JAVA/J2EE au plus haut niveau

ecoalis

SPRING – Gestion des transactions & Support d'Hibernate

par Samir MILOUDI

Programme

Partie 1 : Rappels sur la gestion des transactions

Partie 2 : Gestion des transactions avec Spring

Partie 3 : Support pour les DAO et Hibernate

Partie 1

Rappels sur la gestion des transactions

Sommaire

- Concept de transaction
- Mode de propagation
- Niveau d'isolation

Concept de transaction

- Définition
- Propriétés d'une transaction

Définition

Une transaction consiste à effectuer une opération cohérente composée de plusieurs tâches unitaires.

Si l'opération globale échoue (une ou plusieurs tâches unitaires n'ont pas été correctement effectuées), l'ensemble des données modifiées reviennent à leur état initial (*rollback*).

Si l’opération globale est un succès, les données modifiées sont alors validées (*commit*).

Propriétés d'une transaction (1/3)

Toute transaction doit respecter les quatre contraintes suivantes, dites *ACID* :

1) **Atomicité** : la transaction doit s'effectuer complètement ou pas du tout.

2) **Cohérence** : entre chaque étape d'une procédure complexe, le système doit rester cohérent. Des règles fonctionnelles doivent piloter les changements d'état des données concernées (dans les deux sens).

Propriétés d'une transaction (2/3)

3) **Isolation** : définit la visibilité des états internes de la transaction par le reste de l'application au cours de sa réalisation.

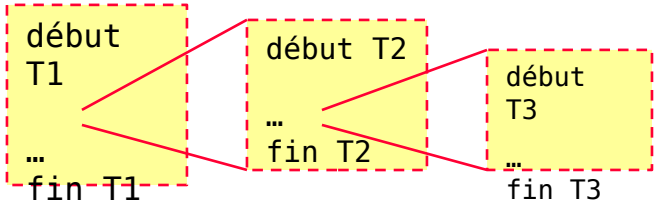
4) **Durabilité** : lorsque la transaction est achevée, le système est dans un état *stable durable*, soit à l'issue d'une modification transactionnelle réussie, soit à l'issue d'un échec qui se solde par le retour à l'état stable antérieur.

Propriétés d'une transaction (3/3)

Une transaction peut être *suspendue* puis *reprise.*

Le déclenchement d'évènements synchronisés à ces états (suspension et reprise) est envisageable.

Enfin, plusieurs transactions peuvent être imbriquées. Les mécanismes « d'ACIDité » sont respectés.



Mode de propagation

- Définition
- Effet sur une transaction
- modes de propagation disponibles

Définition

Le « mode de propagation » correspond au comportement transactionnel souhaité lors de l’enchaînement des méthodes d'un composant ou lors de l’appel d’autres composants transactionnels.

Effet sur une transaction

Lorsqu'un composant transactionnel en appelle un autre, le mode de propagation implique pour la transaction en cours :

- qu'elle soit suspendue
- qu'elle soit réutilisée
- qu'une erreur soit levée
- qu'une nouvelle transaction démarre

Modes de propagation disponibles

mode de propagation	description
PROPAGATION_REQUIRED	Utilise la transaction en cours. En crée une s'il n'en existe pas. Mode par défaut.
PROPAGATION_SUPPORTS	Utilise la transaction en cours. Ne crée pas de transaction s'il n'en existe pas.
PROPAGATION_MANDATORY	Requiert l'existence d'une transaction en cours. Jette une Exception sinon.
PROPAGATION_REQUIRES_NEW	Crée une transaction juste pour la méthode. Suspend l'éventuelle transaction en cours.
PROPAGATION_NOT_SUPPORTED	Suspend l'éventuelle transaction en cours. Les opérations de la méthode ne s'exécutent pas dans le cadre d'une transaction
PROPAGATION_NEVER	Jette une Exception si la méthode est appelée dans le cadre d'une transaction.
PROPAGATION_NESTED	Crée une transaction imbriquée.

