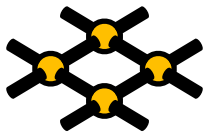
Vláknové programování
část VII

Lukáš Hejmánek, Petr Holub

{xhejtman,hopet}@ics.muni.cz



Laboratoř pokročilých síťových technologií

PV192

2011–04–07

Přehled přednášky

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`**

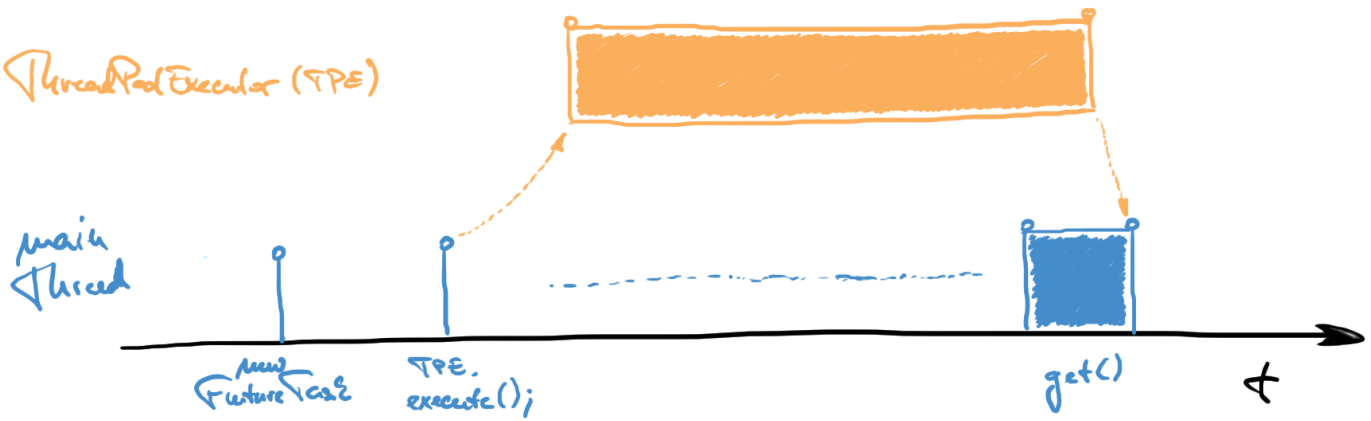
Executors

```
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

```
1  import java.util.concurrent.*;
2
3  public class Futures {
4      public static class StringCallable implements Callable {
5          public String call() throws Exception {
6              System.out.println("FT: Pocitam.");
7              Thread.sleep(5000);
8              System.out.println("FT: Vypocet hotov.");
9              return "12345";
10         }
11     }
12     public static void main(String[] args) {
13         ThreadPoolExecutor tpe = new ThreadPoolExecutor(2, 8, 60L,
14                 TimeUnit.SECONDS, new LinkedBlockingQueue<Runnable>());
15         FutureTask ft = new FutureTask(new StringCallable());
16         System.out.println("main: Poustim vypocet.");
17         tpe.execute(ft);
18         // alternativa: Future ft = tpe.submit(new StringCallable());
19         try {
20             System.out.println("main: Chci vysledek.");
21             String s = (String) ft.get();
22             System.out.println("main: Mam vysledek: " + s);
23             tpe.shutdown();
24             tpe.awaitTermination(1, TimeUnit.MINUTES);
25         } catch (InterruptedException e) {}
26         catch (ExecutionException e) {}
27     }
28 }
```

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) {
2       ArrayList<FileData> ald = new ArrayList<FileData>();
        CompletionService<FileData> completionService =
4               new ExecutorCompletionService<FileData>(
                        new ThreadPoolExecutor(1, 10, 60, TimeUnit.SECONDS,
6                               new LinkedBlockingQueue<Runnable>()));
        for (final String s : list) {
8           completionService.submit(new Callable<FileData>() {
                public FileData call() throws Exception {
10                  FileData fd = new FileData();
                    fd.s = s; fd.data = getFile(s);
12                  return fd;
                }
14          });
        }
16      try {
            for (int i = 0, size = list.size(); i < size; i++) {
18              Future<FileData> f = completionService.take();
                ald.add(f.get());
20          }
        } catch (InterruptedException e) {
22          Thread.currentThread().interrupt();
        } catch (ExecutionException e) { launderThrowable(e.getCause()); }
24      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
  - 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!

Typy úloh pro TPE

- Nezávislé úlohy – ideální
- Problémy
  - závislost/komunikace úloh zaslaných do jednoho TPE
    - ohraničená velikost TPE
  - jednovláknový executor → TPE
  - úlohy citlivé na latenci odpovědi
    - ohraničená velikost TPE
    - dlouho běžící úlohy
  - problém s úlohami využívajícími `ThreadLocal`
    - recyklace vláken
  - nestejně velké úlohy v jednom TPE

Typy úloh pro TPE

Je tohle správně?

