Receptury projektowe – niezbędnik początkującego architekta

Część VIII: Zarządzanie transakcjami w systemach klasy enterprise

<<LEAD>>
Do czego może przydać mi się propagacja transakcji inna niż REQUIRED? Jak zachowa się wówczas EnityManager i cache pierwszego poziomu? Jak uniknąć zakleszczeń? Dlaczego moje transakcje tylko-do-odczytu nie są tylko do odczytu? Kiedy oddać sterowanie transakcjami klientom zamiast obsługiwać je aspektowo? Jakie anomalie w spójności danych mi zagrażają?
Odpowiem na te problemy i opiszę ich rozwiązania w kontekście Spring oraz Java EE, jednak użytkownicy innych platform znajdą w artykule również coś dla siebie: wskazanie generycznych problemów i ich rozwiązań.
<</LEAD>>

<<RAMKA>>

O serii „Receptury projektowe”

Intencją serii „Receptury projektowe” jest dostarczenie usystematyzowanej wiedzy bazowej początkującym projektantom i architektom. Przez projektanta lub architekta rozumiem tutaj rolę pełnioną w projekcie. Rolę taką może grać np. starszy programista lub lider techniczny, gdyż bycie architektem to raczej stan umysłu niż formalne stanowisko w organizacji.

Wychodzimy z założenia, że jedną z najważniejszych cech projektanta/architekta jest umiejętność klasyfikowania problemów oraz dobieranie klasy rozwiązania do klasy problemu. Dlatego artykuły będą skupiały się na rzetelnej wiedzy bazowej i sprawdzonych recepturach pozwalających na radzenie sobie z wyzwaniami niezależnymi od konkretnej technologii, wraz z pełną świadomością konsekwencji płynących z podejmowanych decyzji projektowych.
<</RAMKA>>

Po co mi to?

Środowisko Enterprise Java (rozumiane jako Java EE lub Spring) oferuje elastyczny model zarządzania transakcjami pod względem poziomów izolacji, propagacji i ogólnej strategii zarządzania. W prostych przypadkach możemy zdać się na wygodne zarządzanie transakcjami poprzez aspekty lub interceptory z domyślnymi ustawieniami. Wszystko działa dopóki nie zaczynamy realizować złożonych scenariuszy pod większym obciążeniem. Okazuje się wówczas, że w systemie pojawiają się dziwne zjawiska: zakleszczenia, spadek wydajności oraz co najgorsze: niespójne dane.

Wprowadzenie

Zaczniemy od wstępu teoretycznego, w którym zdefiniuję podstawowe pojęcia oraz nazwę problemy. Wstęp jest na potrzeby, aby w dalszej części świadomie wybrać model i strategię transakcji oraz dobrać politykę propagacji transakcji do sytuacji. Czytelnicy, którzy znają pojęcia: ACID, anomalie spójności danych i poziomy izolacji mogą pominąć ten rozdział i przejść od razu do rozdziału poświęconego modelom transakcji.

ACID

Od bazy danych oczekujemy następujących cech niezawodności:

- **Atomic** – wszystkie operacje w danej transakcji zostaną wykonane jako spójna jednostka zmiany lub nie zostaną wykonanie w ogóle
- **Consistent** – transakcja zmieni dane z jednego spójnego stanu w inny spójny stan (respektując constrains), nie naruszając integralności
- **Isolated** – współbieżne transakcje nie oddziaływają między sobą – w jakim stopniu, to zależy od poziomu izolacji (w dalszej części)
- **Durable** – transakcja zatwierdzona pozostawia trwałe zmiany odporne na późniejsze usterki bazy danych

Anomalie

W przypadku, gdy dwie transakcje T1 i T2 operują na wspólnych danych, możemy spodziewać się różnych anomalii w spójności danych:

- **dirty read** – T1 odczytuje dane zmienione przez T2, nawet gdy w późniejszym czasie T2 jest wycofana. W efekcie T1 „widzi" nieistniejące dane
- **unrepeatable read** – T1 wykonuje kilkukrotnie to samo zapytanie, ale otrzymuje inne dane (w sensie taka sama ilość wierszy, ale inne wartości w wierszach). Dzieje się to z powodu tego, że T2 modyfikuje te dane
- **phantoms -** T1 wykonuje kilkukrotnie to samo zapytanie, ale otrzymuje inną ilość danych (w sensie innej ilości wierszy). Dzieje się to z powodu tego, że T2 modyfikuje te dane.