Niveau d'isolation

- Définition
- Problématiques
- Niveaux d'isolation disponibles

Définition

Du fait de leur durée, les transactions nécessitent des mécanismes de gestion de la concurrence d'accès et des verrous, qui sont fournis par les sources de données et les technologies utilisées.

La stratégie d'isolation d'une transaction par rapport aux autres est appelé son « niveau d'isolation ».

Problématiques (1/2)

Trois problèmes peuvent survenir lors de l'exécution de transactions concurrentes :

1) *lecture sale* (Dirty Read) : une transaction lit des données écrites par une transaction concurrente non encore validée.

2) *lecture fantôme* (Phantom Read) : Une transaction exécute deux opérations de lecture qui introduisent une différence dans les résultats.

Problématiques (2/2)

3) *lecture non reproductible* (Non-Repeatable Read) : une transaction relit des données qu'elle a lu précédemment et trouve que les données ont été modifiées par une autre transaction (validée depuis la lecture initiale).

Niveaux d'isolation disponibles

Niveau d'isolation	lecture sale	lecture non reproductible	lecture fantôme
DEFAULT	utilise le niveau d'isolation par défaut du système sous-jacent (valeur par défaut)		
READ_UNCOMMITTED	possible	possible	possible
READ_COMMITTED	impossible	possible	possible
REPEATABLE_READ	impossible	impossible	possible
SERIALIZABLE	impossible	impossible	impossible

Le choix du niveau d'isolation des données sera un compromis entre *une dégradation des performances* et la *qualité et la cohérence* des données.

Partie 2

Gestion des transactions avec Spring

- API proposée par Spring
- Transactions programmatiques
- Transactions déclaratives

API proposée par Spring

Pour la gestion des transactions, Spring propose une API composée des interfaces suivantes :

- `PlatformTransactionManager` qui permet de démarrer une transaction, de la valider ou de l'annuler
- `TransactionDefinition` qui définit les niveaux d'isolation et de propagation
- `TransactionStatus`, qui conserve l'état de la transaction en cours

Spring fournit également les implémentations du gestionnaire de transactions pour :

- JDBC
- Hibernate
- Ibatis
- JPA
- JMS
- JDO
- JCA
- XA

Transactions programmatiques

- Configuration
- Utilisation de l'API générique
- Utilisation du *template* transactionnel

Configuration

Le gestionnaire de transaction (1) qui est basé par exemple sur JPA doit simplement être injecté dans le composant transactionnel (2).

```
<bean id="transactionManag 1 "
class="org.springframework.orm.jpa.JpaTransactionManager">
   <property name="entityManagerFactory"
ref="entityManagerFactory"/>
</bean>

                          2
<bean id="userManager"
class="com.ecoalis.service.impl.UserManagerImpl">
   <property name="transactionManager" ref="transactionManager"/>
</bean>
```

Ensuite, deux solutions sont proposées par Spring pour mettre en œuvre les transactions :

1)Utilisation de l'API générique

2)Utilisation du *template* transactionnel

Utilisation de l'API générique

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);
def.setIsolationLevel(TransactionDefinition.ISOLATION_SERIALIZABLE);

TransactionStatus status = transactionManager.getTransaction(def);

try {
     [...]
} catch (BusinessException ex) {
     transactionManager.rollback(status);
     throw ex;
}
transactionManager.commit(status);
```

1) Définition du niveau d'isolation et du mode de propagation
2) Acquisition et démarrage d'une nouvelle transaction
3) Traitements métier et accès aux données
4) Annulation de la transaction en cas d'erreur
5) Validation de la transaction

Utilisation du template transactionnel

```
TransactionTemplate template = new TransactionTemplat(1);

template.setTransactionManager(transactionManag(2);