```
    static ExecutorService exec = Executors.newSingleThreadExecutor();
2
    public static class RenderPageTask implements Callable<String> {
4       public String call() throws Exception {
            Future<String> header, footer;
6           header = exec.submit(new LoadFileTask("header.html"));
            footer = exec.submit(new LoadFileTask("footer.html"));
8           String page = renderBody();
            return header.get() + page + footer.get();
10      }

12      private String renderBody() {
            return " body ";
14      }
    }
```

Typy úloh pro TPE

ANO

```
1           ExecutorService mainExec = Executors.newSingleThreadExecutor();
            Future<String> task = mainExec.submit(new RenderPageTask());
3           try {
                System.out.println("Vysledek: " + task.get());
5           } catch (InterruptedException e) {
                e.printStackTrace();
7           } catch (ExecutionException e) {
                e.printStackTrace();
9           }
            exec.shutdown();
11          mainExec.shutdown();
```

Typy úloh pro TPE

NE

```
1           Future<String> task = exec.submit(new RenderPageTask());
            try {
3               System.out.println("Vysledek: " + task.get());
            } catch (InterruptedException e) {
5               e.printStackTrace();
            } catch (ExecutionException e) {
7               e.printStackTrace();
            }
9           exec.shutdown();
```

Typy úloh pro TPE

- Nezávislé úlohy – ideální
- Problémy
  - závislost/komunikace úloh zaslaných do jednoho TPE
    - ohraničená velikost TPE
  - jednovláknový executor → TPE
  - úlohy citlivé na latenci odpovědi
    - ohraničená velikost TPE
    - dlouho běžící úlohy
  - problém s úlohami využívajícími `ThreadLocal`
    - recyklace vláken
  - nestejně velké úlohy v jednom TPE

Velikost TPE

- Doporučení Javy: $N_{CPU} + 1$ pro výpočení úlohy
- Obecněji

$$N_{vlaken} = N_{CPU} \cdot U_{CPU} \cdot \left(1 + \frac{W}{C}\right)$$

  kde U_{CPU} je cílové využití CPU, W je čas čekání, C je výpočetní čas

- `Runtime.getRuntime().availableProcessors();`

Vytváření a ukončování vláken v TPE

- `corePoolSize` – cílová velikost zásobárny vláken
  - startují se, až jsou potřeba (default policy)
  - `prestartCoreThread()` – nastartuje jedno core vlákno a vrátí `boolean`, zda se povedlo
  - `prestartAllCoreThreads()` – nastartuje všechna core vlákna a vrátí jejich počet
- `maximumPoolSize` – maximální velikost zásobárny vláken
- `keepAliveTime` – doba lelkujícího života
  - od Javy 6: `allowCoreThreadTimeOut` – dovoluje timeout i core vláknům

Správa front v TPE

- Kdy se množí vlákna v TPE?
  - pokud je fronta plná
  - co se stane, pokud `corePoolSize = 0` a používáme neomezenou frontu?
- Použití synchronní fronty
  - `SynchronousQueue` není fronta v pravém slova smyslu!
  - synchronní předávání dat mezi úlohami
  - pokud žádné vlákno na předání úlohy nečeká, TPE natvoří nové
  - při dosažení limitu se postupuje podle saturační politiky
  - lze použít při neomezeném počtu vláken (`Executors.newCachedThreadPool`) nebo pokud je akceptovatelné použití saturační politiky
  - efektivní (čas i zdroje) – `Executors.newCachedThreadPool` je efektivnější než `Executors.newCachedThreadPool`, který využívá `LinkedBlockingQueue`
  - implementováno pomocí neblokujícího algoritmu v Java 6, 3× větší výkon než Java 5

Správa front v TPE

- Použití prioritní fronty
  - task musí implementovat `Comparable` (přirozené pořadí) nebo `Comparator`
- Saturační politiky
  - nastupuje v okamžiku zaplnění fronty
  - nastavuje se pomocí `setRejectedExecutionHandler` nebo konstruktoru TPE
  - `AbortPolicy` – default, úloha dostane `RejectedExecutionException`
  - `CallerRunsPolicy` – využití volajícího vlákna
    - řízení formou zpětné vazby
  - `DiscardPolicy` – vyhodí nově zaslanou úlohu
  - `DiscardOldestPolicy` – vyhodí „nejstarší“ úlohu
    - vyhazuje z hlavy front $\Longrightarrow$ nevhodné pro použití s prioritními frontami
    - pomáhá vytlačit problém do vnějších vrstev: např. pro web server – nemůže zavolat další `accept` – spojení čekají v TCP stacku

Správa front v TPE

