

Operacje kaskadowe w Hibernate

Bolesław Ziobrowski

Artykuł opisuje mechanizm operacji kaskadowych w narzędziu Hibernate, problemy z nim związane oraz możliwe ich rozwiązania.

Operacje kaskadowe w Hibernate pozwalają na automatyczne przenoszenie działań wykonanych na jednym obiekcie na obiekty z nim powiązane. Korzystając z niego można mieć pewność, że np. przy zapisie obiektu A w dowolnym fragmencie aplikacji (korzystającej z danego mapowania) zostanie również utrwalony obiekt B. Podobnie jak w przypadku mechanizmu automatycznego pobierania obiektów, opisywanego w poprzednim artykule, wszystko brzmi ładnie i zachęcająco, ale ma zasadniczo te same wady. Problem tkwi tutaj w tym, że mapowanie jest ustawieniem globalnym, wpływającym potencjalnie na całą aplikację. Jeśli już zdecydujemy się na konkretne ustawienie operacji kaskadowych to Hibernate będzie zawsze starał się to ustawienie zrealizować. O ile jednak próby obejścia mechanizmu pobierania automatycznego bez zmiany samego mapowania wprowadzają nowe problemy, to operacje kaskadowe można w sposób względnie prosty i czytelny wyłączyć w danym fragmencie aplikacji. Oczywiście rozwiązanie te należy stosować tylko wtedy, gdy globalnych ustawień nie można, z różnych przyczyn, zmienić.

Prezentowane przykłady opierają się na modelu składającym się z dwóch klas-encji : User oraz Entry. hipotetycznej aplikacji – bloga. Jest to model wykorzystany w poprzednim artykule, pozbawiony, dla uproszczenia przykładów, klasy Comment.

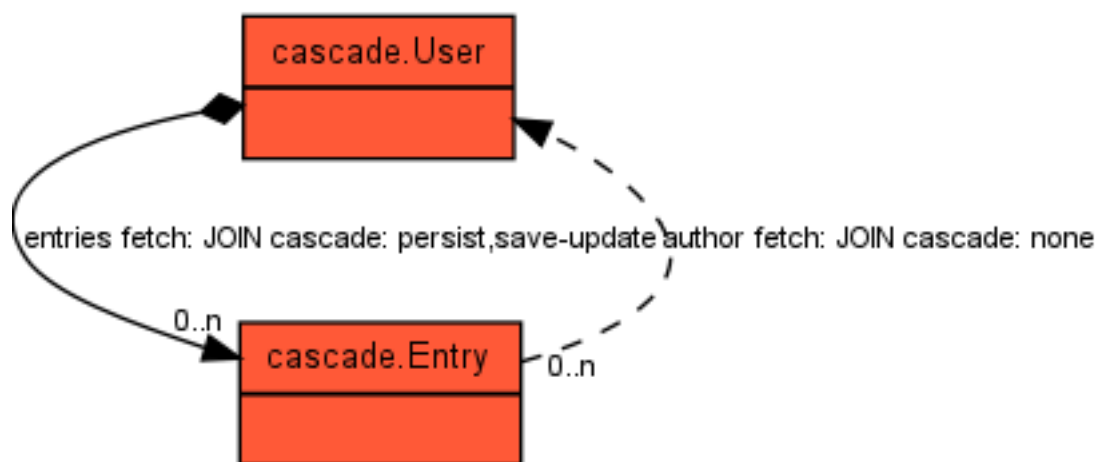


diagram klas utworzony z wykorzystaniem linugine maps

Treść klas wraz z ich adnotacyjnym mapowaniem jest następująca:

```
@Entity
@Table(name="customer")
public class User implements Identifiable
{
    @Id
    @GeneratedValue
    private Integer id;

    @OneToMany(mappedBy="author")
    @Cascade( { CascadeType.PERSIST, CascadeType.SAVE_UPDATE } )
    private Set<Entry> entries = new HashSet<Entry>();
}
```

```

private String name;

User(){ /* required by hiberante */ }

public User( String name )
{
    this.name = name;
}

public Set<Entry> getEntries()
{
    return entries;
}

public Integer getId()
{
    return id;
}

public Entry writeEntry( String content )
{
    Entry newEntry = new Entry( content );
    this.entries.add( newEntry );
    newEntry.setAuthor(this);
    return newEntry;
}

public String getName()
{
    return name;
}

public void setName(String name)
{
    this.name = name;
}
}

