Documentation de référence d'Hibernate

Version: 3.2final

Table des matières

Préface

Traducteur(s): Vincent Ricard, Sebastien Cesbron, Michael Courcy, Vincent Giguère, Baptiste Mathus, Emmanuel Bernard, Anthony Patricio

Travailler dans les deux univers que sont l'orienté objet et la base de données relationnelle peut être lourd et consommateur en temps dans le monde de l'entreprise d'aujourd'hui. Hibernate est un outil de mapping objet/relationnel pour le monde Java. Le terme mapping objet/relationnel (ORM) décrit la technique consistant à faire le lien entre la représentation objet des données et sa représentation relationnelle basée sur un schéma SQL.

Non seulement, Hibernate s'occupe du transfert des classes Java dans les tables de la base de données (et des types de données Java dans les types de données SQL), mais il permet de requêter les données et propose des moyens de les récupérer. Il peut donc réduire de manière significative le temps de développement qui aurait été autrement perdu dans une manipulation manuelle des données via SQL et JDBC.

Le but d'Hibernate est de libérer le développeur de 95 pourcent des tâches de programmation liées à la persistance des données communes. Hibernate n'est probablement pas la meilleure solution pour les applications centrées sur les données qui n'utilisent que les procédures stockées pour implémenter la logique métier dans la base de données, il est le plus utile dans les modèles métier orientés objets dont la logique métier est implémentée dans la couche Java dite intermédiaire. Cependant, Hibernate vous aidera à supprimer ou à encapsuler le code SQL spécifique à votre base de données et vous aidera sur la tâche commune qu'est la transformation des données d'une représentation tabulaire à une représentation sous forme de graphe d'objets.

Si vous êtes nouveau dans Hibernate et le mapping Objet/Relationnel voire même en Java, suivez ces quelques étapes :

1. Lisez Chapitre 1, *Introduction à Hibernate* pour un didacticiel plus long avec plus d'instructions étape par étape.

2. Lisez Chapitre 2, *Architecture* pour comprendre les environnements dans lesquels Hibernate peut être utilisé.

3. Regardez le répertoire `eg` de la distribution Hibernate, il contient une application simple et autonome. Copiez votre pilote JDBC dans le répertoire `lib/` et éditez `src/hibernate.properties`, en positionnant correctement les valeurs pour votre base de données. A partir d'une invite de commande dans le répertoire de la distribution, tapez `ant eg` (cela utilise Ant), ou sous Windows tapez `build eg`.

4. Faîtes de cette documentation de référence votre principale source d'information. Pensez à lire *Hibernate in Action* (http://www.manning.com/bauer) si vous avez besoin de plus d'aide avec le design d'applications ou si vous préférez un tutoriel pas à pas. Visitez aussi http://caveatemptor.hibernate.org et téléchargez l'application exemple pour Hibernate in Action.

5. Les questions les plus fréquemment posées (FAQs) trouvent leur réponse sur le site web Hibernate.

6. Des démos, exemples et tutoriaux de tierces personnes sont référencés sur le site web Hibernate.

7. La zone communautaire (Community Area) du site web Hibernate est une bonne source d'information sur les design patterns et sur différentes solutions d'intégration d'Hibernate (Tomcat, JBoss, Spring Framework, Struts, EJB, etc).

Si vous avez des questions, utilisez le forum utilisateurs du site web Hibernate. Nous utilisons également l'outil de gestion des incidents JIRA pour tout ce qui est rapports de bogue et demandes d'évolution. Si vous êtes

intéressé par le développement d'Hibernate, joignez-vous à la liste de diffusion de développement.

Le développement commercial, le support de production et les formations à Hibernate sont proposés par JBoss Inc (voir http://www.hibernate.org/SupportTraining/). Hibernate est un projet Open Source professionnel et un composant critique de la suite de produits JBoss Enterprise Middleware System (JEMS).