O ile dane fantomowe czy odczyty niepowtarzalne mogą być akceptowalne w dużej mierze przypadków, o tyle brudne odczyty są poważnym problemem dla spójności danych.

Poziomy izolacji

Z powodu powyższych anomalii chcemy sterować poziomami izolacji transakcji T1 i T2, które wykonują się współbieżnie:

- **read_uncommited** – niezatwierdzone przez T1 zmiany są widoczne w T2, występują wówczas anomalie: dirty-reads, non-repeatable-reads, phantoms
- **read_commited** – zmiany poczynione przez T1 są widoczne dopiero po jej zatwierdzeniu (commit), występują wówczas anomalie: non-repetable-reads, phantoms
- **repeatable_read** – pobrane dane są blokowane, więc kolejne odczyty zwracają te same dane, możliwe anomalie: phantoms
- **serializable** – transakcje serializowane, nie występują żadne anomalie

Domyślny poziom izolacji zależy od konkretnej bazy danych. Oczywiście im większy poziom izolacji, tym większy koszt.

Springu lub Java EE pozwala na ustawienie poziomu izolacji transakcji, np. w sposób deklaratywny w adnotacji @Transactional – Listing 1.

Strategie transakcji

W złożonym systemie możemy zdecydować się na wybór następujących strategii zarządzania transakcjami:

Strategie zarządzane przez API

Strategia transakcji zarządzanych w warstwie API jest najbardziej powszechna i stosowana, gdy projektujemy gruboziarniste metody, realizujące całościowe, konkretne Use Case – Listing 1. Najwygodniej przyjąć wówczas model deklaratywny i propagację REQUIRED.

Orkiestrowanie przez kod kliencki

Możemy przyjąć również zupełnie odmienny styl naszego API. Nie dostarczamy całościowych Use Case a jedynie drobnoziarniste metody, z których kod kliencki (np. silnik Business Process Management) „układa" wywołanie własnych scenariuszy.

Nie chcemy przygotowywać wysokopoziomowego API (które samo zarządza transakcjami), aby nie usztywniać systemu. Oddajemy wówczas sterowanie transakcjami stronie klienckiej – to ona odpowiada za rozpoczęcie, zatwierdzenie i ew. wycofanie transakcji pomiędzy wywołaniami naszego drobnoziarnistego API.

Implementacja tej strategii opiera się na propagacji MANDATORY opisanej w dalszej części artykułu.

Strategia zorientowana na wysoką współbieżność

Strategia ta stosowana jest, gdy transakcje nie mogą długo działać na wysokości warstwy API ze względu na swój koszt. Głównym założeniem tej strategii transakcji jest skrócenie zakresu transakcji, tak aby zminimalizować blokady w bazie danych przy zachowaniu atomowości dla danego żądania klienta. Implementacja opiera się na propagacji REQUIRES_NEW i sterowaniu

poziomem izolacji.

Strategia szybkiego przetwarzania

Najbardziej ekstremalna strategia używana, aby uzyskać bezwzględnie najkrótszy czasie przetwarzania (a więc wydajność) aplikacji i nadal utrzymać pewien stopień atomowości. Generalnie sprowadza się do unikania transakcji i tworzenia mechanizmów kompensacji na wypadek błędów.

Propagacja transakcji

Wprowadzenie do konwencji

Przy deklaratywnym sterowaniu transakcjami (aspekty konfigurowane adnotacjami, które komunikują się z managerem transakcji) obowiązuje konwencja:

- wyjątki checked (dziedziczące po Exception) nie wycofują transakcji – wyjątek taki oznacza błąd, który możemy próbować naprawić, więc nie ma powodu do wycofania transakcji
- wyjątki unchecked (dziedziczące po RuntimeExcepion) wycofują transakcję – wyjątek taki oznacza katastrofę, której naprawa nie ma sensu, dlatego wycofujemy transakcję

Od kilku lat promowane jest unikanie wyjątków checked, dlatego możemy określić, które wyjątki mają wycofywać transakcje, a które nie.