```

```

@Entity
public class Entry implements Identifiable
{
    @Id
    @GeneratedValue
    private Integer id;

    @ManyToOne(optional=false)
    private User author;

    private String content;

    @Temporal(TemporalType.TIMESTAMP)
    private Date creationDate;

    Entry() { /* required by hiberante */ }

    public Entry( String content )
    {
        this.content = content;
        creationDate = new Date();
    }
}

```

```

public Integer getId()
{
    return id;
}

public void setAuthor(User author)
{
    this.author = author;
}
}

```

Testy korzystają z bazy danych Postgresql w wersji 8.4, działającej na tym samym komputerze, co uruchamiane testy.

Dzięki ustawieniu "**hibernate.hbm2ddl.auto**" na wartość „**create**” schemat bazy jest odtwarzany w momencie kreacji SessionFactory, zatem do uruchomienia przykładów wystarczy utworzyć użytkownika i bazę "test" - odpowiednie polecenia znajdują się w pliku db.sql.

Rozpatrzmy następujący scenariusz: po utworzeniu użytkownika i kilku jego wpisów, a następnie - zatwierdzeniu transakcji, chcemy zmienić jego imię korzystając z posiadanych obiektów. Zatem próbujemy:

```

User john = new User( "John" );
john.writeEntry( "alamakota" );
john.writeEntry( "tralalala" );
john.writeEntry( "bumcykcyk" );

Session session = HibernateUtility.getSession( persistentClasses );
Transaction transaction = session.beginTransaction();
transaction.begin();

session.persist( john );
session.flush();
session.clear();

john.setName( "Jonathan" );
session.update( john );
session.flush();

transaction.rollback();
session.close();

```

W przykładzie tym utrwalony zostaje nowy użytkownik o imieniu „John” wraz trzema wpisami. Odbywa się to za pomocą propagacji operacji „persist” z użytkownika na wpisy – tak jak jest to określone w mapowaniu. Następnie wymuszam tzw. synchronizację stanu sesji z bazą, czyli w tym wypadku – faktyczne zapisanie danych w bazie. Aby nie zamykać sesji i otwierać nowej – odłączam wszystkie obiekty od istniejącej sesji, co daje w zasadzie ten sam efekt. Sesja w narzędziu Hibernate jest istotnie pamięcią podręczną/buforem, który przechowuje migawki podłączonych obiektów z czasu ostatniej synchronizacji z bazą danych. Porównując bieżące wersje obiektów z ich migawkami, narzędzie jest w stanie określić które z nich zostały zmodyfikowane, i wymagają wykonania odpowiednich operacji bazodanowych. Gorzej jest natomiast, gdy pracujemy na obiektach nieskojarzonych z bieżącą sesją, gdyż wtedy Hibernate nie dysponuje odpowiednimi migawkami, więc ogranicza się do jakiejś domyślnej strategii. Widać to dobrze w bieżącym przykładzie, gdyż poza poleceniami SQL utrwalającymi dane, zmiana imienia użytkownika John powoduje wykonanie :

```

update
customer

```

```

set
    name=?
where
    id=?

update
    Entry
set
    author_id=?,
    content=?,
    creationDate=?
where
    id=?

update
    Entry
set
    author_id=?,
    content=?,
    creationDate=?
where
    id=?

update
    Entry
set
    author_id=?,
    content=?,
    creationDate=?
where
    id=?