Object result = template.execute(new TransactionCallback() {
    public Object doInTransaction(TransactionStatus status) {
        // Traitements métier et accès aux données
        [...]          (3)
        return [...];
    }
});
```

La validation ou annulation de la transaction est effectuée automatiquement par le template.

1) Acquisition du template transactionnel
2) Injection du gestionnaire de transaction, qui a lui-même été injecté par Spring dans le composant
3) Exécution via le template d'une implémentation personnelle de l'interface TransactionCallback

Transactions déclaratives

- Avantages
- Configuration XML
- Exemple en XML
- Exemple avec des annotations
- Configuration pour les annotations

Avantages

L'approche déclarative n'est pas intrusive pour le code métier.

Le code technique relatif à la gestion des transactions est ajouté par POA.

Configuration XML

```
<beans ...
    xmlns:tx="http://www.springframework.org/schema/tx
    xmlns:aop="http://www.springframework.org/schema/a
    xsi:schemaLocation=
         "...
         http://www.springframework.org/schema/aop
         http://www.springframework.org/schema/aop/spring-aop.xsd
         http://www.springframework.org/schema/tx
         http://www.springframework.org/schema/tx/spring-tx.xsd">

    <bean id="transactionManager" class="[...]">
         [...]
    </bean>

<beans>
```

La configuration nécessite l'ajout, dans le fichier de configuration de Spring, des schémas XML « tx » et « AOP »

le schéma XML « tx » permet grâce aux balises advice, attributes et method, de définir les propriétés transactionnelles des méthodes des composants.

Le schéma « aop » permet d'appliquer les coupes aspect aux composants.

Exemple en XML

```
<beans ...
    >
  <tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
      <tx:method name="create*" propagation="REQUIRED" isolation="SERIALIZABLE"/>
      <tx:method name="update*" ... />
1     <tx:method name="*" read-only="true"/>
    </tx:attributes>
  </tx:advice>

  <aop:config>
    <aop:advisor pointcut="execution(* *..UserManagerImpl.*(..))" advice-ref="txAdvice"/>
  </aop:config>
2
<beans>
```

1) Déclaration d'un comportement transactionnel pour les méthodes indiquées
2) Application aux composants du comportement défini ci-dessus

Aucune modification des données n'est autorisée pour les autres méthodes (qui ne commencent pas par create* ou update*)

Exemple avec des annotations

Les annotation permettent également de définir les comportements transactionnels des composants. Cette déclaration est effectuée dans *l'interface* du composant.

```
@Transactional(readOnly=true)  1
public interface UserManager {

    @Transactional(readOnly = false,
                   propagation = Propagation.REQUIRED,  2
                   isolation = Isolation.SERIALIZABLE)
    void createUser(User user);

    @Transactional(readOnly = false, propagation = Propagation.REQUIRED)
    void updateUser(String userId);

    [...]
}
```

1) par défaut toutes les méthodes ne peuvent faire de modifications sur les données
2) la méthode de création d'utilisateur peut effectuer des modifications de données (surdéfinition de l'attribut `readOnly` de la classe) et les niveaux de propagation et d'isolation sont spécifiés.

Configuration pour les annotations

La configuration des transactions fondée sur les annotations s'active simplement dans le fichier de configuration de Spring :

