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

Plan du cours

2 Les opérations de lecture
- Accès par la clé
- Accès par navigation
- Le langage HQL
- L'API Criteria

3 Résumé: savoir et retenir

Opérations de lecture

- Maintenant que vous comprenez le fonctionnement interne d'Hibernate, au moins pour les grands principes, nous allons regarder rapidement les différentes options de lecture de données.
- Voici la liste des possibilités:
  - par **navigation** dans le graphe des objets, si nécessaire chargé à la volée par Hibernate;
  - par identifiant: méthode basique, rapide, et de fait utilisée implicitement par d'autres méthodes comme la navigation;
  - par le langage de requêtes HQL;
  - par l'API **Criteria**, qui permet de construire par programmation objet une requête à exécuter;
  - enfin, directement par SQL, ce qui n'est pas une méthode portable et devrait donc être évité.

Accès par clé

- Deux méthodes permettent d'obtenir un objet par la valeur de sa clé.
- La première est **get** :

```
return (Film) session.get(Film.class, id);
```

- La seconde est **load** dont l'appel est strictement identique :

```
return (Film) session.load(Film.class, id);
```

- Dans les deux cas, Hibernate examine d'abord le **cache** de la session pour trouver l'objet, et transmet une requête à la base si ce dernier n'est pas dans le cache.
- Les différences entre ces deux méthodes sont assez simples.
  - si **load()** ne trouve par un objet, ni dans le cache, ni dans la base, une exception est levée; **get()** ne lève jamais d'exception;
  - la méthode **load()** renvoie parfois un **proxy** à la place d'une instance réelle.

Notion de proxy

- Un **proxy**, en général, est un intermédiaire entre deux composants d'une application.
- Dans notre cas, un **proxy** est un objet non persistant, qui joue le rôle de ce dernier, et se tient prêt à accéder à la base si **vraiment** des informations complémentaires sont nécessaires.

- Retenez qu'un **proxy** peut décaler dans le temps l'accès réel à la base, et donc la découverte que l'objet n'existe pas en réalité.
- Il semble préférable d'utiliser systématiquement **get()**, quitte à tester un retour avec la valeur **null**.

Accès par navigation

- Considérons l'expression **film.getRealisateur().getNom()** en java, ou plus simplement **film.realisateur.nom** en JSTL.
- Deux objets sont impliqués: le film et le réalisateur, instance de **Artiste**.
- Seul le premier (le film) est **à coup sûr** instancié sous forme d'objet persistant.
- L'artiste peut avoir déjà été instancié, ou pas.

- Hibernate va d'abord tenter de trouver l'objet **Artiste**, en cherchant dans la table de hachage avec la valeur de l'attribut **id_realisateur** du film.
- Si l'objet ne figure pas dans le cache, Hibernate transmet une requête à la base :

```
SELECT * FROM Artiste WHERE id=:film.id_realisateur
```

charge l'objet persistant dans le cache, et le lie au film.

- Cette méthode de matérialisation progressive du graphe en fonction des actions de l'application est appelée "par navigation".
- Elle nous amène à une question très intéressante: dans quelle mesure Hibernate peut-il "anticiper" la matérisalition du graphe pour éviter d'effectuer trop de requêtes SQL?

Comprendre la navigation

- Pour bien comprendre les enjeux du mécanisme de matérialisation associé à la navigation, exécutons la vue suivante qui, en plus d'accéder au réalisateur du film, va chercher tous les films mis en scène par ce réalisateur.

```
1     <%@ page language="java" contentType="text/html; charset=UTF-8"
2       pageEncoding="UTF-8"%>
3
4   <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
5
6   <html>
7   <head>
8   <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
9   <title>Accès à un film par la clé, et navigation</title>
10  </head>
11  <body>
12
13    <h2>Le film no 1</h2>
14    Nous avons bien ramené le film ${film.titre}
15    <h2>Son réalisateur</h2>
16    Et son réalisateur est ${film.realisateur.nom}
17    <h2>Lequel a également réalisé ...</h2>
18
19    <ul>
20  <c:forEach items="${film.realisateur.filmsRealises}" var="film">
21  <li>${film.titre}</li>
22  </c:forEach>
23    </ul>
24  </body>
25  </html>
```

Navigation (suite)

- Associons cette vue à une action qui lit un film par sa clé
- Testons la vue avec Hitchock ou Eastwood,
- Regardons soigneusement les requêtes générées par Hibernate affichées dans la console.
- Que peut-on en conclure sur la stratégie de chargement?