```

Jak widać, aktualizacja użytkownika wywołała również zapis wpisów jego autorstwa. Narzut wprowadzony przez nadmiarowe aktualizacje wpisów jest oczywiście proporcjonalny do ich liczby. Strata wydajności byłaby jeszcze większa gdyby klasa Entry propagowała operację aktualizacji na inne powiązane klasy – encje, które zostałyby pobrane w pierwszej sesji. Problem ten najlepiej rozwiązać wyłączając przenoszenie operacji aktualizacji z klasy User na Entry, kłopot w tym, że czasami nie można tego zrobić. Konfiguracja propagacji operacji ma charakter globalny, więc zmieniając ją narażamy się na to, że w jakimś fragmencie aplikacji przestanie działać zapisywanie, aktualizacja, bądź usuwanie obiektów. Dobrze napisane testy powinny wykryć błędy tego rodzaju, aby jednak zadziałały, trzeba je najpierw posiadać (co rzadko ma miejsce). Jak więc usunąć owe niepożądane operacje nie zmieniając mapowania ? Wiemy na pewno, że jawne odłączenie wpisów od sesji nic nie zmieni, gdyż w momencie aktualizacji użytkownika już są odłączone. Spróbujemy zatem opróżnić kolekcję wpisów użytkownika przed zmianą jego imienia w następujący sposób:

```

List<Entry> entries = new ArrayList<Entry>();
entries.addAll( john.getEntries() );

john.getEntries().clear();

```

W dzienniku zdarzeń Hibernate zostaje zarejestrowane tylko :

```

update
    customer
set
    name=?
where
    id=?

```

co oznacza, że wyeliminowaliśmy niechciane operacje. Jeśli chcemy ponownie wykorzystać obiekty użytkownika i jego wpisy, to musimy uważać by nie odtworzyć stanu kolekcji wpisów użytkownika przed końcem sesji.

```
john.getEntries().addAll( entries );
session.flush();
```

Jeśli to zrobimy, to Hibernate radośnie wykona wszystkie aktualizacje, których chcemy uniknąć. Niestety nie zawsze możemy w prosty sposób umieścić kod odtwarzający stan kolekcji po zakończeniu sesji. Dla przykładu - gdy korzystamy z `OpenSessionInViewFilter` w szkieletcie aplikacji Spring, to sesja jest zamykana dopiero po wygenerowaniu widoku. Od problemu tego wolne jest następne rozwiązanie, w którym przyłączamy obiekty w jawny sposób do sesji za pomocą metody `Session.lock()` w następującym fragmencie:

```
for ( Entry entry : john.getEntries() )
{
    session.lock( entry, LockMode.NONE );
}
```

Użycie trybu `LockMode.None` powoduje skojarzenie obiektów z sesją bez wykonywania jakichkolwiek operacji bazodanowych. Teraz gdy aktualizujemy użytkownika, narzędzie przeglądanie całą kolekcję jego wpisów i, jako że posiada migawki z czasu wywołania operacji `lock()`, spróbuje wykryć modyfikacje, ale oczywiście żadnej nie znajdzie. Tym samym wykona tylko polecenie:

```
update
  customer
set
  name=?
where
  id=?
```

Po ustawieniu rejestrowanego poziomu zdarzeń w Hibernate na `TRACE`, widzimy dokładnie jak przebiega proces przyłączania i aktualizacji w połączeniu z propagacją operacji :

#przyłączenie trzech wpisów

```
id unsaved-value: null
reassociating transient instance: [cascade.Entry#2]
id unsaved-value: null
reassociating transient instance: [cascade.Entry#3]
id unsaved-value: null
reassociating transient instance: [cascade.Entry#4]
```

#aktualizacja użytkownika

```
updating detached instance
updating [cascade.User#1]
updating [cascade.User#1]
```