Warto w Springu stworzyć własną adnotację, aby nie powtarzać tych deklaracji w każdym miejscu – Listing 1. Od tej pory możemy posługiwać się własną adnotacją @ ApplicationService.

<<LISTING lang=java>>

```
@Service
@Transactional(rollbackFor=CriticalException.class,
        noRollbackFor=ItsCoolException.class)
@Retention(RUNTIME)
@Target(TYPE)
```

```
public @interface ApplicationService{}

@ApplicationService
public class MyAPIService{

}
```

<<</LISTING>>

Listing 1. Własna adnotacja dla serwisów aplikacyjnych, która deklaruje, jakie wyjątki powinny wycofywać transakcje, a jakie nie

Łańcuch wywołań

Co stanie się, gdy jeden serwis będący pod ochroną aspektu transakcyjnego wywoła inny serwis będący również pod ochroną takiego aspektu?

Bo jeżeli metoda wywoła metodę z tej samej klasy, to wówczas sytuacja jest obsłużona bez dodatkowych ceremonii. Jeżeli natomiast wywołanie robi „przeskok” do innego obiektu, to mamy wówczas do czynienia ze zjawiskiem propagacji transakcji. W prostych przypadkach możemy polegać na domyślnej polityce propagacji i nie zastanawiać się nad problemem. Istnieje natomiast siedem polityk, z czego warto rozważyć stosowanie sześciu.

Omówimy teraz kolejno sześć przykładowych scenariuszy propagacji transakcji wraz z praktycznymi przykładami klas problemów, jakie można dzięki nimi rozwiązać.

REQUIRED

Jest to domyślna polityka, która zakłada, że jeżeli metoda rozpoczyna transakcję, to musi również ją zakończyć (commit/rollback). Jeżeli transakcja nie istnieje, to wówczas jest tworzona.

Na poziomie bazy danych istnieje jedna transakcja fizyczna. Natomiast na poziomie frameworka Spring podczas propagacji są tworzone osobne logiczne transakcje – dla każdej z wywoływanych metod. Jeżeli jakaś metoda zwróci

odpowiedni wyjątek (patrz rozdział „Wprowadzenie do konwencji”), to w transakcji logicznej jest ustawiany marker rolbackOnly. Teraz, gdy marker jest ustawiony, to gdy sterowanie wraca do metody, która rozpoczęła transakcję (a właściwie Aspekt, który ją okala), wycofa całość.

Zewnętrzna metoda może obsłużyć wyjątek wewnętrznej, tak jak na Listingu 2. Mimo tego, że wyjątek został złapany w metodzie nadrzędnej `testRequired`, to jednak metoda podrzędna `doRequired` wyrzuciła wyjątek, który wg konwencji oznacza transakcję logiczną jako do wycofania. Zatem na końcu metody nadrzędnej `testRequired` transakcja fizyczna będzie wycofana.

<<LISTING lang=java>>

```
@Service
public class OuterService{
  @Autowired
  private MyDAO myDAO;

  @Autowired
  private InnerService innerService;

  @Transactional(propagation=Propagation.REQUIRED)
  public void testRequired(User user) {
    myDAO.insertUser(user);
    try{
      innerService.doRequired();
    } catch(RuntimeException e){
      // handle exception
    }
    //dane zapisane przez myDAO nie będą zatwierdzone w bazie. Mimo
przechwycenia wyjątku transakcja fizyczna będzie wycofana z powodu
wyjątku unchecked
  }
}
//==================================

@Service
public class InnerService{

  @Transactional(propagation=Propagation.REQUIRED)
  public void doRequired() {
    throw new RuntimeException("Rollback this transaction!");
  }
}
```

<</LISTING>>

Listing 2: Propagacja REQUIRED

REQUIRES_NEW

Polityka, która wymusza założenie osobnej transakcji fizycznej. Jeżeli transakcja istnieje, to jest zawieszana. Zewnętrzna metoda może kontynuować pomimo rollback wewnętrznej – jeżeli przechwyci wyjątek. Możemy zatem zrealizować dwa scenariusze:

- metoda wewnętrzna wykona się z powodzeniem (dane będą zapisane), a metoda zewnętrza w późniejszym czasie zakończy się błędem (wycofanie tylko danych zapisanych przez nią)
- metoda wewnętrzna zakończy się błędem (i wycofaniem danych), ale metoda zewnętrzna będzie kontynuowana, a dane przez nią zapisane zostaną zatwierdzone

Należy pamiętać, że całość działa jedynie przy wywołaniu metod z innych obiektów. Wołanie metody z tego samego obiektu nie przynosi skutku.