```
<tx:annotation-driven transaction-manager="transactionManager"/>
```

Partie 3

Support pour les DAO et Hibernate

Support pour les DAO et Hibernate
Sommaire

- Rappel sur le pattern DAO
- Rappel sur l'ORM
- Apport de Spring
- Datasource
- DAO avec Spring JDBC
- DAO avec Spring / Hibernate

Rappel sur le pattern DAO

- Définition

Définition

Le pattern DAO (*Data Access Object*) propose toutes les méthodes pour créer, récupérer, mettre à jour et effacer des données (CRUD pour *create*, *retreive*, *update delete*).

L'utilisation de ce pattern permet de s'abstraire de la façon dont les données sont stockées au niveau des objets métier.

Rappel sur l'ORM

- Définition

Définition

Le mapping objet / relationnel, ou ORM (*Object Relational Mapping*), désigne l'ensemble des technologies permettant de faire correspondre un modèle objet et une base de données relationnelle.

Les frameworks Hibernate, Ibatis et JPA en sont des implémentations.

Il permettent de modéliser la **couche domaine** en gérant l'héritage, le polymorphisme, et les relations complexes entre les objets.

Apport de Spring

- Classes de l'API
- Rôle du *template*
- Rôle de la classe de support

Classes de l'API

Spring propose, pour chaque framework ORM et pour JDBC, des *templates* et des classes de support pour les DAO :

Technologie	template	support DAO
JDBC	`JdbcTemplate`	`JdbcDaoSupport`
Hibernate	`HibernateTemplate`	`HibernateDaoSupport`
TopLink	`TopLinkTemplate`	`TopLinkDaoSupport`
…	…	…

Rôle du *template*

Les *templates* libèrent le développeur :

- de la gestion des ouvertures/fermetures de connexions
- de la gestion des transactions (démarrage, `commit`, `rollback`)
- de la gestion des exceptions

Rôle de la classe de support

Les classes de support des DAO :

- contiennent le *template* correspondant
- permettent, selon la technologie utilisée, de travailler directement avec les *Sessions* (Hibernate et TopLink), Connexions (Cci), ou *EntityManager* (JPA), ...

Datasource

- Définition
- Implémentations Spring
- Exemples de configuration

Définition

Une *DataSource* est une abstraction pour se connecter à une base de données.

Implémentations Spring

Spring en propose quelques implémentations :

- `DriverManagerDataSource` : ouvre une nouvelle connexion à la base de données chaque fois qu'une connexion est demandée.

- `SingleConnectionDataSource` : ouvre une connexion à sa création et l'utilise chaque fois qu'une connexion est demandée.

Généralement les connexions sont dans un pool. On préfèrera utiliser *Apache Commons DBCP* ou encore *c3p0* comme implémentation.

Exemples de configuration

Exemple de configuration d'un pool de connexions avec DBCP dans un contexte Spring :

```
<bean id="mysqlDataSource" class="org.apache.commons.dbcp.BasicDataSource"
  destroy-method="close">
    <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
    <property name="url" value="jdbc:mysql://localhost:3306/tudu"/>
    <property name="username" value="tudu"/>
    <property name="password" value="mdp4tudu"/>
    <property name="maxActive" value="50"/>
    <property name="maxIdle" value="30"/>
</bean>
```

on précise quel `Driver` utiliser

Exemples de configuration

Exemple de DataSource définie par JNDI :

```
<jee:jndi-lookup id="dataSource" jndi-name="jdbc/EcoalisDataSource"/>
```

DAO avec Spring / JDBC

- Approche JdbcTemplate
- Approche JdbcDaoSupport

Approche JdbcTemplate (1/3)

La configuration de JDBC requiert la déclaration d'une *DataSource*.

```
<bean id="dataSource" class="..." 1

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
   <property name="dataSource" ref="dataSource" 2>
</bean>

<bean id="userDAO" class="com.ecoalis.domain.dao.hibernate.UserDAOJdbc">
   <property name="jdbcTemplate" ref="jdbcTemplate"/>
</bean>
                                                3
```

1) La Datasource est déclarée
2) puis injectée dans le Template JDBC
3) le template est alors injecté dans un ou plusieurs DAO

Approche JdbcTemplate (2/3)

Dans le DAO, l'utilisation du *template* est immédiate :

```
jdbcTemplate.update(
   "insert into user (nom, prenom) values (?,?)",
   new Object[]{"Touchunmot","Jean"}
);
```

Approche JdbcTemplate (3/3)

D'autres templates existent pour JDBC :

- `NamedParameterJdbcTemplate` : permet d'utiliser des HashMap pour les paramètres des requêtes au lieu des « ? »
- `SimpleJdbcTemplate` : utilise les génériques pour éviter le transtypage, ainsi que les méthodes à nombre d'arguments variables (Java 5 requis)
- ...

Approche JdbcDaoSupport (1/2)