Chapitre 1. Introduction à Hibernate

1.1. Préface

Ce chapitre est un didacticiel introductif destiné aux nouveaux utilisateurs d'Hibernate. Nous commençons avec une simple application en ligne de commande utilisant une base de données en mémoire, et la développons en étapes faciles à comprendre.

Ce didacticiel est destiné aux nouveaux utilisateurs d'Hibernate mais requiert des connaissances Java et SQL. Il est basé sur un didacticiel de Michael Gloegl, les bibliothèques tierces que nous nommons sont pour les JDK 1.4 et 5.0. Vous pourriez avoir besoin d'autres bibliothèques pour le JDK 1.3.

Le code source de ce tutoriel est inclus dans la distribution dans le répertoire `doc/reference/tutorial/`.

1.2. Partie 1 - Première application Hibernate

D'abord, nous créerons une simple application Hibernate en console. Nous utilisons une base de données en mémoire (HSQL DB), donc nous n'avons pas à installer de serveur de base de données.

Supposons que nous ayons besoin d'une petite application de base de données qui puisse stocker des événements que nous voulons suivre, et des informations à propos des hôtes de ces événements.

La première chose que nous faisons est de configurer notre répertoire de développement et de mettre toutes les bibliothèques dont nous avons besoin dedans. Téléchargez la distribution Hibernate à partir du site web d'Hibernate. Extrayez le paquet et placez toutes les bibliothèques requises trouvées dans `/lib` dans le répertoire `/lib` de votre nouveau répertoire de travail. Il devrait ressembler à ça :

```
.
+lib
  antlr.jar
  cglib-full.jar
  asm.jar
  asm-attrs.jars
  commons-collections.jar
  commons-logging.jar
  ehcache.jar
  hibernate3.jar
  jta.jar
  dom4j.jar
  log4j.jar
```

Ceci est l'ensemble minimum de bibliothèques requises (notez que nous avons aussi copié hibernate3.jar, l'archive principale) pour Hibernate. Lisez le fichier `README.txt` dans le répertoire `lib/` de la distribution Hibernate pour plus d'informations à propos des biliothèques tierces requises et optionnelles. (En fait, log4j n'est pas requis mais préféré par beaucoup de développeurs.)

Ensuite, nous créons une classe qui réprésente l'événement que nous voulons stocker dans notre base de données.

1.2.1. La première classe

Notre première classe persistante est une simple classe JavaBean avec quelques propriétés :

```
package events;

import java.util.Date;

public class Event {
    private Long id;

    private String title;
    private Date date;

    public Event() {}

    public Long getId() {
        return id;
    }

    private void setId(Long id) {
        this.id = id;
    }

    public Date getDate() {
        return date;
    }

    public void setDate(Date date) {
        this.date = date;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }
}
```

Vous pouvez voir que cette classe utilise les conventions de nommage standard JavaBean pour les méthodes getter/setter des propriétés, ainsi qu'une visibilité privée pour les champs. Ceci est la conception recommandée - mais pas obligatoire. Hibernate peut aussi accéder aux champs directement, le bénéfice des méthodes d'accès est la robustesse pour la refonte de code. Le constructeur sans argument est requis pour instancier un objet de cette classe via reflexion.

La propriété `id` contient la valeur d'un identifiant unique pour un événement particulier. Toutes les classes d'entités persistantes (ainsi que les classes dépendantes de moindre importance) auront besoin d'une telle propriété identifiante si nous voulons utiliser l'ensemble complet des fonctionnalités d'Hibernate. En fait, la plupart des applications (surtout les applications web) ont besoin de distinguer des objets par des identifiants, donc vous devriez considérer ça comme une fonctionnalité plutôt que comme une limitation. Cependant, nous ne manipulons généralement pas l'identité d'un objet, dorénavant la méthode setter devrait être privée. Seul Hibernate assignera les identifiants lorsqu'un objet est sauvegardé. Vous pouvez voir qu'Hibernate peut accéder aux méthodes publiques, privées et protégées, ainsi qu'aux champs (publics, privés, protégés) directement. Le choix vous est laissé, et vous pouvez l'ajuster à la conception de votre application.