#propagacja aktualizacji użytkownika

```
processing cascade ACTION_SAVE_UPDATE for: cascade.User
cascade ACTION_SAVE_UPDATE for collection: cascade.User.entries
cascading to saveOrUpdate: cascade.Entry
persistent instance of: cascade.Entry
ignoring persistent instance
object already associated with session: [cascade.Entry#2]
cascading to saveOrUpdate: cascade.Entry
persistent instance of: cascade.Entry
ignoring persistent instance
```

```

object already associated with session: [cascade.Entry#3]
cascading to saveOrUpdate: cascade.Entry
persistent instance of: cascade.Entry
ignoring persistent instance
object already associated with session: [cascade.Entry#4]
done cascade ACTION_SAVE_UPDATE for collection: cascade.User.entries
done processing cascade ACTION_SAVE_UPDATE for: cascade.User
#synchronizacja stanu sesji z bazą
flushing session
processing flush-time cascades
processing cascade ACTION_SAVE_UPDATE for: cascade.User
cascade ACTION_SAVE_UPDATE for collection: cascade.User.entries
cascading to saveOrUpdate: cascade.Entry
persistent instance of: cascade.Entry
ignoring persistent instance
object already associated with session: [cascade.Entry#2]
cascading to saveOrUpdate: cascade.Entry
persistent instance of: cascade.Entry
ignoring persistent instance
object already associated with session: [cascade.Entry#3]
cascading to saveOrUpdate: cascade.Entry
persistent instance of: cascade.Entry
ignoring persistent instance
object already associated with session: [cascade.Entry#4]
done cascade ACTION_SAVE_UPDATE for collection: cascade.User.entries
done processing cascade ACTION_SAVE_UPDATE for: cascade.User
dirty checking collections
Flushing entities and processing referenced collections
Updating entity: [cascade.User#1]
Collection found: [cascade.User.entries#1], was: [cascade.User.entries#1]
(initialized)
Processing unreferenced collections
Scheduling collection removes/(re)creates/updates
Flushed: 0 insertions, 1 updates, 0 deletions to 4 objects
Flushed: 0 (re)creations, 0 updates, 0 removals to 1 collections

```

W ten sposób osiągnęliśmy zmierzony cel – Hibernate aktualizuje wyłącznie obiekt użytkownika, a my nie musimy się martwić o przyłączanie wpisów po zamknięciu sesji. Niestety także obecne podejście ma wady. Załóżmy, że wpisy posiadają komentarze (klasa Entry posiada mapowaną kolekcję obiektów trwałych Comment), na które propagują wyłącznie operację aktualizacji. W takiej sytuacji narzędzie przegłębłoby wszystkie wpisy i nie wykryło w nich zmian. Następnie sprawdzone zostałyby wszystkie ich komentarze, z które zostałyby potraktowane tak, jak z niepodłączone do sesji wpisy. Mimo że Hibernate ignoruje skojarzone z sesją wpisy, to nie „powstrzymują” one mechanizmu operacji kaskadowych. Możemy więc albo skonfigurować przenoszenie operacji lock (co można zrobić wyłącznie za pomocą adnotacji `org.hibernate.annotations.Cascade` lub plikach XML) z klasy Entry na Comment (i dalej na wszystkie związki dla których włączona jest propagacja aktualizacji) albo zastąpić obiekty Entry pośrednikami (proxy) w następujący sposób:

```

List<Entry> replacedEntries =
HibernateUtility.proxify(session, john.getEntries() );
john.setName( "Jonathan" );
session.update( john );
session.flush();

HibernateUtility.unproxify(session, replacedEntries,
john.getEntries());
session.flush();

```

W przykładzie tym najpierw zastępuje wpisy pośrednikami, następnie aktualizuję użytkownika i synchronizuje stan sesji z bazą, po czym odtwarzam stan pierwotny stan kolekcji. Aby wykryć ewentualne skutki uboczne (nadmiarowe polecenia SQL), które mogą objawić się w aplikacjach korzystających z ustawienia **org.hibernate.FlushMode.Commit**, po ostatnim kroku ponownie przeprowadzam synchronizację. W dzienniku poleceń widzimy wyłącznie polecenie :

```
update
  customer
set
  name=?
where
  id=?
```

Metody **proxify** i **unproxify** mają następującą treść:

```
public interface Identifiable
{
    public Integer getId();
}

public static <T extends Identifiable>
List<T> proxify( Session session, Collection<T> collection )
{
    List<T> replaced = new ArrayList<T>();
    List<T> proxies = new ArrayList<T>();

    for ( T object : collection )
    {
        if ( Hibernate.initialized(object) &&
            object.getId() != null )
        {
            session.evict(object);
            replaced.add(object);
            proxies.add( (T)session.load( object.getClass(), object.getId() ) );
        }
    }

    collection.removeAll(replaced);
    collection.addAll(proxies);

    return replaced;
}

public static <T extends Identifiable> void unproxify( Session session,
Collection<T> proxiedCollection, Collection<T> proxies )
{
    for ( T entity : proxiedCollection )
    {
        session.lock( entity, LockMode.NONE );
    }

    for ( T proxy : proxies )
    {
        Hibernate.initialize(proxy);
    }
}
```

