L'héritage avec Hibernate

Table des matières

L'héritage est une des notions fondamentales de la programmation orientée objet. La plupart des bases de données relationnelles ne connaissent pas cette notion. Nous allons voir dans cet article comment Hibernate essaye de faire le grand écart entre le monde objet et le monde relationnel.

L'auteur

Gérald Quintana

L'article

Publié le 5 septembre 2009 - Mis à jour le 5 septembre 2009

Version PDF Version hors-ligne

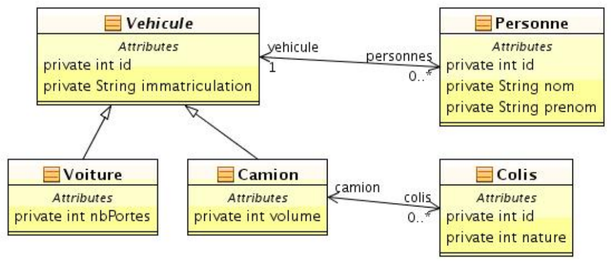
ePub, Azw et Mobi

I. Stratégies de mapping▲

Pour commencer, il faut savoir qu'il existe 3 stratégies pour stocker le contenu d'un modèle objet dans un modèle relationnel. Changer de stratégie en cours de route, ne serait-ce que pour une portion de l'arbre d'héritage, n'est guère possible. Bref, le choix d'une stratégie de mapping est très important puisqu'il s'applique à toute une hiérarchie de classes. Nous allons passer chacune d'elles en revue. En s'appuyant sur un jeu de scénarios, nous mettrons en évidence ce qui les différencie, leurs forces et leurs faiblesses.

I.1. Scénarios de comparaison▲

Les scénarios d'exemple s'appuieront sur un modèle objet simpliste.



Modèle objet

Qui s'écrit en Java de la manière suivante :

Sélectionnez

```
@Entity
public abstract class Vehicule {
  @Id @GeneratedValue Integer id;
  String immatriculation;
  ...
```

```
}

@Entity
public class Voiture extends Vehicule {
  int nbPortes;
  ...
}

@Entity
public class Camion extends Vehicule {
  int volume;
  ...
}
```

Les véhicules peuvent être de 2 types : Voiture ou Camion. Chaque personne peut monter à bord d'un véhicule. Des colis peuvent être chargés dans les Camions.

Dans le premier scénario, on enregistre un véhicule de chaque type en base.

Sélectionnez

```
entityManager.persist(new Voiture("1234AB",5));
entityManager.persist(new Camion("5678XZ", 9));
```

L'objectif est de comparer les écritures en base. Les instructions ci-dessus vont produire des insert en SQL. On pourrait remplacer ces insert par des update ou des delete, le résultat serait sensiblement le même.

Dans le deuxième scénario, on recherche tous les véhicules dont l'immatriculation commence par

Sélectionnez

```
Query query=entityManager.createQuery("select v from Vehicule v where v.immatriculation like
query.setParameter("immatriculation", "1%");
List<Vehicule> vehicules=query.getResultList();
```

La recherche porte sur des critères communs (la clause where), c'est à dire des attributs de la classe mère, mais ce seront, au final, des objets de types différents (tantôt des Voitures, tantôt des Camions) qui seront chargés (dans la clause select).

Dans le troisième scénario, on recherche toutes les Voitures dont le nombre de portes est

Sélectionnez

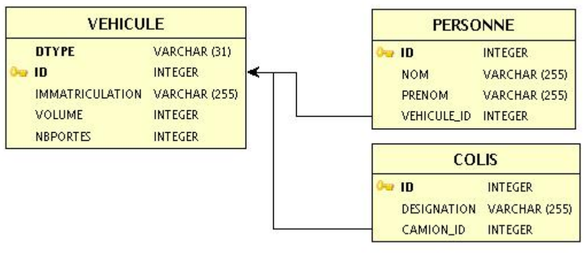
```
Query query=entityManager.createQuery("select v from Voiture v where v.nbPortes = :nbPortes"
query.setParameter("nbPortes", 3);
List<Vehicule> vehicules=query.getResultList();
```