```
        ThreadPoolExecutor tpe =
2               new ThreadPoolExecutor(1, 10, 60, TimeUnit.SECONDS,
                new LinkedBlockingQueue<Runnable>(100));
4       tpe.setRejectedExecutionHandler
                (new ThreadPoolExecutor.CallerRunsPolicy());
```

- Implementace omezení plnění fronty pomocí semaforu
  - semafor se nastaví na požadovanou velikost fronty + počet běžících úloh

```
1   @ThreadSafe
    public class BoundedExecutor {
3       private final Executor exec;
        private final Semaphore semaphore;
5
        public BoundedExecutor(Executor exec, int bound) {
7           this.exec = exec;
            this.semaphore = new Semaphore(bound);
9       }
```

Správa front v TPE

```
    public void submitTask(final Runnable command)
2           throws InterruptedException {
        semaphore.acquire();
4       try {
            exec.execute(new Runnable() {
6               public void run() {
                    try {
8                       command.run();
                    } finally {
10                      semaphore.release();
                    }
12              }
            });
14      } catch (RejectedExecutionException e) {
            semaphore.release();
16      }
    }
```

Kvízy

1. Zkuste navrhnout a implementovat thread pool, který se bude dynamický zvětšovat/zmenšovat podle počtu čekajících požadavků ve frontě.
2. Zkuste rozmyslet a navrhnout, jak by bylo možno implementovat afinitu k procesoru u Javovskych vláken a za jakých okolností by tato konstrukce fungovala.

Java NIO

- Zavedeno v Javě 1.4 (JSR 51)
- Abstraktní třída `Buffer`
  - umožňuje držet pouze primitivní typy

```
ByteBuffer
CharBuffer
DoubleBuffer
FloatBuffer
IntBuffer
LongBuffer
ShortBuffer
```

- direct vs. non-direct buffery
  - přímé buffery se snaží vyhýbat zbytečným kopiiím mezi JVM a systémem
- vytváření pomocí metod
  - `allocate` – alokace požadované velikosti
  - `allocateDirect` – alokace požadované velikosti typu direct
  - `wrap` – zabalí existující pole bytů (bytearray)

Java NIO

- ByteBuffer
  - **`http://download.oracle.com/javase/6/docs/api/java/nio/ByteBuffer.html`**
  - přístup k binárním datům, např.

```
float  getFloat()
float  getFloat(int index)
 void  putFloat(float f)
 void  putFloat(int index, float f)
```

  - mapování souborů do paměti (`FileChannel`, metoda `map`)
  - čtění/vložení z/do bufferu bez parametru index (`get`/`put`) inkrementuje pozici
  - pokud není řečeno jinak, metody vrací odkaz na buffer – řetězení volání

```
buffer.putShort(10).putInt(0x00ABBCCD).putShort(11);
```

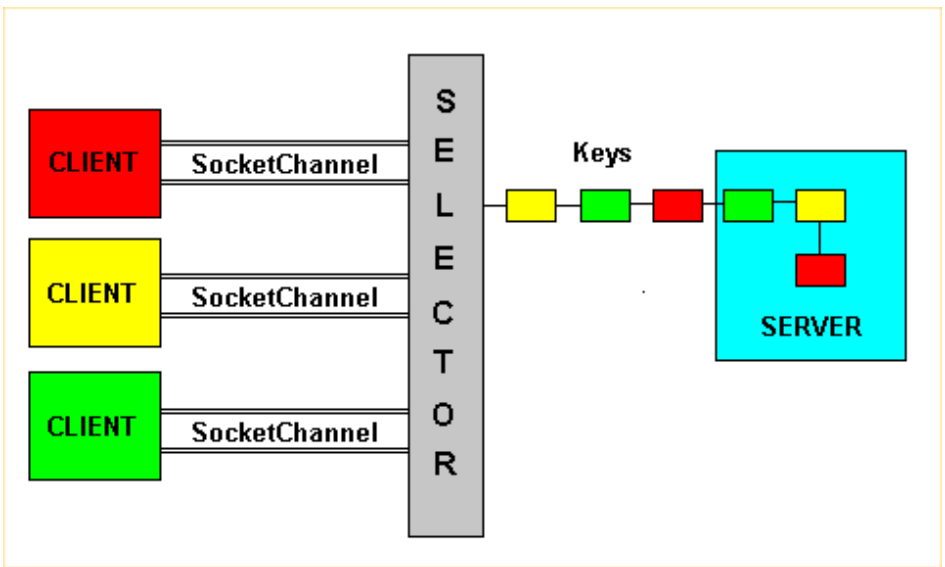
Java NIO

- Vlastnosti bufferů
  - **capacity** celková kapacita bufferu
  - **limit** umělý limit uvnitř bufferu, využití s metodami `flip` (nastaví limit na současnou pozici a skočí na pozici 0) či `remaining`
  - **mark** pomocná značka, využití např. s metodou `reset` (skočí na označkovanou pozici)