Metoda **proxify** najpierw odłącza, przy użyciu metody **Session.evict**, od sesji wszystkie obiekty znajdujące się w przekazanej do niej kolekcji, a następnie metodą **Session.load** tworzy dla nich obiekty – pośredników i umieszcza je w kolekcji. Metoda **unproxify** z kolei najpierw podłącza

wszystkie obiekty do sesji (metodą **Session.lock**), a następnie inicjalizuje obiekty – pośredników. Jako, że obiekty już znajdują się w sesji, Hibernate nie wykona przy tym żadnego polecenia bazodanowego.

W rozwiązaniu tym nie musimy już martwić się konfiguracją propagacji operacji przyłączania obiektów Entry do sesji. Z drugiej strony, operacja zamiany obiektu na pośrednika wymaga alokacji nowego obiektu, a inicjalizacja pośrednika – przeszukania zawartości sesji. Dla kolekcji o dużej liczbie elementów może się okazać, że działania wprowadzają pewien narzut. Warto wtedy spróbować zastosować podejście opisane w poprzednim przykładzie.

Ostatni przykład pokazał, że można w sposób czytelny i dosyć prosty zmusić mechanizm operacji kaskadowych do działania w określony sposób. Pamiętać musimy o tym, że jest to wyłącznie obejście najprawdopodobniej złego ustawienia w mapowaniu, które może objawić się we fragmentach aplikacji, które będziemy tworzyć w przyszłości. Moim zdaniem o wiele lepiej jest zastosować strategię jak najmniejszego korzystania z propagacji operacji. Tak jak w przypadku mechanizmu automatycznego pobierania obiektów należy się zastanowić, czy w każdym przypadku, gdy wykonam operację X (persist, update, merge itd.) będę chciał by ta operacja była również wykonana na powiązonym z nim obiekcie, bądź kolekcji Y. Zamiast korzystać w tej chwili z ustawienia CascadeType.**SAVE_UPDATE**, łatwo napisać metodę

```
public void updateUserWithEntries( Session session, User user )
{
    session.update( user );
    if ( Hibernate.initialized( user.getEntries() ) )
    {
        for ( Entry entry : user.getEntries() )
        {
            session.update( entry );
        }
    }
}
```

, która pozwala nam precyzyjnie kontrolować zestaw obiektów, na którym chcemy przeprowadzić daną operację teraz (czyli User i Entry) oraz w przyszłości (wyłącznie obiekt User). Gdyby po pewnym czasie okazało się, że zawsze modyfikujemy zarówno obiekty User jak i Entry to zawsze możemy ustawić przenoszenie aktualizacji w mapowaniu, po czym odnaleźć miejsca wywołania powyższej metody i je usunąć. Nieporównywalnie trudniej jest takie ustawienie wyłączyć, gdyż musimy wtedy wyszukać wszystkie miejsca w kodzie, w których modyfikujemy obiekty klasy User, oraz wszystkich klas, które z kolei przenoszą aktualizację na klasę User.

Podsumowując, mechanizm propagacji zmian w Hibernate (tak jak automatyczne pobieranie obiektów) pozwala na „*szybkie rozwiązanie*” bieżących problemów np. z zapisem obiektów do bazy bez potrzeby napisania dodatkowych linii kodu. Co charakterystyczne dla tego narzędzia, bardzo łatwo przy tym zrobić sobie krzywdę (i to potężną*) i później być zmuszonym walczyć z nieprzewidywanymi skutkami tych „*szybkich rozwiązań*”. Dlatego też należy każdą taką decyzję **poważnie** przemyśleć (kilka razy) i tylko w rzadkich, naprawdę uzasadnionych przypadkach, decydować się na stosowanie operacji kaskadowych.

* - jest to, moim zdaniem, prawdziwe uzasadnienie użycia przymiotnika *potężny* w opisie narzędzia zawartym na stronie hibernate.org, mianowicie „Hibernate is a powerful, high performance object/relational persistence and query service”