Applications orientées données (NSY135)

10 – Lecture de données

Auteurs: Raphaël Fournier-S'niehotta et Philippe Rigaux
(philippe.rigaux@cnam.fr,fournier@cnam.fr)

Département d'informatique
Conservatoire National des Arts & Métiers, Paris, France

Introduction

- Nous avons résolus les problèmes d'association entre le modèle objet et le schéma relationnel
- Nous nous intéressons maintenant à l'accès aux données.
- Après l'aperçu des chapitres précédents, nous allons adopter une approche systématique et détailler les concepts
- Le but est de comprendre quels sont les mécanismes à l'oeuvre et de s'interroger sur les performances d'un accès à une base de données via un ORM.

En pratique

- Pour les exercices et exemples de ce chapitre, on crée un nouveau contrôleur, nommé **Requeteur**, associé à l'URL **requeteur**
- Pour les différentes méthodes de lectures étudiées, on crée une classe **Lectures.java** qui tiendra lieu de modèle.

Plan du cours

2 Comment fonctionne Hibernate
- L'architecture
- La session et le cache de premier niveau
- À propos de **hashCode()** et **equals()**

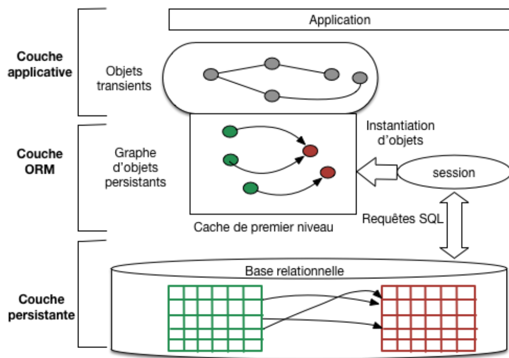
Fonctionnement d'Hibernate

- Nous avons défini le **mapping** ORM, sur lequel Hibernate s'appuie pour accéder aux données, en écriture et en lecture.
- Regardons maintenant comment sont gérés, en interne, ces accès.
- Pour l'instant nous allons nous contenter de considérer les **lectures** de données à partir d'une base existante, sans effectuer aucune mise à jour.
- Cela exclut donc la question des **transactions** qui, comme nous le verrons plus tard, est assez délicate.
- En revanche cela nous permet d'aborder sans trop de complication l'architecture d'Hibernate et le fonctionnement interne du **mapping** et de la matérialisation du graphe d'objet à partir de la base relationnelle.

Fonctionnement d'Hibernate

- On utilise le terme **lecture**, et pas celui de **requête** plus habituel dans un contexte Base de données
- Une application ORM accède aux données (en lecture donc) par différents mécanismes, dont la **navigation** dans le graphe d'objet.
- En revanche, les requêtes effectuées sont souvent déterminées et exécutées par la couche ORM, sans directive explicite du programmeur.
- Rappel : **notre application gère un graphe d'objet, pas une base de données tabulaires**

Structure en couches une application avec ORM



Objets manipulés par l'application

L'application, écrite en Java, manipule des objets que nous pouvons séparer en deux catégories :

- les objets **transients** sont des objets Java standard, instanciés par l'opérateur **new**, dont le cycle de vie est géré par le **garbage collector**;

Objets manipulés par l'application

L'application, écrite en Java, manipule des objets que nous pouvons séparer en deux catégories :

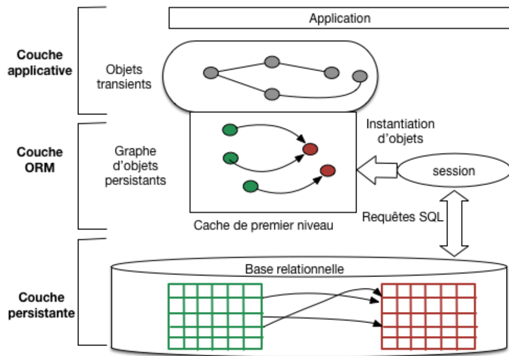
- les objets **persistants** sont également des objets java, instances d'une classe persistante (définie par l'annotation **@Entity**) et **images d'une ligne dans une table de la base relationnelle**.
- L'appartenance à une classe persistante est une condition **nécessaire** pour qu'un objet devienne persistant, mais ce n'est pas une condition **suffisante**.
- Il faut également que l'objet soit placé sous le contrôle du composant chargé de la persistance, soit, pour Hibernate, l'objet **session**.
- Pour le dire autrement, on peut très bien instancier un objet d'une classe persistante et l'utiliser dans un cadre de programmation normal (dans ce cas c'est un objet transient), sans le stocker dans la base.