On ne s'intéresse ici qu'aux objets d'un type donné, pourtant le chargement portera aussi sur les attributs de la classe mère. Une recherche par identifiant d'un objet donné produira des effets semblables à la requête précédente :

Sélectionnez

```
Voiture voiture=(Voiture)entityManager.get(Voiture.class, 456);
```

I-2. Une seule table▲



Une seule table

Dans cette stratégie, qui est celle par défaut, le modèle relationnel est fait d'une seule table pour toute la hiérarchie de classes. Les 2 types de véhicules sont stockés dans une même table. Celle-ci est constituée de l'ensemble des colonnes de la hiérarchie de classes, auquel vient s'ajouter une colonne technique appelée discriminant (nommée DTYPE par défaut), qui permet à Hibernate de déterminer le type de véhicule et donc la classe à instancier.

Pour spécifier cette stratégie, on ajoute une annotation @Inheritance(strategy=InheritanceType.SINGLE_TABLE) sur la classe racine :

Sélectionnez

```
@Entity @Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public abstract class Vehicule {
  ...
}
```

Vu qu'il n'y a qu'une seule table, toutes les écritures se font dedans, quelle que soit la nature de l'objet persisté. Au final certaines colonnes sont laissées nulles : nbPortes pour un Camion et volume pour une Voiture.

Sélectionnez

```
INSERT INTO Vehicule (immatriculation, nbPortes, DTYPE, id) VALUES ('1234AB', 5, 'Voiture',
INSERT INTO Vehicule (immatriculation, volume, DTYPE, id) VALUES ('5678XY', 20, 'Camion', 2)
```

En base, on ne peut pas créer :

- De contrainte de non nullité sur la colonne NBPORTES parce que lorsque la classe est Camion, cette colonne n'est pas renseignée. Ou alors il faut passer par une contrainte "check" plus évoluée qui ne s'active que lorsque DTYPE vaut "Voiture".
- De clé étrangère sur une colonne COLONNE_ID qui référencerait un Camion de manière sûre, car Voiture et Camions sont mélangés dans une même table.

La recherche sur la classe mère n'a rien d'extraordinaire :

Sélectionnez

```
SELECT v.id, v.immatriculation, v.volume, v.nbPortes, v.DTYPE
  FROM Vehicule v
  WHERE v.immatriculation LIKE ?
```

Chaque requête sur une classe fille introduit une condition sur la colonne DTYPE :

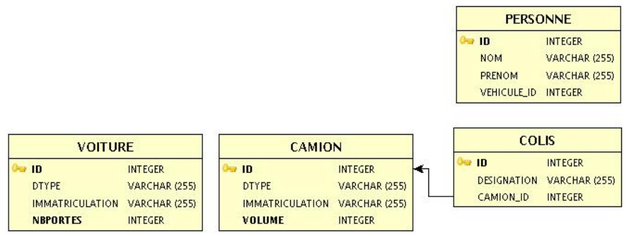
Sélectionnez

```
SELECT x.id, x.immatriculation, x.nbPortes FROM Vehicule x
  WHERE x.DTYPE='Voiture' AND x.nbPortes=?
```

De ce fait, il peut s'avérer judicieux de placer un index sur cette colonne afin d'optimiser ce genre de requête.

L'avantage de cette stratégie est de proposer de bonnes performances quel que soit le scénario. Par contre, dans les hiérarchies de classes importantes, le nombre de colonnes peut rapidement devenir conséquent et l'espace se retrouver gaspillé. Cette solution est très adaptée lorsque les classes diffèrent surtout par leur comportement (méthodes) et peu par leurs données (attributs) ou que la hiérarchie de classes est de petite taille.

I.3. Une table par classe concrète▲



Une table par classe concrète

Avec cette stratégie de mapping, il y a une table pour chaque classe concrète : chaque type de véhicule est stocké dans sa propre table. Chaque table reprend les colonnes de la classe mère, grand-mère, etc... Par contre les classes abstraites comme Vehicule ne sont pas représentées.