```
<bean id="dataSource" class="..." 1

<bean id="userDAO" class="com.ecoalis.domain.dao.hibernate.UserDAOSupportJdbc">
   <property name="dataSource" ref="dataSource"/>
</bean>
                                  2
```

1) la datasource est déclarée
2) puis injectée **directement** dans le DAO qui étend l'une des classes `xxJdbcDAOSupport`

Approche JdbcDaoSupport (2/2)

Le DAO doit étendre la classe du support choisie pour JDBC :

```
public class UserDAOSupportJdbc extends SimpleJdbcDaoSupport implements UserDAO {

   public void createUser(User user) {
      getSimpleJdbcTemplate().update(
         "insert into user (nom, prenom) values (?,?)",
         user.getNom(), user.getPrenom()
      );
   }
   [...]
}
```

Le *template* est disponible via un getter,
bien qu'il n'ait pas été configuré explicitement

DAO avec Spring / Hibernate

- Configuration de la *SessionFactory*
- Approche HibernateTemplate
- Approche HibernateDaoSupport

Configuration de la SessionFactory (1/3)

La configuration d'Hibernate requiert la déclaration d'une *DataSource* et d'une *SessionFactory*.

Cette *factory* crée des *Sessions* Hibernate, points d'entrée pour une unité de travail de persistance.

Configuration de la SessionFactory (2/3)

```
<bean id="dataSource" class="..." />

<bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
   <property name="dataSource" ref="dataSource" />
   <property name="mappingResources">
      <list>
         <value>user.hbm.xml</value>
         [...]
      </list>
   </property>
   <property name="hibernateProperties" ref="hibernateProperties"/>
</bean>

<util:properties id="hibernateProperties">
    <prop name="dialect">org.hibernate.dialect.Oracle10gDialect</prop>
    <prop name="show_sql">true</prop>
    <prop name="cache.use_query_cache">true</prop>
</util:properties>
```

1) la `datasource` est injectée dans la `SessionFactory`
2) les mappings objet/relationnel des entités du domaine sont listés
3) des propriétés Hibernate sont précisées, en particulier le « `dialect` » à utiliser

Configuration de la SessionFactory (3/3)

L'utilisation d'annotations pour le mapping ORM, dans les classes du domaine, peut remplacer les fichiers `hbm.xml`.

Il faut dans ce cas utiliser l'implémentation `org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean` pour la *SessionFactory*, et lister les classes annotées (`annotatedClasses`) à la place des `mappingResources`.

Approche HibernateTemplate (1/2)

```
<bean id="sessionFactory" class="..." />

<bean id="hibernateTemplate" class="org.springframework.orm.hibernate3.HibernateTemplate">
   <property name="sessionFactory" ref="sessionFactory"/>
</bean>

<bean id="userDAO" class="com.ecoalis.domain.dao.hibernate.UserDAOHibernate">
   <property name="hibernateTemplate" ref="hibernateTemplate"/>
</bean>

[autres DAO]
```

1) Le template HibernateTemplate est déclaré et la SessionFactory y est injectée
2) Le template est injecté dans le ou les DAO

Approche HibernateTemplate (2/2)

Le DAO se présente alors sous la forme :

```
public class UserDAOHibernate implements UserDAO {

   private HibernateTemplate hibernateTemplate;

   public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {
      this.hibernateTemplate = hibernateTemplate;
   }

   public Property getUser(String key) {
      return (User) hibernateTemplate.get(User.class, key);
   }

     [...]
}
```

Approche HibernateDaoSupport (1/2)

La SessionFactory est directement injectée dans le DAO:

```
<bean id="sessionFactory" class="..." />

<bean id="userDAO" class="com.ecoalis.domain.dao.hibernate.UserDAOHibernate">
   <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

Approche HibernateDaoSupport (2/2)

qui se présente alors sous la forme :

```
public class UserDAOSupportHibernate extends HibernateDaoSupport implements UserDAO {

   public Property getUser(String key) {
     return (User) getHibernateTemplate().get(User, key);
  }

  [...]
}
```

Cette approche plus directe implique néanmoins l'introduction d'une hiérarchie dans les classes DAO.