Nowości wprowadzone w programowaniu wielowątkowym w Javie 1.5

java.util.concurrent

Jakub Jarząbek, Jakub Janczak

Historia i geneza java.util.concurrent

java.util.concurrent:

- zapoczątkowana przez prof. Douga Lea (wraz z grupą JSR-166) w pakiecie util.concurrent, została zaadoptowana w Javie 1.5
- ma na celu poprawienie efektywności programowania rozproszonego i rozwiązania podstawowych problemów z nim związanych (przede wszystkim zapobieganiu “sytuacji wyścigu”)
- ostatnio back-portowana do Javy 1.4 (22 Oct 2004) http://altair.cs.oswego.edu/pipermail/concurrency-interes

1. Zamienniki interfejsu Runnable

W oparciu o generics stworzono interfejsy:

- *Interface Callable<V>* - w odróżnieniu do *Runnable* po uruchomieniu (metoda *V Call())* zwraca wynik, a w razie niepowodzenia zrzuca *Exception*
- *Interface Future<V>* - interfejs obsługujący asynchroniczne wykonanie zadań. Oferuje metody:
  - *boolean isDone()* - sprawdza czy zadanie zostało wykonane
  - *boolean isCancelled()* - sprawdza czy zadanie zostało anulowane
  - *V get()* - pobranie wyniku
  - *cancel()* - anuluje wykonanie zadania

Zamienniki interfejsu Runnable cd..

- *Delayed* – interfejs zapewniający odkładanie zadań w czasie na podstawie rozszerzeń obsługi czasu w JVM (o tym później)
- rozszerzeniem *Future*<*V*> i *Delayed* jest interfejs *ScheduledFuture*<*V*>, pozwalający operować na “odłożonym” zadaniu.

Po co to??? (czyli Executors)

W wersji 1.5 wprowadzono interfejs Executor, który ma tylko jedną metodę: *execute(Runnable zadanie),* która niezwłocznie uruchamia zadanie. Pozwala to na zrobienie czegoś takiego:

```
class ThreadPerTaskExecutor implements Executor {
    public void execute(Runnable r) {
        new Thread(r).start();
    }
 }
```

Właściwie to nic nowego – taką metodę odpalania zadan juz znamy, ale sam interfejs *Executor* służy tylko jako baza dla potężnych rozszerzeń korzystających z zamienników Runnable, takich jak...

ExecutorService

Interfejs zapewnia:

- metodę *submit()* operującą na obiektach typu Runnable lub Callable<V>, która rozkazuje wykonać zadanie zwracając przy tym obiekt Future<V> (Jak już wcześniej wspomniano pozwala on nam na sprawdzenie czy zadanie już się wykonało, czy może stało sie z nim coś czego wcale byśmy nie chcieli ;).
- metody *InvokeAll()* i *InvokeAny()* nakazujące odpalenie wszystkich zadań podanych jako parametr (w postaci kolekcji) i zwrócenie wszystkich wyników w postaci listy (InvokeAll()) lub pierwszego wykonanego (InvokeAny())
- metody do zestrzeliwania executorów: Shutdown...() ;)

ImIplementacje ExecutorService

Na podstawie poznanego interfesju stworzono kilka implementacji ExecutorService takich jak:

- *ThreadPoolExecutor* – potężna (!) klasa obiektów realizujących żądane zadania w obrębie zadanej puli wątków, ich podstawowe cechy to:
  - uruchamianie zadań w wolnych wątkach lub tworzenie nowych (można to ograniczyć)
  - automatyczne “ubijanie” bezczynnych wątków po określonym czasie
  - możliwość określania polityki odrzucania zadań (np. wtedy kiedy nie można stworzyć więcej wątków)

ImImplementacje ExecutorService cd..

- możliwość wykonywania zadanych funkcji przed i po wykonaniu zakolejkowanych zadań (uff...)
- możliwość operowania na zakolejkowanych zadaniach (dobre do debugowania)

- *ScheduledThreadPoolExecutor* – ThreadPoolExecutor rozszerzony o możliwość odkładania zadań w czasie lub uruchamiać je co zadany okres czasu.

Klasa Executors

Powstała, aby ułatwić manipulowanie zadaniami, stanowi także fabrykę dla typowych executorów (a właściwie *ExecutorService*). Posiada takie “metody fabryczne” jak:

- *newCachedThreadPool* – zwraca executora który pozwala swoim wątkom “żyć” trochę dłużej w nadzieji że jeszcze się przydadzą (jeśli nie ubija je)
- *newFixedThreadPool* – zwraca executora który utrzymuje ściśle określoną ilość wątków
- *newScheduledThreadPool* – zwraca executor zadan odłożonych na później lub periodycznych
- *newSingleThreadExecutor* i *newSingleThreadScheduledExecutor* – zwraca executory jednowątkowe

Przykładowe wykorzystanie executorów

```
class NetworkService {
    private final
        ServerSocket serverSocket;
   private final ExecutorService pool;

public NetworkService(int port,
    int ileWątków) throws IOException {
   serverSocket =
    new ServerSocket (port);
   pool = Executors.newFixedThreadPool
    (ileWątków); // tutaj tworzymy ex.
     }
```

```
public void serve() {
    try {
    for (;;) {
    pool.execute(
   new Handler (serverSocket.accept()));
    }
    } catch (IOException ex) {
        pool.shutdown();
    }
   }
} // NetworkService ~koniec
class Handler implements Runnable {
    private final Socket socket;
    Handler(Socket socket) {
   this.socket = socket;
   }
    public void run() {
      // obsługujemy klienta
   }
}
```

Obsługa czasu (timing)

