8 RZECZY, KTÓRYCH TWOI KOLEDZY NIE WIEDZĄ O WĄTKACH W JAVIE

Dariusz Mydlarz
sztukakodu.pl

Cześć. Dobrze, że jesteś!

Jeżeli jesteś jak ja, to:
1) lubisz poznawać szczegóły technologii, w której pracujesz,
2) chciałbyś lepiej rozumieć jak działa współbieżność w Javie i na co zwracać uwagę.

Na podstawie mojego wieloletniego doświadczenia pracy z aplikacjami w Javie, przygotowałem dla Ciebie dokument prezentujący 8 rzeczy, których Twoi koledzy nie wiedzą o współbieżności.

Dzięki niemu dowiesz się jakie błędy często popełniają i jak ich uniknąć w przyszłości. Stosując się do nich sprawisz, że Twoje aplikacje będą po prostu bezpieczniejsze i nie będą powodować frustracji w przypadku nieoczekiwanych błędów na produkcji.

Jeżeli tylko tworzysz aplikacje webowe, to na pewno znajdziesz tu coś dla siebie.

Pozdrawiam,

Dariusz Mydlarz
Sztuka Kodu
darek@sztukakodu.pl

1. Wątki typu "daemon"

W Javie występują dwa rodzaje wątków: użytkownika i daemon. Maszyna wirtualna Javy przetwarza aplikację do momentu aż wszystkie wątki użytkownika nie zakończą swojej pracy.

Wątki daemon nie mają takiego działania. Nawet jeśli nadal są przy życiu, aplikacja może się zatrzymać.

Warto je wykorzystać do wykonywania zadań w tle, takich jak obsługa zadań z kolejki, wykonywanie periodycznych akcji, czy regularnego odpytywania zewnętrznych serwisów.

W JDK wątki daemon używane są przy sprzątaniu śmieci (garbage collection), zwalnianiu pamięci nieużywanych obiektów i usuwaniu starych wpisów z cache-ów.

Aby stworzyć wątek typu daemon należy zawołać metodę *Thread#setDaemon(true):*

```
Thread t1 = new Thread(...)
t1.setDaemon(true);
t1.start();
```

Aby sprawdzić czy wątek jest typu daemon wołamy z kolei *Thread#isDaemon()*

```
boolean isDaemon = t1.isDaemon();
```

2. Nazywanie wątków

Każdy wątek uruchomiony w wirtualnej maszynie Javy posiada swoją nazwę. Domyślne nazwy nadawane wątkom niewiele o nich mówią, na przykład: *Thread-0*, *Thread-1*, itd.

Jeśli wątki uruchamiane są w pulach wątków - Executorach (o nich za chwilę), to w najlepszym wypadku ich nazwy będą wyglądać podobnie do: *pool-1-thread-1*.

Odpowiednie nazywanie wątków jest kluczowe w późniejszym debugowaniu aplikacji i wyszukiwaniu błędów na produkcji, podczas przeglądania logów.

W pojedynczych wątkach do nadawania nazwy służy metoda *Thread#setName*.

```
Thread t1 = new Thread(...);
t1.setName("Email-Sender-t0");
```

W pulach wątków (Executorach) można ustawiać nazwy poprzez przekazywanie odpowiedniej fabryki wątków - *ThreadFactory*.

Aby ją zbudować najlepiej skorzystać z klasy *ThreadFactoryBuilder* pochodzącej z darmowej biblioteki **Guava** i użyć metody *setNameFormat(...)* jak w przykładzie poniżej. Wzorzec *"email-sender-t%d"* spowoduje tworzenie kolejnych wątków z nazwą: *email-sender-t0*, *email-sender-t1*, itd.

```
ThreadFactory threadFactoryBuilder = new ThreadFactoryBuilder()
   .setNameFormat("email-sender-t%d")
   .build();
Executor executor = Executors.newFixedThreadPool(10, threadFactoryBuilder);
executor.execute(runnable);
```

Widząc taki wątek w logach, dużo łatwiej będzie można doszukiwać się źródeł potencjalnych błędów.

3. Pule wątków (Executors)

Samodzielne tworzenie wątków nie jest najlepszym pomysłem. Trudno nimi zarządzać, obserwować, a ich nadmierna liczba może prowadzić do problemów z wydajnością aplikacji.

Lepiej jest korzystać z pól wątków, dzięki którym aplikacja będzie reużywać konkretnych instancji do przetwarzania zadań.

Tworzenie nowego wątku w aplikacji jest kosztowne, gdyż wiąże się z jego fizycznym utworzeniem w systemie operacyjnym. Każdorazowe tworzenie nowych wątków do wykonywania zadań w aplikacji generowałoby duży narzut zarówno na aplikację, jak i system operacyjny.