Le mapping change très peu de la stratégie précédente, la classe racine est toujours la seule concernée :

Sélectionnez

```
@Entity @Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public abstract class Vehicule {
  ...
}
```

Chaque type de véhicule est inscrit dans sa propre table. Pourtant les identifiants sont communs (les 2 partagent une séquence commune par exemple) :

Sélectionnez

```
INSERT INTO Voiture (immatriculation, nbPortes, id) VALUES ('1234AB', 5, 1);
INSERT INTO Camion (immatriculation, volume, id) VALUES ('5678XY', 20, 2);
```

Ce modèle relationnel ne permet pas :

- De clé étrangère sur la colonne VEHICULE_ID car elle peut pointer sur plusieurs tables
- De contrainte d'unicité sur la colonne IMMATRICULATION parce qu'elle est répartie sur plusieurs tables
- De champ de type IDENTITY (i.e. auto-générés) pour les colonnes ID des tables CAMION et VOITURE parce que les identifiants ne doivent pas se chevaucher

Chaque requête sur la classe mère se traduit par une union sur l'ensemble des tables qui en descendent :

Sélectionnez

```
SELECT v.id, v.immatriculation, v.volume, v.nbPortes, v.clazz_
  FROM (
    SELECT id, immatriculation, NULL AS nbPortes, volume, 1 AS clazz_ FROM Camion
    union ALL
    SELECT id, immatriculation, nbPortes, NULL AS volume, 2 AS clazz_ FROM Voiture
  ) v
  WHERE v.immatriculation LIKE ?
```

Du fait de l'union, ce genre de requête peut s'avérer couteux. De plus, une colonne CLAZZ_ est générée, elle joue le rôle de discriminant et des colonnes dont la valeur est null apparaissent pour palier à leur absence dans les classes sœurs. Contrairement à la recherche précédente, une requête sur une classe fille est triviale :

Sélectionnez

```
SELECT x.id, x.immatriculation, x.nbPortes
  FROM Voiture x
  WHERE x.nbPortes=?
```

Cette stratégie est intéressante lorsque la classe mère ne sert qu'à partager des données entre plusieurs classes et qu'au final ce ne sont que ces classes filles qui sont utilisées. Les principaux inconvénients sont : le coût des requêtes polymorphiques (union) et l'impossibilité d'exprimer des contraintes d'unicité.

A noter que cette stratégie n'est pas imposée par la spécification JPA, son utilisation peut mettre en défaut la portabilité.

I.4. Une table et une jointure par classe▲

Ce modèle relationnel est le plus proche du modèle objet : à chaque classe, qu'elle soit concrète ou abstraite, correspond une table. Autrement dit, les informations concernant une instance de véhicule sont réparties sur plusieurs tables. La seule colonne commune entre les tables est la colonne ID qui permet de faire les jointures table mère et table fille.

Le mapping se configure exactement comme pour la stratégie précédente :

Sélectionnez

```
@Entity @Inheritance(strategy=InheritanceType.JOINED)
public abstract class Vehicule {
  ...
}
```

Contrairement aux précédentes stratégies, on peut créer à peu près tous les types de contraintes relationnelles.

Pour enregistrer un simple objet, il faut faire plusieurs écritures :

Sélectionnez

```
INSERT INTO Vehicule (immatriculation, id) VALUES ('1234AB', 1)
INSERT INTO Voiture (nbPortes, id) VALUES (5, 1)
INSERT INTO Vehicule (immatriculation, id) VALUES ('5678XY', 2)
INSERT INTO Camion (volume, id) VALUES (20, 1)
```

Ce genre de scénario est évidemment couteux. Dès qu'il s'agit de faire une lecture, là aussi il faut parcourir plusieurs tables et effectuer des jointures. Dans le cas d'une requête sur une classe mère, Hibernate doit faire des jointures sur l'ensemble des tables filles :

Sélectionnez

```
SELECT v.id, v.immatriculation, c.volume, x.nbPortes,
    case when c.id IS NOT NULL then 1 when x.id IS NOT NULL then 2 when v.id IS NOT NULL the
  FROM Vehicule v
    LEFT OUTER JOIN Camion c ON c.id=v.id
    LEFT OUTER JOIN Voiture x ON x.id=v.id
  WHERE v.immatriculation LIKE ?
```

La fonction case when ... qui alimente la colonne CLAZZ_ est encore une fois un discriminant, il est cette fois calculé en fonction de la présence (ou pas) d'une relation.