<<LISTING lang=java>>

```
@Service
public class OuterService{
  @Autowired
  private MyDAO myDAO;

  @Autowired
  private InnerService innerService;

  @Transactional(propagation=Propagation.REQUIRED)
  public void testRequired(User user) {
    myDAO.insertUser(user);
    try{
      innerService.doRequiresNew();
    } catch(RuntimeException e){
      // handle exception
    }
    //wyjątek unchecked wycofa transakcję wewnętrzną, ale ponieważ jest przechwycony, to nie wpłynie na transakcję zewnętrzną.
  }
}
//==================================
```

```
@Service
public class InnerService{

  @Transactional(propagation=Propagation.REQUIRES_NEW)
  public void doRequiresNew() {
    throw new RuntimeException("Rollback this transaction!");
  }
}
```

<</LISTING>>

Listing 3. Propagacja REQUIRES_NEW

REQUIRES_NEW – skutki separacji w JPA

Wywołując metody z propagacją REQUIRES_NEW, zawsze stworzona zostanie nowa transakcja.

Jeżeli używamy JPA, to pracujemy z nową instancją EntityManager, czyli mamy do czynienia z nowym, osobnym cache pierwszego poziomu (L1 cache).

Musimy pamiętać o konsekwencjach:

- Dane pobrane przez metodę zewnętrzną nie znajdują się w tym samym cache co dane dla metody zewnętrznej
- Pobieranie danych przez „wewnętrzny” cache powoduje opróżnienia „zewnętrznego” cache – nie widzimy tych danych w bazie przy odczycie

REQUIRES_NEW – zastosowanie do sterowania izolacją

Możemy niezależnie sterować izolacją takiej nowej transakcji (izolacje transakcji i anomalie zostały opisane wcześniej).

Jeżeli potrzebujemy silnej (a co za tym idzie kosztownej) izolacji, to możemy nakładać ją na krótsze okresy czasu – jedynie metody wewnętrzne. Może być to przydatne np. gdy korzystamy z generatorów kluczy, do których dostęp chce uzyskać wiele wątków równocześnie i chcemy z tego powodu utrzymywać blokady jak najkrócej.

Uwaga na rekurencję metod o tej propagacji!

NESTED

Polityka wymusza jedną transakcję fizyczną i savepoints na poziomie bazy danych. Zatwierdzenie (commit) następuje na końcu metody rozpoczynającej transakcję. Wyjątek metody wewnętrznej wycofuje pod-transakcję do poprzedniego savepoint. Zewnętrzna metoda może kontynuować pomimo wycofania (rollback) wewnętrznej – Listing 4.

<<LISTING lang=java>>

```
@Service
public class OuterService{
  @Autowired
  private MyDAO myDAO;

  @Autowired
  private InnerService innerService;

  @Transactional(propagation=Propagation.REQUIRED)
  public void testRequired(User user) {
    myDAO.insertUser(user);
    try{
      innerService.doNested();
    } catch(RuntimeException e){
      // handle exception
    }
    //wyjątek unchecked wycofa dane metody wewnętrznej, gdyby nie został
przechwycony, to wycofały całość
  }
}
//==================================

@Service
public class InnerService{

  @Transactional(propagation=Propagation.NESTED)
  public void doNested() {
    throw new RuntimeException("Rollback this transaction!");
  }
}
```

<</LISTING>>

Listing 4. Propagacja NESTED

NESTED – zastosowanie do alternatywnych ścieżek

Zagnieżdżanie transakcji jest użyteczne w przypadku długich procesów (np. import dużych sekcji danych), gdzie pewne ścieżki mogą się nie powieść i wówczas chcemy się z tej ścieżki wycofać, aby pójść inną.

SUPPORTS

Metoda o takiej polityce może działać bez kontekstu transakcji. Ale jeżeli kontekst istnieje, to jest do niego dołączana. Skutkuje to widocznością kontekstu persystencji (Session/Entitymanager) stworzonego wcześniej.