Realizowana przez *TimeUnit*, oferuje nam następujące własności:

- Wbudowane jednostki czasu: sekundy, milisekundy, mikrosekundy, nanosekundy i konwersje między nimi (metoda *convert()*)
- Dokonanie operacji *Thread.wait()*, *Thread.join()* i *Thread.sleep()* przez bardzo dokładnie określony okres czasu

Klasa *TimeUnit* jest szeroko wykorzystywana jako argument metod klas java.util.concurrent (szczególnie – zadania *Delayed*)

Synchronizacja danych

Obiekty tworzące blokady synchronizacyjne:

- *Semaphore* (klasyczne narzędzie):
  - inicjowany pulą zezwoleń, przydziela je aż do wyczerpania, następnie blokuje obiekt, aż do odzyskania wystarczającej ilości zezwoleń by dopuścić nowy
  - zainicjowany jedynką służy za tzw. “mutual exclusion”
  - można sprawić, że *semafor* będzie się zachowywał nieuczciwie i przydzielał zasób losowo

Synchronizacja cd..

- *CountDownLatch*:
  - zainicjowany daną liczbą n, blokuje wątki do momentu aż wykona się n operacji *countDown()*
  - można go użyć aby n wątków poczekało na siebie lub aby pewna operacja została wykonana n razy
  - *CountDownLatch* jest obiektem “jednorazowego użytku”
- *CyclicBarrier:*
  - klasa bardzo podobna do *CountDownLatch*

Synchronizacja cd..

- obiekt tego typu czeka aż n wątków wykona na nim await(), zaraz po tym odblokowuje wątki (licznik zostaje wyzerowany)
- przydatny przy blokowaniu operacji których wyniki mają na siebie bezpośredni wpływ np. pobieranie informacji z kilku miejsc i łączenie ich w jedną całość
- *Exchanger* – pozwala dwóm wątkom wymieniać kontrolę nad danym obiektem.

CyclicBarrier - przykład

```
class Solver {
  final int N;
  final float[][] data;
  final CyclicBarrier barrier;

  class Worker implements Runnable {
    int myRow;
    Worker(int row) { myRow = row; }
    public void run() {
      while (!done()) {
        processRow(myRow);

        try {
          barrier.await();
        } catch (InterruptedException ex) {
          return;
        } catch (
      BrokenBarrierException ex ) {
                return;
        }
      }
    }
  }

public Solver(float[][] matrix) {
    data = matrix;
    N = matrix.length;
    barrier = new CyclicBarrier(N,
                                new Runnable() {
                                  public void run() {
                                    mergeRows(...);
                                  }
                                });
    for (int i = 0; i < N; ++i)
      new Thread(new Worker(i)).start();

    waitUntilDone();
  }
}
```

Kolejki współbieżne

java.util.concurrent udostępnia pięć kolejek różnego typu specjalnie utworzonych dla programowania współbieżnego :

- *ConcurrentLinkedQueue* – standardowa kolejka FIFO o dostępie asynchronicznym (z tego powodu metoda *size()* nie jest stałoczasowa)
- Blokujące się kolejki, bezpieczne w aplikacjach wielowątkowych z powodu metody z jaką umieszcza się w nich elementy (atomic locks):
  - *LinkedBlockingQueue* – blokująca kolejka FIFO
  - *ArrayBlockingQueue* – blokująca kolejka FIFO oparta na statycznej tablicy
  - *PriorityBlockingQueue* – blokująca kolejka priorytetowa (czyli taka która szereguje elementy według określonego algorytmu)

Kolejki współbieżne cd..

- *DelayedQueue* – kolejka blokująca się w której dostępne są tylko te elementy (typu Delayed), których delay upłynął (szeregowane są one w ten sposób, że im dawniej to się stało tym wyżej stoją w hierarchii kolejki)
- *SynchronousQueue* – kolejka w której każdy *put()* musi poczekać na *take(),* czyli generalnie rzecz biorąc ciągle jest pusta. (może służyć jako kanał wymiany między dwoma wątkami)

Nowe Kolekcje

Poza kolejkami java.util.concurrent wprowadziła trzy inne interesujące kolekcje:

- *ConcurrentHashMap* – tablica rozproszona, do której pisać może jednocześnie określona liczba wątków (domyślnie 16). Odczytywanie z kolekcji jest nieograniczone.
- *CopyOnWriteArrayList* – *ArrayList* w którym każda zmiana jest wprowadzana do “świeżej” kopii tablicy (przez to iteratory są aktualne tylko do pierwszej zmiany w kolekcji, poza tym nie pozwalają one na zmiany danych)
- *CopyOnWriteArraySet* – zbiór oparty na *CopyOnWriteArray*List ze wszystkimi tego konsekwencjami

Bibliografia (linki ;))

- Dokumentacja API java.util.concurrent
- Strona domowa Douga Lea
- Strona domowa JSR-166
- Dokumentacja util.concurrent dla Javy 1.4

Artykuły:

- JavaPro o java.util.concurrent
- Artykuł Davida Wheelera o zapobieganiu Race Condition

Książka:

podobno dobra (4 gwiazdki na Amazon.com):
Concurrent Programming in Java by Doug Lea

Podsumowanie

- Historia i geneza java.util.concurrent
- Zamienniki interfejsu Runnable
- Executors
- ExecutorService
- ImIplementacje ExecutorService
- Klasa Executors
- Przykładowe wykorzystanie executorów
- Obsługa czasu (timing)
- Synchronizacja danych
- CyclicBarrier - przykład
- Kolejki współbieżne
- Nowe Kolekcje
- Bibliografia