovat, pokud vlákno vykonává úplně cizí kód
  - strategie:
    - zachytit, uložit, pokračovat
      `try {...} catch (...) {...}`
      v případě, že se vlákno o sebe musí postarat samo
    - ukončit a dát vědět vlastníkovi
      `try {...} finally {...}`
      možnost předat `Throwable`

```
Throwable thrown = null;
try {runTask(getTaskFromQueue());}
catch (Throwable e) {thrown = e;}
finally { threadExited (this, thrown);}
```

Ošetření abnormálního ukončení vlákna

- **`UncaughtExceptionHandler`**
  - aplikace si může nastavit vlastní zpracování nezachycených výjimek
  - pokud není nastaven, vypisuje se stacktrace na **`System.err`**
  1. **`Thread.setUncaughtExceptionHandler`**
     - Java $\geq$ 5.0
     - per vlákno
  2. **`ThreadGroup`**
     - Java $<$ 5.0
  - zavolá se pouze první
  - pro TPE se nastavuje pomocí vlastní **`ThreadFactory`** přes konstruktor TPE
    - standardní TPE nechá po nezachycené výjimce ukončit dané vlákno
    - bez **`UncaughtExceptionHandler`** mohou vlákna tiše mizet
    - možnost task obalit do dalšího Runnable/Callable
    - vlastní TPE s alternativním **`afterExecute`**
- Propagace nezachycených výjimek
  - do **`UncaughtExceptionHandler`** se dostanou pouze úlohy zaslané přes **`execute()`**
  - **`submit()`** vrací výjimku jakou součást návratové hodnoty/stavu – **`Future.get()`**

Ukončování JVM

- Normální ukončení (orderly termination)
  - ukončení posledního nedémonického vlákna
  - volání `System.exit();`
  - platformově závislé ukončení (SIGINT, Ctrl-C)
- Abnormální ukončení (abrupt termination)
  - volání `Runtime.halt();`
  - platformově závislé ukončení (SIGKILL)
- Háčky při ukončení (shutdown hooks)
  - `Runtime.addShutdownHook`
  - předává se implementace vlákna
  - JVM negarantuje pořadí
  - pokud v době ukončování běží jiná vlákna, poběží paralelně s háčky
  - háčky musí být thread-safe: synchronizace
  - např. signalizace ukončení jiným vláknům, mazání dočasných souborů, …
  - pokud nějaké vlákno počítá se signalizací ukončení při ukončování JVM, může si samo zaregistrovat háček (ale ne z konstruktoru!)
  - použití jednoho velkého háčku: odpadá problém se synchronizací, možnost zajištění definovaného pořadí ukončování komponent

Ukončování JVM

- Démonická vlákna
  - metoda `setDaemon()`
  - démonický stav se dědí
  - ukončování JVM: pokud běží jen démonická vlákna, JVM se normálně ukončí
    - neprovedou se bloky `finally`
    - neprovede se vyčištění zásobníku
  - příklad: garbage collection, čištění dočasné paměťové cache
  - nepoužívat z lenosti!
- Finalizers
  - týká se objektů s netriviální metodou `finalize()`
    - obtížné napsat správně
    - musí být synchronizovány
    - není garantováno pořadí
    - výkonnostní penalta
    - obvykle jde nahradit pomocí bloku `finally` a explicitního uvolnění zdrojů
  - po doběhnutí háčku se spustí finalizers pokud `runFinalizersOnExit == true`
  - vyhýbat se jim!