```
buffer.position(10);
buffer.flip();
while (buffer.hasRemaining()) {
byte b = buffer.get();
// neco
}
```

Java NIO

- Selektor
  - serializace požadavků
  - výběr požadavků
- Klíč
  - identifikace konkrétního spojení



Zdroj: http://onjava.com/lpt/a/2672

Java NIO – Server

- Generický postup

```
create SocketChannel;
create Selector
associate the SocketChannel to the Selector
for(;;) {
  waiting events from the Selector;
  event arrived; create keys;
  for each key created by Selector {
    check the type of request;
    isAcceptable:
      get the client SocketChannel;
      associate that SocketChannel  to the Selector;
      record it for read/write operations
      continue;
   isReadable:
      get the client SocketChannel;
      read from the socket;
      continue;
   isWriteable:
      get the client SocketChannel;
      write on the socket;
      continue;
  }
}
```

Zdroj: http://onjava.com/lpt/a/2672

Java NIO – Server

```
// Create the server socket channel
ServerSocketChannel server = ServerSocketChannel.open();
// nonblocking I/O
server.configureBlocking(false);
// host-port 8000
server.socket().bind(new java.net.InetSocketAddress(host,8000));
// Create the selector
Selector selector = Selector.open();
// Recording server to selector (type OP_ACCEPT)
server.register(selector,SelectionKey.OP_ACCEPT);
```

Zdroj: http://onjava.com/lpt/a/2672

Java NIO – Server

```
// Infinite server loop
for(;;) {
  // Waiting for events
  selector.select();
  // Get keys
  Set keys = selector.selectedKeys();
  Iterator i = keys.iterator();

  // For each keys...
  while(i.hasNext()) {
    SelectionKey key = (SelectionKey) i.next();

    // Remove the current key
    i.remove();

    // if isAccetable = true
    // then a client required a connection
    if (key.isAcceptable()) {
      // get client socket channel
      SocketChannel client = server.accept();
      // Non Blocking I/O
      client.configureBlocking(false);
      // recording to the selector (reading)
      client.register(selector, SelectionKey.OP_READ);
      continue;
    }
```

Zdroj: http://onjava.com/lpt/a/2672

Java NIO – Server

```
    // if isReadable = true
2   // then the server is ready to read
    if (key.isReadable()) {
4
      SocketChannel client = (SocketChannel) key.channel();
6
      // Read byte coming from the client
8     int BUFFER_SIZE = 32;
      ByteBuffer buffer = ByteBuffer.allocate(BUFFER_SIZE);
10    try {
        client.read(buffer);
12    }
      catch (Exception e) {
14      // client is no longer active
        e.printStackTrace();
16      continue;
      }
18
      // Show bytes on the console
20    buffer.flip();
      Charset charset=Charset.forName(''ISO-8859-1'');
22    CharsetDecoder decoder = charset.newDecoder();
      CharBuffer charBuffer = decoder.decode(buffer);
24    System.out.print(charBuffer.toString());
      continue;
26  }
  }
28}
```

Java NIO

- Další čtení:
  - **http://onjava.com/lpt/a/2672**
  - **http://onjava.com/lpt/a/5127**
  - **http://download.oracle.com/javase/6/docs/api/java/nio/channels/Selector.html**
  - **http://download.oracle.com/javase/6/docs/api/java/nio/channels/SelectionKey.html**

Asynchronní programování versus vlákna

- Asynchronní programování
  - \+ umožňuje obsluhovat řádově větší množství klientů
  - – za cenu zvýšení latence
  - – složitější, náchylnější na chyby
- Vláknové programování
  - \+ jednodušší
  - \+ poměrně efektivní do „rozumného” počtu vláken
  - – nativní vlákna nejsou stavěna na (deseti)tisíce vláken a více
- Potenciálně lze kombinovat