Objets manipulés par l'application

L'application, écrite en Java, manipule des objets que nous pouvons séparer en deux catégories :

- Les objets **persistants** sont placés dans un espace nommé **le cache de premier niveau** dans Hibernate, que l'on peut simplement voir comme l'emplacement où se trouve matérialisé (partiellement) le graphe des objets utilisés par l'application.
- Cette matérialisation est donc contrôlée et surveillée par un objet **session** que nous avons déjà utilisé pour accéder aux données, mais qu'il est maintenant nécessaire d'examiner en détail car son rôle est essentiel.
- Il existe en fait une troisième catégorie, les objets **détachés**, que nous présenterons dans la chapitre 14, **Applications concurrentes**.

Structure en couches une application avec ORM



Session Hibernate et cache de premier niveau

- La **session Hibernate** définit l'espace de communication entre l'application et la base de données, avec pour responsabilité de **synchroniser** la base de données et le graphe d'objet.
- Pour les lectures, cette synchronisation consiste à transmettre des requêtes **SELECT** via JDBC, pour lire des lignes et instancier des objets à partir des valeurs de ces lignes.

- Des lignes de chaque table sont représentées, sous forme d'objet persistant (rouge ou vert), dans le cache de premier niveau associé à la session.
- L'instantiation de ces objets a été déclenchée par des demandes de lecture de l'application.
- Ces demandes passent **toujours** par la session, soit **explicitement**, comme quand une requête HQL est exécutée, soit **implicitement**, quand par exemple lors d'une navigation dans le graphe d'objet.
- La session est donc un objet absolument essentiel. Pour les lectures en particulier, son rôle est étroitement associé au cache de premier niveau.

Session Hibernate et cache de premier niveau

- La session assure le respect de la propriété suivante : **Dans le contexte d'une session, chaque ligne d'une table est représentée par au plus un objet persistant.**
- Ainsi, une application, quelle que soit la manière dont elle accède à une ligne d'une table (requête, parcours de collection, navigation), obtiendra toujours la référence au même objet.
- Cette propriété d'unicité est donc très importante.
- Imaginons le cas contraire : je fais par exemple plusieurs accès à un même film, et j'obtiens deux objets Java **distincts**, **A** et **B**. Alors :
  - si je fais des modifications **sur A *et* sur B**, quelle est celle qui prend priorité au moment de la sauvegarde dans la base?
  - toute modification sur **A** ne serait pas immédiatement visible sur **B**, d'où des incohérences sources de **bugs** très difficiles à comprendre.

Session Hibernate et cache de premier niveau

- Le cache est donc une **sorte de copie de la base de données au plus près de l'application** (dans la machine virtuelle java) et sous forme d'objets.
- On peut aussi exprimer cela avec le corollaire suivant : **Dans le contexte d'une session, l'identité des objets est équivalente à l'identité base de données.** Deux objets sont identiques (== renvoie **true**) si et seulement si ils ont les mêmes clés primaires.
- Ainsi, le test `a == b` est toujours équivalent à

```
A.getId().equals(B.getId())
```

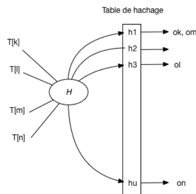
- ceci implique un fonctionnement assez contraint pour la session :
  à chaque accès à la base ramenant un objet, il faut vérifier, **dans le cache de premier niveau**, si la ligne correspondante n'a pas déjà été instanciée, et si oui renvoyer l'objet déjà présent dans le cache.

Session Hibernate et cache de premier niveau

- Pour reprendre l'exemple précédent :
  - l'application fait une lecture (par exemple par une requête) sur la table **Film**, l'objet **A** correspondant à la ligne **l** est instancié, placé dans le cache de premier niveau, et sa référence est transmise à l'application;
  - l'application fait une seconde demande de lecture (par exemple en naviguant dans le graphe); cette lecture a pour paramètre la clé primaire de la ligne **l** :
  - alors la session va d'abord chercher dans le cache de premier niveau si un objet correspondant à **l** existe;
  - si oui sa référence est renvoyée,
  - si non une requête est effectuée;
  - l'application ré-exécute une requête sur la table **Film**, et parcourt les objets avec un itérateur; alors à chaque itération il faut vérifier si la ligne obtenue est déjà instanciée dans le cache de premier niveau.
- Attention on parle ici "du contexte d'une session". Si vous fermez une session **s1** pour en ouvrir une autre **s2**, tout en gardant la référence vers **A**, la propriété n'est plus valable car **A** ne sera pas dans le cache de premier niveau de **s2**.
- À plus forte raison, deux applications distinctes, ayant chacune leur session, ne partageront pas leur cache de premier niveau.