Langage HQL : Motivation

- l'accès à une ligne/objet par son identifiant trouve rapidement ses limites, et il est indispensable de pouvoir également exprimer des requêtes complexes.
- HQL (pour **Hibernate Query Language** bien sûr) est un langage de requêtes **objet** qui sert à interroger le graphe (virtuel au départ) des objets java constituant la vue orientée-objet de la base.
- On interroge un ensemble d'objets java liés par des associations, et pas directement la base relationnelle qui permet de les matérialiser.
- Hibernate se charge d'effectuer les requêtes SQL pour matérialiser la partie du graphe qui satisfait la requête.

Exemple de HQL

- un exemple simple de recherche de films par titre avec HQL.

```
public List<Film> parTitre(String titre)
{
Query q = session.createQuery("from Film f where f.titre= :titre");
q.setString ("titre", titre);
return q.list();
}
```

- la clause **select** est optionnelle en HQL: on interroge des **objets**, et la projection sur certains attributs offre peu d'intérêt.
- Elle a également le grave inconvénient de produire une structure (un ensemble de listes de valeurs) qui n'est pas pré-existante dans l'application, contrairement au modèle objet **mappé** sur la base.
- Sans la clause **select**, on obtient directement une collection des objets du graphe, sans acun travail de décryptage complémentaire.

Exemple de HQL

- un exemple simple de recherche de films par titre avec HQL.

```
public List<Film> parTitre(String titre)
{
Query q = session.createQuery("from Film f where f.titre= :titre");
q.setString ("titre", titre);
return q.list();
}
```

- Comme en JDBC, on peut coder dans la requête des paramètres (ici, le titre) en les préfixant par ":" ("?" est également accepté).
- Hibernate se charge de protéger la syntaxe de la requête, par exemple en ajoutant des barres obliques devant les apostrophes et autres caractères réservés.

Exemple de HQL (suite)

- Il est totalement déconseillé de construire une requête comme une chaîne de caractères, à grand renfort de concaténation pour y introduire des paramètres.

- Insistons sur le fait que HQL est un langage **objet**, même s'il ressemble beaucoup à SQL.

- Il permet de naviguer dans le graphe par exemple avec la clause **where** :

```
from Film f
where f.realisateur.nom='Eastwood'
```

API Criteria

- Hibernate propose un ensemble de classes et de méthodes pour **construire** des requêtes sans avoir à respecter une syntaxe spécifique très différente de java.

```
1  public List<Film> parTitreCriteria(String titre)
2  {
3  Criteria criteria = session.createCriteria(Film.class);
4  criteria.add (Expression.eqOrIsNull("titre", titre));
5  return criteria.list();
6  }
```

- On ajoute ici des **expressions** pour indiquer les restrictions de la recherche.
- Il n'y a aucune possibilité de commettre une erreur syntaxique, et une requête construite avec **Criteria** peut donc être vérifiée à la complilation.
- C'est, avec le respect d'une approche tout-objet, l'argument principal pour cette API au lieu de HQL. Cela dit, on peut aussi estimer qu'une requête HQL est plus concise et plus lisible. (débat ouvert, objet *vs* SQL)
- Dans ce qui suit, nous nous limiterons à HQL (explorez Criteria si vous le souhaitez)

Plan du cours

2 Les opérations de lecture
- Accès par la clé
- Accès par navigation
- Le langage HQL
- L'API Criteria

3 Résumé: savoir et retenir

Résumé

- L'information essentielle à retenir de ce chapitre est le rôle joué par la session Hibernate et le cache des objets maintenu par cette session.
- Il doit être clair pour vous qu'Hibernate consacre beaucoup d'efforts à maintenir dans le cache une image objet cohérente et non redondante de la base.
- Cela impacte l'exécution de toutes les méthodes d'accès dont nous avons donné un bref aperçu.
- Manipulez l'objet **Session** avec précaution.
- Une méthode saine (dans le contexte d'une application Web) est d'ouvrir une session en début d'action, et de la fermer à la fin.