Lepszym rozwiązaniem jest korzystanie z grupy wcześniej utworzonych wątków, i dzieleniu ich na różne zadania wykonywane w aplikacji. Łatwiej wtedy dbać o limity i nie doprowadzać do problemów wydajnościowych.

Java od wersji 1.5 udostępnia dwa interfejsy służące temu celowi: *Executor* i *ExecutorService*.

4. Executor, ExecutorService i Executors

Executor to prosty interfejs posiadający tylko jedną metodę *execute*.

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

ExecutorService jest jego specjalizacją i udostępnia dużo więcej metod służących do uruchamiania zadań i zarządzania całą pulą, takich jak między innymi:

```
public interface ExecutorService extends Executor {
    void execute(Runnable command);
    void shutdown();
    <T> Future<T> submit(Callable<T> task);
    Future<?> submit(Runnable task);
    <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)
throws InterruptedException;
 // ...
}
```

Dodatkowym ważnym elementem całej układanki jest fabryka *Executors* ułatwiająca tworzenie nowych pól wątków. Za jej pomocą możemy utworzyć takie pule jak:

- *newSingleThreadExecutor* - pula wątków z jednym wątkiem,
- *newFixedThreadPoolExecutor* - pula wątków ze stałą liczbą wątków,
- *newCachedThreadPoolExecutor* - pula wątków korzystająca z od 0 do maksymalnie 2 mld wątków (*Integer.MAX_VALUE*). Utrzymuje przy życiu oczekujące wątki przez 60 sekund od ostatniego uruchomienia.

4. Executor, ExecutorService i Executors - c.d.

Warto także wspomnieć o **SchedulerExecutorService**, który służy do uruchamiania zadań w określonym czasie, a także cyklicznych. Jego instancje można utworzyć za pomocą *Executors.newScheduledThreadPool*.

Udostępnia on między innymi następujące metody:

- *schedule* - uruchamia zadanie jeden raz po upływie określonego czasu,
- *scheduleAtFixedRate* - uruchamia cykliczne zadanie co określoną ilość czasu,
- *scheduleWithFixedDelay* - uruchamia cykliczne zadanie co określoną ilość czasu licząc od zakończenia poprzedniego wykonania.

Oznacza to, że *scheduleAtFixedRate* będzie wykonywał zadania zawsze co określoną ilość czasu, np. co 10 sekund, a *scheduleWithFixedDelay* będzie wykonywał kolejne zadanie 10 sekund po ukończeniu poprzedniego.

5. ExecutorService: *submit* a *execute*

ExecutorService udostępnia dwa rodzaje metod służących do wykonania zadań: *submit* i *execute*.

```
public interface ExecutorService extends Executor {
    void execute(Runnable command);
    <T> Future<T> submit(Callable<T> task);
    <T> Future<T> submit(Runnable task, T result);
    Future<?> submit(Runnable task);
    // ...
}
```

Metoda *execute* pochodząca z interfejsu *Executor* przyjmuje jako argument typ *Runnable* i nie zwraca żadnej wartości. Wysyłając zadanie w ten sposób tracimy możliwość sprawdzenia jego wyniku i zobaczenia co się z nim stało.

Metoda *submit* natomiast zwraca klasę *Future* na której możemy zawołać metodę *get* by uzyskać rezultat przekazanego zadania, a także obsłużyć wyjątek w przypadku jego wystąpienia (więcej o tym w następnym punkcie).

Kiedy więc używać której? Gdy wysyłasz rutynowe zadanie, które ma się wykonać kiedyś w tle (ale niekoniecznie interesuje cię jego wynik), wystarczy zawołać metodę *execute*. Może to być na przykład wysłanie jakiejś metryki, zapisanie logów, i tym podobne. Jeśli natomiast musisz mieć pewność co do rezultatu wykonanej akcji, wtedy zdecydowanie powinieneś korzystać z metody *submit* i sprawdzić jej wynik.

6. Łapanie wyjątków

Metody *execute* i *submit* mają też duże implikacje jeśli chodzi o obsługiwanie wyjątków. Jeśli uruchomisz poniższy kod przekonasz się, że błąd łapany jest tylko jeden raz. I dotyczy to metody *Executor.execute(...)*.

```
Runnable runnable = () -> {
   throw new IllegalStateException("Oops, I am sorry...");
};
ExecutorService executor = Executors.newFixedThreadPool(10);
executor.execute(runnable);
executor.submit(runnable);
```

W przypadku metody *execute* obsługa błędów przekazywana jest do instancji klasy *UncaughtExecutionHandler* znajdującej się w *Executorze*.