SUPPORTS – zastosowanie do odczytu

Jest to idealna propagacja do metod odczytujących dane. Taka metoda może być wołana bez transakcji, lub wewnątrz już istniejącej. W drugim przypadku metoda odczytująca „widzi" dane zapisane (jeszcze bez commit) przez metodę nadrzędną – są to dane z transaction log. Przy braku transakcji odczyt nastąpi z danych z tabeli.

NOT_SUPPORTED

Na czas wywołania metody o takiej polityce propagacji zawieszana jest istniejąca transakcja. Skutkuje to brakiem widoczności zasobów takich jak EntityManager.

Polityka ta jest odpowiednia w przypadku wywołania procedur bazodanowych, które same zarządzają transakcjami - a łącznie (chaining) nie jest wspierane przez bazę danych.

MANDATORY

Metoda o tej polityce propagacji dołącza się do istniejącego kontekstu transakcji, a jeżeli takowy nie istnieje, to dostajemy wyjątek.

Do czego może służyć takie „wybredne” zachowanie.

Możemy przyjąć ziarniste API i strategię zarządzania transakcjami przez kod kliencki (patrz rozdział „Strategie transakcji”).

Metoda nie wywoła się, jeżeli wcześniej kod kliencki nie rozpoczął transakcji.

NEVER

Wywołując metodę o tej polityce, dostaniemy wyjątek, gdy transakcja już istnieje. Jakie może być praktyczne zastosowanie takiej polityki? Jedyne, jakie przychodzi mi do głowy, to zastosowanie na potrzeby testów automatycznych, które testują naszą konfigurację transakcyjności.

Test sprawdza, czy kontekst transakcji istnieje, po tym, że otrzyma spodziewany wyjątek.

Transakcje tylko do odczytu?

Uzupełnieniem artykułu niech będzie link (ramka „W sieci”), traktujący o specyficznym zachowaniu flagi readOnly, która jest często stosowana w celu optymalizacji działania systemu.

Blokowanie optymistyczne

W systemie pracującym pod dużym obciążeniem zwykle unikamy blokowania pesymistycznego (na poziomie zasobów) i spójność danych opieramy o blokowanie optymistyczne (na poziomie aplikacji). Czyli zakładamy optymistycznie, że nie będzie konfliktu przy współbieżnym dostępie do danych, ale gdyby takowy wystąpił, to aplikacja zwróci wyjątek. Wyjątek powinien wycofać transakcję (zakres jej wycofania zależy od polityki propagacji).

Zainteresowanych tym zagadnieniem odsyłam do ramki „W sieci”, gdzie można znaleźć link do artykułu poświęconego blokowaniu optymistycznemu w JPA 2.

<<W_SIECI>>

- Blokowanie optymistyczne w JPA 2: http://www.bottega.com.pl/pdf/materialy/receptury/orm.pdf
- Strategie i modele transakcji: http://www.ibm.com/developerworks/java/library/j-ts2/index.html
- Specyfika działania flagi read only: http://www.ibm.com/developerworks/java/library/j-ts1/index.html
- Poziomy izolacji: http://en.wikipedia.org/wiki/Isolation_(database_systems)

<</W_SIECI>>

<<O_AUTORZE posx=9;0l posy=b fit=W grow=H>>

Sławomir Sobótka

Programujący architekt oprogramowania specjalizujący się w technologiach Java i efektywnym wykorzystaniu zdobyczy inżynierii oprogramowania. Trener i konsultant w firmie Bottega IT Solutions. Entuzjasta Software Craftsmanship.

Do jego zainteresowań należy szeroko pojęta inżynieria oprogramowania: architektury wysokowydajnych systemów webowych (w szczególności CqRS), modelowanie (w szczególności DDD), wzorce, zwinne procesy wytwórcze. Hobbystycznie interesuje się psychologią i kognitywistyką.

W wolnych chwilach działa w community jako: prezes Stowarzyszenia Software Engineering Professionals Polska (http://ssepp.pl), lider lubelskiego Java User Group, publicysta w prasie branżowej i blogger (http://art-of-software.blogspot.com).

Kontakt z autorem: slawomir.sobotka@bottega.com.pl

<</O_AUTORZE>>