Le constructeur sans argument est requis pour toutes les classes persistantes ; Hibernate doit créer des objets pour vous en utilisant la réflexion Java. Le constructeur peut être privé, cependant, la visibilité du paquet est requise pour la génération de proxy à l'exécution et une récupération des données efficaces sans instrumentation du bytecode.

Placez ce fichier source Java dans un répertoire appelé `src` dans le dossier de développement. Ce répertoire devrait maintenant ressembler à ça :

```
.
+lib
```

```
  <Hibernate et bibliothèques tierces>
+src
  +events
    Event.java
```

Dans la prochaine étape, nous informons Hibernate de cette classe persistante.

1.2.2. Le fichier de mapping

Hibernate a besoin de savoir comment charger et stocker des objets d'une classe persistante. C'est là qu'intervient le fichier de mapping Hibernate. Le fichier de mapping indique à Hibernate à quelle table dans la base de données il doit accéder, et quelles colonnes de cette table il devra utiliser.

La structure basique de ce fichier de mapping ressemble à ça :

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
[...]
</hibernate-mapping>
```

Notez que la DTD Hibernate est très sophistiquée. Vous pouvez l'utiliser pour l'auto-complétement des éléments et des attributs de mapping XML dans votre éditeur ou votre IDE. Vous devriez aussi ouvrir le fichier DTD dans votre éditeur de texte - c'est le moyen le plus facile d'obtenir une vue d'ensemble de tous les éléments et attributs, et de voir les valeurs par défaut, ainsi que quelques commentaires. Notez qu'Hibernate ne chargera pas le fichier DTD à partir du web, mais regardera d'abord dans le classpath de l'application. Le fichier DTD est inclus dans `hibernate3.jar` ainsi que dans le répertoire `src` de la distribution Hibernate.

Nous omettrons la déclaration de la DTD dans les exemples futurs pour raccourcir le code. Bien sûr il n'est pas optionnel.

Entre les deux balises `hibernate-mapping`, incluez un élément `class`. Toutes les classes d'entités persistantes (encore une fois, il pourrait y avoir des classes dépendantes plus tard, qui ne sont pas des entités mère) ont besoin d'un mapping vers une table de la base de données SQL :

```
<hibernate-mapping>

    <class name="Event" table="EVENTS">

    </class>

</hibernate-mapping>
```

Plus loin, nous disons à Hibernate comment persister et charger un objet de la classe `Event` dans la table `EVENTS`, chaque instance est représentée par une ligne dans cette table. Maintenant nous continuons avec le mapping de la propriété de l'identifiant unique vers la clef primaire de la table. De plus, comme nous ne voulons pas nous occuper de la gestion de cet identifiant, nous utilisons une stratégie de génération d'identifiant d'Hibernate pour la colonne de la clef primaire subrogée :

```
<hibernate-mapping>

    <class name="Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="increment"/>
        </id>
    </class>
```

```
</hibernate-mapping>
```

L'élément `id` est la déclaration de la propriété de l'identifiant, `name="id"` déclare le nom de la propriété Java - Hibernate utilisera les méthodes getter et setter pour accéder à la propriété. L'attribut `column` indique à Hibernate quelle colonne de la table `EVENTS` nous utilisons pour cette clef primaire. L'élément `generator` imbriqué spécifie la stratégie de génération de l'identifiant, dans ce cas nous avons utilisé `increment`, laquelle est une méthode très simple utile surtout pour les tests (et didacticiels). Hibernate supporte aussi les identifiants générés par les bases de données, globalement uniques, ainsi que les identifiants assignés par l'application (ou n'importe quelle stratégie que vous avez écrit en extension).

Finalement nous incluons des déclarations pour les propriétés persistantes de la classe dans le fichier de mapping. Par défaut, aucune propriété de la classe n'est considérée comme persistante :