Jeśli chcesz zastosować dedykowaną obsługę błędów, możesz przekazać własny handler wyjątków:

```
ThreadFactory factory = new ThreadFactoryBuilder()
   .setUncaughtExceptionHandler(new Thread.UncaughtExceptionHandler() {
       @Override
       public void uncaughtException(final Thread t, final Throwable e) {
           System.err.println("An error during execution: " +
                   e.getMessage() + ", thread: " + t.getName());
           e.printStackTrace();
       }
    })
   .build();
ExecutorService executor = Executors.newFixedThreadPool(10, factory);
```

6. Łapanie wyjątków - c.d.

Teraz, w przypadku wystąpienia błędu wykonana zostanie Twoja dedykowana logika. Możesz w niej umieścić na przykład logowanie, wysyłanie alertów, czy mierzenie częstości występowania.

W przypadku metody *submit* łapanie błędów odbywa się dopiero w momencie zawołania metody *get()* na zwróconym obiekcie *Future*.

Jakie ma to konsekwencje? W sytuacji, gdy korzystasz z tej metody, musisz się upewnić, że zawsze będziesz sprawdzał wynik z *Future*. W przeciwnym wypadku może okazać się, że Twoje zadania kończą się błędem, a system w żaden sposób o tym nie informuje, nie zostawiając nawet wpisów w logach.

Jak to wygląda w praktyce? Zobacz kod poniżej. Zauważ, że blok *try-catch* został założony na wywołaniu *future.get()* a nie w chwili wysyłania zadania do *Executora*.

```
Future<String> future = executor.submit(callable);
try {
   String result = future.get();
   System.out.println("Result is: " + result);
} catch (ExecutionException e) {
   System.err.println("Execution exception on: " + e.getMessage());
   e.printStackTrace();
}
```

7. Pomijanie timeoutu

Bardzo często zdarza się, że programiści korzystają z metod blokujących wywołanie kodu w nieskończoność. Takie metody to np. *Lock.try()*, *Future.get()*, czy *CountDownLatch.await()*. Popatrz na przykład.

```
Lock lock = ...;
lock.lock(); // blokuje w nieskończoność :( - NIE RÓB TEGO
try {
  // praca do wykonania
} finally {
  lock.unlock();
}
```

Jeśli metoda *lock.lock()* nie będzie mogła uzyskać *locka*, wówczas zablokuje wykonywanie programu w nieskończoność.

Zamiast tego, lepszym podejściem jest wołanie ich odpowiedników z parametrem `timeout` mówiącym ile maksymalnie można czekać na wywołanie metody.
Dla powyższych metod będą to: *Lock.tryLock(long time, TimeUnit unit)*, *Future.get(long time, TimeUnit unit)* oraz *CountDownLatch.await(long time, TimeUnit unit)*.

Jak powinien wyglądać powyższy przykład? W ten sposób.

```
Lock lock = ...;
if(lock.tryLock(100, TimeUnit.MILLISECONDS)) {
  try {
    // praca do wykonania
  } finally {
    lock.unlock();
  }
}
```

Za każdym razem sprawdzaj na liście metod danej klasy, czy nie posiada ona odpowiednika pozwalającego przekazać timeoutu - i jeśli tak, zawsze z niego skorzystaj.

8. Pułapka *parallelStream()*

Na koniec wisienka. Czy wiedziałeś, że korzystanie z *parallelStream* to najkrótsza droga do problemów?

Parallel Streamy, które weszły do języka w Javie 8, były obietnicą łatwego korzystania ze współbieżności dla każdego programisty. W praktyce okazuje się, że można sobie nimi narobić więcej problemów niż korzyści.

Przetwarzanie parallel streamów odbywa się przy pomocy wspólnej puli wątków dla całej aplikacji. Nie można jej przekazać samodzielnie, nie można używać różnych pul w różnych miejscach. Twórcy Javy zmusili nas do korzystania z jednej puli, którą jest *ForkJoinPool#commonPool()*.

Pula ta posiada liczbę wątków odpowiadającą liczbie rdzeni procesora. Wszystkie wywołania *parallelStream()* w aplikacji konkurują de facto o tę jedną pulę. Nie ważne czy dotyczą przetwarzania danych z bazy danych, odpytywania zewnętrznych serwisów, zapisywania wiadomości do systemów kolejkowych, czy przeliczania modeli matematycznych.

Jeśli nie chcesz mieć problemów, po prostu zapomnij o tej metodzie i używaj dedykowanych pól wątków, którymi sam zarządzasz i które sam monitorujesz.

Dziękuję

Mam nadzieję, że dowiedziałeś się czegoś nowego o wątkach. Jeśli masz dodatkowe pytania napisz do mnie na **darek@sztukakodu.pl**. Odpisuję na każdy email ;)

Pozdrawiam
Darek Mydlarz
Założyciel Sztuki Kodu