Cache de premier niveau

- Le cache de premier niveau est structuré de manière à répondre très rapidement au test suivant : "Donne moi l'objet dont la clé primaire est **C**".
- La structure la plus efficace est une table de hachage :



- Une fonction **H** prend en entrée une valeur de clé primaire **k, l, m** ou **n** (ainsi que le nom de la table **T**) et produit une **valeur de hachage** comprise entre h_1 et h_u.
- Un répertoire associe à chacune de ces **valeurs de hachage** une **entrée** comprenant un ou plusieurs objets java : o_k, o_l, o_m ou o_n.
- Il peut y avoir des **collisions** : deux clés distinctes mènent à une même entrée, contenant les objets correspondants.

Cache de premier niveau

- En résumé, pour tout accès à une ligne de la base, c'est toujours le même objet qui est retourné à l'application.
- Le cache de premier niveau conserve tous les objets persistants, et c'est dans ce cache que l'application (ou plutôt la session courante) vient piocher. Il **n'est pas** partagé avec une autre application (ou même avec une autre session).
- il existe un cache **de second niveau** qui, lui a des propriétés différentes.

hashCode() et **equals()**

- *dans certains cas*, il est recommandé d'implanter les méthodes **hashCode()** et **equals()**, dans les classes d'objets persistants
- **Quand ?** :
  - Si des instances d'une classe persistante doivent être conservées dans une collection (**Set**, **List** ou **Map**) qui couvre plusieurs sessions Hibernate, alors il est nécessaire de fournir une implantation spécifique de **hashCode()** et **equals()**
- Il vaut mieux éviter cette situation, car elle soulève des problèmes qui n'ont pas de solution entièrement satisfaisante.
- On peut donc décider de toujours éviter de se mettre dans ce mauvais cas.
- Mais on va tâcher de bien comprendre ce qui se passe, et voir la fragilité introduite dans l'application par ce type de pratique.

Le problème

```
// On maintient un liste des utilisateurs
Set<User> utilisateurs;

// Ouverture d'une première session
Session s1 =  sessionFactory.openSession();

// On ajoute l'utilisateur 1 à la liste
User u1 = s1.load (User.class, 1);
utilisateurs.add (u1);

// Fermeture de s1
s1.close();

// Idem, avec une session 2
Session s2 =  sessionFactory.openSession();
User u2 = s2.load (User.class, 1);
utilisateurs.add (u2);
s2.close();
```

- Les deux objets persistants **u1** et **u2**, correspondent à la même ligne,
- **Mais ils ne sont pas identiques** puisqu'ils ont été chargés par deux sessions différentes.
- Ils sont insérés dans le **Set utilisateurs**, et comme leur **hashCode** (qui repose par défaut sur l'identité des objets) est différent, ce **Set** contient donc un doublon, ce qui ouvre la porte à toutes sortes de **bugs**.

- Il faut dans ce cas implanter **hashCode()** et **equals()** non pas sur l'identité objet, mais en tenant compte de la **valeur** des objets pour détecter que ce sont les mêmes.
- le problème ne se pose pas si on reste dans le cadre d'une seule session.

Solution

- Une solution immédiate, mais qui ne marche pas, est d'implanter l'égalité sur l'identifiant en base de données (l'attribut **id**).
- Cela ne marche pas dans le cas des identifiants auto-générés, car la valeur de ces identifiants n'est pas connue au moment où on les instancie.
- Exemple :

```
1         utilisateurs.add(new User("philippe"));
2         utilisateurs.add(new User("raphaël"));
```

- On crée deux instances, pour lesquelles, tant qu'on n'a pas fait de **save()**, l'identifiant est à **null**.
- Si on base la méthode **hashCode()** sur l'**id**, seul le second objet sera placé dans le **Set utilisateur**.
- La seule solution est donc de trouver un ou plusieurs attributs de l'objet persistant qui forment une clé dite "naturelle", à savoir :
qui identifie l'objet de manière unique, dont la valeur est toujours connue et qui ne change jamais.
- Il n'existe à peu près jamais un cas où ces conditions sont pleinement remplies.

Solution (fin)

- Voici un exemple d'implantation de ces deux méthodes, en supposant que le nom de l'utilisateur est une clé naturelle (ce qui est faux, bien sûr).

```
1     @Override
2     public int hashCode() {
3         HashCodeBuilder hcb = new HashCodeBuilder();
4         hcb.append(nom);
5         return hcb.toHashCode();
6     }
7
8     @Override
9     public boolean equals(Object obj) {
10        if (this == obj) {
11            return true;
12        }
13        if (!(obj instanceof User)) {
14            return false;
15        }
16        User autre = (User) obj;
17        EqualsBuilder eb = new EqualsBuilder();
18        eb.append(nom, autre.nom);
19        return eb.isEquals();
20    }
```

- Il est donc souvent déconseillé d'ouvrir et fermer des sessions, à moins d'avoir une excellente raison et de comprendre l'impact
- De même, évitez de stocker dans des structures annexes des objets persistants, la base de données est là pour ça.