```
<hibernate-mapping>

    <class name="Event" table="EVENTS">
        <id name="id" column="EVENT_ID">
            <generator class="increment"/>
        </id>
        <property name="date" type="timestamp" column="EVENT_DATE"/>
        <property name="title"/>
    </class>

</hibernate-mapping>
```

Comme avec l'élément `id`, l'attribut `name` de l'élément `property` indique à Hibernate quels getters/setters utiliser.

Pourquoi le mapping de la propriété `date` inclut l'attribut `column`, mais pas `title` ? Sans l'attribut `column` Hibernate utilise par défaut le nom de la propriété comme nom de colonne. Ca fonctionne bien pour `title`. Cependant, `date` est un mot clef réservé dans la plupart des bases de données, donc nous utilisons un nom différent pour le mapping.

La prochaine chose intéressante est que le mapping de `title` manque aussi d'un attribut `type`. Les types que nous déclarons et utilisons dans les fichiers de mapping ne sont pas, comme vous pourriez vous y attendre, des types de données Java. Ce ne sont pas, non plus, des types de base de données SQL. Ces types sont donc appelés des *types de mapping Hibernate*, des convertisseurs qui peuvent traduire des types Java en types SQL et vice versa. De plus, Hibernate tentera de déterminer la bonne conversion et le type de mapping lui-même si l'attribut `type` n'est pas présent dans le mapping. Dans certains cas, cette détection automatique (utilisant la réflexion sur la classe Java) pourrait ne pas donner la valeur attendue ou dont vous avez besoin. C'est le cas avec la propriété `date`. Hibernate ne peut pas savoir si la propriété "mappera" une colonne SQL de type `date`, `timestamp` ou `time`. Nous déclarons que nous voulons conserver des informations avec une date complète et l'heure en mappant la propriété avec un `timestamp`.

Ce fichier de mapping devrait être sauvegardé en tant que `Event.hbm.xml`, juste dans le répertoire à côté du fichier source de la classe Java `Event`. Le nommage des fichiers de mapping peut être arbitraire, cependant le suffixe `hbm.xml` est devenu une convention dans la communauté des développeurs Hibernate. La structure du répertoire devrait ressembler à ça :