Une recherche sur la classe fille réduit le nombre de jointures mais ne s'en affranchit pas complètement car il faut malgré tout charger les informations stockées dans la table mère :

Sélectionnez

```
SELECT v.id, v.immatriculation, x.nbPortes
  FROM Voiture x
    INNER JOIN Vehicule v ON x.id=v.id
  WHERE x.nbPortes=?
```

Quelle que soit la lecture, une ou plusieurs jointures sont nécessaires.

Que ce soit en lecture ou en écriture, cette stratégie n'est guère performante. Ce qui fait sa force, c'est sa propreté en termes de modélisation relationnelle et la possibilité d'exprimer des contraintes d'intégrités relationnelles claires.

I.5. Bilan▲

	Single table	Table per Class	Joined
Colonnes répétées	Colonnes des classes filles cumulées + discriminant	Colonnes des classes mères cumulées	
Clés étangères	Relation sur la classe mère uniquement	Relation sur la classe fille uniquement	
Unicité		Unicité à cheval sur plusieurs tables	
Non Nullité	Les colonnes d'une classe fille sont laissées nulle chez ses soeurs		
Ecritures (Insert, Update, Delete)			Plusieurs écritures pour une seule instance
Recherche sur classe mère		Une union sur plusieurs tables	Des jointures sur toutes les tables
Recherche sur classe fille			Beaucoup de jointures

II. Astuces d'utilisation▲

II.1. Mapping avancé▲

Dans le cas d'une stratégie SINGLE_TABLE, il est possible de personnaliser le nom de la colonne discriminante et les valeurs qu'elle pourra prendre :

Sélectionnez

```
@Entity @Inheritance(strategy=InheritanceType.SINGLE_TABLE) @DiscriminatorColumn(name="TYPE_
public abstract class Vehicule { ... }

@Entity @DiscriminatorValue("V")
public class Voiture extends Vehicule { ... }

@Entity @DiscriminatorValue("C")
public class Camion extends Vehicule { ... }
```

Ainsi, la colonne discriminante baptisée TYPE_VEHICULE pourra prendre les valeurs V pour Voiture et C pour Camion. Il est aussi possible de mapper cette colonne sur un attribut, à condition de préciser insertable=false et updatable=false :

Sélectionnez

```
@Entity @Inheritance(strategy=InheritanceType.SINGLE_TABLE) @DiscriminatorColumn(name="TYPE_
public abstract class Vehicule {
  @Id @GeneratedValue Integer id;
  String immatriculation;
  @Column(name="TYPE_VEHICULE",insertable=false,updatable=false)
  String discriminator;
  ...
```

En théorie, il n'est pas possible de mixer plusieurs stratégies d'héritage au sein d'une même hiérarchie de classes. En pratique, une astuce permet de commencer par du SINGLE_TABLE puis de passer à quelque chose qui se comporte comme du JOINED pour une classe fille donnée. Pour cela on utilise la notion de table secondaire :

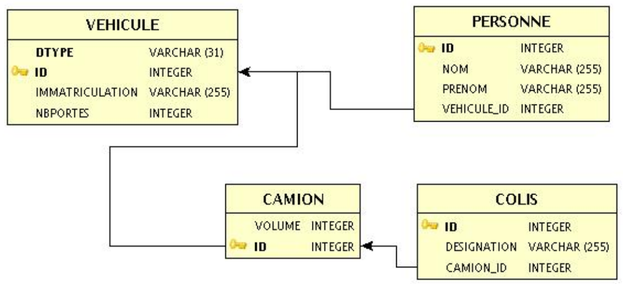
Sélectionnez

```
@Entity @Inheritance(strategy=InheritanceType.SINGLE_TABLE)
public abstract class Vehicule {
  @Id @GeneratedValue Integer id;
  String immatriculation;
  ...
}

@Entity
public class Voiture extends Vehicule {
  int nbPortes;
  ...
}

@Entity
@SecondaryTable(name="CAMION")
public class Camion extends Vehicule {
  @Column(table="CAMION")
  private int volume;
  ...
}
```