```
.
+lib
  <Hibernate et bibliothèques tierces>
+src
  Event.java
  Event.hbm.xml
```

Nous poursuivons avec la configuration principale d'Hibernate.

1.2.3. Configuration d'Hibernate

Nous avons maintenant une classe persistante et son fichier de mapping. Il est temps de configurer Hibernate. Avant ça, nous avons besoin d'une base de données. HSQL DB, un SGBD SQL basé sur Java et travaillant en mémoire, peut être téléchargé à partir du site web de HSQL. En fait, vous avez seulement besoin de `hsqldb.jar`. Placez ce fichier dans le répertoire `lib/` du dossier de développement.

Créez un répertoire appelé `data` à la racine du répertoire de développement - c'est là que HSQL DB stockera ses fichiers de données. Démarrez maintenant votre base de données en exécutant `java -classpath ../lib/hsqldb.jar org.hsqldb.Server` dans votre répertoire de données. Vous observez qu'elle démarre et ouvre une socket TCP/IP, c'est là que notre application se connectera plus tard. Si vous souhaitez démarrez à partir d'une nouvelle base de données pour ce tutoriel (faites `CTRL + C` dans la fenêtre the window), effacez le répertoire `data/` et redémarrez HSQL DB à nouveau.

Hibernate est la couche de votre application qui se connecte à cette base de données, donc il a besoin des informations de connexion. Les connexions sont établies à travers un pool de connexions JDBC, que nous devons aussi configurer. La distribution Hibernate contient différents outils de gestion de pools de connexions JDBC open source, mais pour ce didacticiel nous utiliserons le pool de connexions intégré à Hibernate. Notez que vous devez copier les bibliothèques requises dans votre classpath et utiliser une configuration de pool de connexions différente si vous voulez utiliser un logiciel de gestion de pools JDBC tiers avec une qualité de production.

Pour la configuration d'Hibernate, nous pouvons utiliser un simple fichier `hibernate.properties`, un fichier `hibernate.cfg.xml` légèrement plus sophistiqué, ou même une configuration complète par programmation. La plupart des utilisateurs préfèrent le fichier de configuration XML :

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>

        <!-- Database connection settings -->
        <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
        <property name="connection.url">jdbc:hsqldb:hsql://localhost</property>
        <property name="connection.username">sa</property>
        <property name="connection.password"></property>

        <!-- JDBC connection pool (use the built-in) -->
        <property name="connection.pool_size">1</property>

        <!-- SQL dialect -->
        <property name="dialect">org.hibernate.dialect.HSQLDialect</property>

        <!-- Enable Hibernate's automatic session context management -->
        <property name="current_session_context_class">thread</property>

        <!-- Disable the second-level cache  -->
        <property name="cache.provider_class">org.hibernate.cache.NoCacheProvider</property>

        <!-- Echo all executed SQL to stdout -->
        <property name="show_sql">true</property>

        <!-- Drop and re-create the database schema on startup -->
        <property name="hbm2ddl.auto">create</property>
```

```
        <mapping resource="events/Event.hbm.xml"/>

    </session-factory>

</hibernate-configuration>
```

Notez que cette configuration XML utilise une DTD différente. Nous configurons une `SessionFactory` d'Hibernate - une fabrique globale responsable d'une base de données particulière. Si vous avez plusieurs base de données, utilisez plusieurs configurations `<session-factory>`, généralement dans des fichiers de configuration différents (pour un démarrage plus facile).

Les quatre premiers éléments `property` contiennent la configuration nécessaire pour la connexion JDBC. L'élément `property` du dialecte spécifie quelle variante du SQL Hibernate va générer. La gestion automatique des sessions d'Hibernate pour les contextes de persistance sera détaillée très vite. L'option `hbm2ddl.auto` active la génération automatique des schémas de base de données - directement dans la base de données. Cela peut bien sûr aussi être désactivé (en supprimant l'option de configuration) ou redirigé vers un fichier avec l'aide de la tâche Ant `SchemaExport`. Finalement, nous ajoutons le(s) fichier(s) de mapping pour les classes persistantes.

Copiez ce fichier dans le répertoire source, il terminera dans la racine du classpath. Hibernate cherchera automatiquement, au démarrage, un fichier appelé `hibernate.cfg.xml` dans la racine du classpath.

1.2.4. Construction avec Ant

Nous allons maintenant construire le didacticiel avec Ant. Vous aurez besoin d'avoir Ant d'installé - récupérez-le à partir de la page de téléchargement de Ant [http://ant.apache.org/bindownload.cgi]. Comment installer Ant ne sera pas couvert ici. Référez-vous au manuel d'Ant [http://ant.apache.org/manual/index.html]. Après que vous aurez installé Ant, nous pourrons commencer à créer le fichier de construction. Il s'appellera `build.xml` et sera placé directement dans le répertoire de développement.