On obtient ainsi le modèle relationnel suivant :



Mix entre stratégies

La table VEHICULE stocke les informations de la classe Voiture et la partie "véhicule" des Camions, tandis que la table CAMION accueille la partie spécifique des Camions.

II.2. Recherche avec jointure▲

On recherche les personnes qui sont dans une voiture dont le nombre de portes est... La difficulté de cette requête est que :

1. La relation depuis Personne pointe non pas sur une Voiture mais sur un véhicule
2. Le nombre de portes est un attribut de la classe Voiture et pas Véhicule.

En Java pur, on aurait écrit quelque chose du genre :

Sélectionnez

```
Vehicule vehicule=personne.getVehicule();
if (vehicule instanceof Voiture) {
  Voiture voiture=(Voiture)vehicule;
  if (voiture.getNbPortes()==...) {
    // Sélectionner personne
  }
}
```

L'opérateur instanceof n'est pas permis dans les requêtes Hibernate, mais Hibernate propose un méta-attribut class qui donne la classe d'un objet. Quant au casting de Véhicule en Voiture, il n'est pas nécessaire. Bref, la requête s'écrira en HQL/JPQL de la manière suivante :

Sélectionnez

```
List<Personne> personnes=
  entityManager.createQuery(
    "select p from Personne as p join p.vehicule as v"
    +" where v.class=Voiture and v.nbPortes=:nbPortes");
  .setParameter("nbPortes", 5)
  .getResultList();
```

On procédera de la même manière en utilisant l'API Criteria :

Sélectionnez

```
Session hibernateSession=(Session)entityManager.getDelegate();
List<Personne> personnes=
    hibernateSession.createCriteria(Personne.class)
      .createCriteria("vehicule")
        .add(eq("class",Voiture.class))
        .add(eq("nbPortes",5))
      .list();
```

Un bug (HHH-3828) fait qu'Hibernate se trompe dans la conversion de la Classe en Valeur de discriminant (il ajoute des quotes superflues). De ce fait, lorsque la stratégie d'héritage choisie est SINGLE_TABLE, il faut mettre la valeur du discriminant plutôt que la classe :

Sélectionnez

```
Session hibernateSession=(Session)entityManager.getDelegate();
List<Personne> personnes=
    hibernateSession.createCriteria(Personne.class)
      .createCriteria("vehicule")
        .add(eq("class","Voiture"))
        .add(eq("nbPortes",5))
      .list();
```

II.3. Parcours de relation lazy▲

On a configuré la relation Personne -> Véhicule pour être lazy :

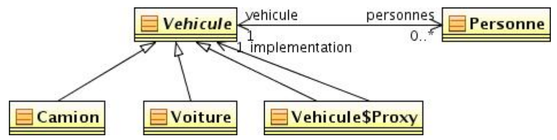
Sélectionnez

```
@ManyToOne(fetch=FetchType.LAZY)
private Vehicule vehicule;
```

Puis on exécute le bout de code suivant :

Sélectionnez

```
Personne personne=entityManager.get(Personne.class, 123);
Vehicule vehicule=personne.getVehicule();
Hibernate.initialize(vehicule);
if (vehicule instanceof Voiture) {
    Voiture voiture=(Voiture)vehicule;
    System.out.println("C'est une Voiture");
} else if (vehicule instanceof Camion) {
    Camion camion=(Camion)vehicule;
    System.out.println("C'est un Camion");
} else {
    System.out.println("C'est autre chose: "+vehicule.getClass().getName());
}
```



A la surprise générale, le résultat affiché n'est ni une Voiture, ni un Camion, mais un Vehicule_$$_javassist_3! Mais que s'est-il passé au juste ? Comme la relation est lazy, au moment où l'objet Personne est chargé, Hibernate remplit l'attribut vehicule avec un objet de type proxy, dont la classe, Vehicule_$$_javassist_3, est générée dynamiquement et dérive de notre classe Vehicule. Le rôle de cette classe, est de déclencher le chargement à la demande de la relation et d'instancier un Vehicule au besoin. Hibernate est confronté à deux problèmes :

- Le premier est qu'au moment du chargement de l'objet personne, il est incapable de savoir de quel type sera son Véhicule vu qu'il n'est pas encore chargé. Il ne peut donc pas anticiper et créer un proxy de Voiture ou de Camion.
- Le second est qu'une fois le proxy de véhicule créé et placé dans l'attribut Véhicule, on ne peut plus transformer sa classe en Voiture ou Camion au moment du chargement.

Pour s'en sortir, il y a plusieurs techniques : la première est de forcer le chargement du véhicule en même temps que la personne, soit en basculant la relation à eager.

Sélectionnez

```
@ManyToOne(fetch=FetchType.EAGER)
private Vehicule vehicule;
```

Soit en utilisant un join fetch au moment de lire l'objet Personne :

Sélectionnez

```
Personne personne=(Personne)
  entityManager.createQuery(
      "select p from Personne p join fetch p.vehicule"
```

```
      +" where p.id=:idPersonne")
    .setParameter("idPersonne", 123)
    .getSingleResult();
```

La seconde technique est de remplacer l'utilisation d'un proxy par l'instrumentation du bytecode. L'idée est d'amener Hibernate à placer le code nécessaire au chargement à la demande, non pas dans une nouvelle classe Vehicule_$$_javassist_3, mais dans notre propre classe Personne.
Pour cela, on ajoute une annotation sur la relation :

Sélectionnez

```
@ManyToOne(fetch=FetchType.LAZY)
@LazyToOne(LazyToOneOption.NO_PROXY)
private Vehicule vehicule;
```

Puis on demande à l'outil livré avec Hibernate de venir modifier notre fichier Personne.class. Ainsi, le chargement du Véhicule ne se fera pas lorsqu'on accède au Véhicule (getImmatriculation() par exemple), mais juste un peu plut tôt : dans le getVehicule() de l'objet Personne.
Enfin, la troisième et dernière façon de faire, est d'écrire manuellement la même chose que l'instrumenteur de code: un getter intelligent pour la relation Personne -> Véhicule :

Sélectionnez

```
public Vehicule getVehicule() {
  Vehicule vehiculeImpl;
  if (vehicule instanceof HibernateProxy) {
    // Véhicule proxifié
    HibernateProxy vehiculeProxy=(HibernateProxy) vehicule;
    vehiculeImpl=(Vehicule) vehiculeProxy.getHibernateLazyInitializer().getImplementation();
  } else {
    // Véhicule véritable
    vehiculeImpl=vehicule;
  }
  return vehiculeImpl;
}
```

En fait, le proxy que génère Hibernate, enveloppe une véritable instance de véhicule. Détaillons l'exemple de code ci-dessus :

- Si l'attribut vehicule de la classe Personne est initialisé avec un proxy, alors je m'assure que celui-ci ait été chargé et j'en extrais la véritable implémentation (getImplementation()),
- Si la relation a été préchargée (join fetch), je peux retourner directement l'instance de vehicule, c'est une vraie.

Ainsi, quelle que soit la façon dont mon objet Personne a été chargé, la méthode getVehicule() ne retournera pas un proxy dont je ne pourrai rien faire.

Vous avez aimé ce tutoriel ? Alors partagez-le en cliquant sur les boutons suivants : Partager

Quels sont les langages de programmation les plus demandés par les employeurs ?

Battle Dev : le concours de programmation en ligne revient pour une nouvelle édition

Visual Studio Code 1.40 apporte un nouvel indicateur dans la barre d'activité

Apprendre à créer en 5 minutes une API REST, avec Java et Vert.x