Un fichier de construction basique ressemble à ça :

```
<project name="hibernate-tutorial" default="compile">

    <property name="sourcedir" value="${basedir}/src"/>
    <property name="targetdir" value="${basedir}/bin"/>
    <property name="librarydir" value="${basedir}/lib"/>

    <path id="libraries">
        <fileset dir="${librarydir}">
            <include name="*.jar"/>
        </fileset>
    </path>

    <target name="clean">
        <delete dir="${targetdir}"/>
        <mkdir dir="${targetdir}"/>
    </target>

    <target name="compile" depends="clean, copy-resources">
      <javac srcdir="${sourcedir}"
             destdir="${targetdir}"
             classpathref="libraries"/>
    </target>

    <target name="copy-resources">
        <copy todir="${targetdir}">
            <fileset dir="${sourcedir}">
                <exclude name="**/*.java"/>
            </fileset>
        </copy>
    </target>
```

```
</project>
```

Cela dira à Ant d'ajouter tous les fichiers du répertoire lib finissant par `.jar` dans le classpath utilisé pour la compilation. Cela copiera aussi tous les fichiers source non Java dans le répertoire cible, par exemple les fichiers de configuration et de mapping d'Hibernate. Si vous lancez Ant maintenant, vous devriez obtenir cette sortie :

```
C:\hibernateTutorial\>ant
Buildfile: build.xml

copy-resources:
     [copy] Copying 2 files to C:\hibernateTutorial\bin

compile:
    [javac] Compiling 1 source file to C:\hibernateTutorial\bin

BUILD SUCCESSFUL
Total time: 1 second
```

1.2.5. Démarrage et aides

Il est temps de charger et de stocker quelques objets `Event`, mais d'abord nous devons compléter la configuration avec du code d'infrastructure. Nous devons démarrer Hibernate. Ce démarrage inclut la construction d'un objet `SessionFactory` global et le stocker quelque part facile d'accès dans le code de l'application. Une `SessionFactory` peut ouvrir des nouvelles `Session`s. Une `Session` représente une unité de travail simplement "threadée", la `SessionFactory` est un objet global "thread-safe", instancié une seule fois.

Nous créerons une classe d'aide `HibernateUtil` qui s'occupe du démarrage et rend la gestion des `Session`s plus facile. Regardons l'implémentation :

```
package util;

import org.hibernate.*;
import org.hibernate.cfg.*;

public class HibernateUtil {
    public static final SessionFactory sessionFactory;

    static {
        try {
            // Création de la SessionFactory à partir de hibernate.cfg.xml
            sessionFactory = new Configuration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            // Make sure you log the exception, as it might be swallowed
            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static final ThreadLocal session = new ThreadLocal();

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```

Cette classe ne produit pas seulement la `SessionFactory` globale dans un initialiseur statique (appelé une seule fois par la JVM lorsque la classe est chargée), elle masque le fait qu'elle exploite un singleton. Elle pourrait aussi obtenir la `SessionFactory` depuis JNDI dans un serveur d'applications.

Si vous nommez la `SessionFactory` dans votre fichier de configuration, Hibernate tentera la récupération

depuis JNDI. Pour éviter ce code, vous pouvez aussi utiliser un déploiement JMX et laisser le conteneur (compatible JMX) instancier et lier un `HibernateService` à JNDI. Ces options avancées sont détaillées dans la documentation de référence Hibernate.

Placez `HibernateUtil.java` dans le répertoire source de développement, et ensuite `Event.java` :

```
.
+lib
  <Hibernate and third-party libraries>
+src
  +events
    Event.java
    Event.hbm.xml
  +util
    HibernateUtil.java
  hibernate.cfg.xml
+data
build.xml
```

Cela devrait encore compiler sans problème. Nous avons finalement besoin de configurer le système de "logs" - Hibernate utilise commons-logging et vous laisse le choix entre log4j et le système de logs du JDK 1.4. La plupart des développeurs préfèrent log4j : copiez `log4j.properties` de la distribution d'Hibernate (il est dans le répertoire `etc/`) dans votre répertoire `src`, puis faites de même avec `hibernate.cfg.xml`. Regardez la configuration d'exemple et changez les paramètres si vous voulez une sortie plus verbeuse. Par défaut, seul le message de démarrage d'Hibernate est affiché sur la sortie standard.

L'infrastructure de ce didacticiel est complète - et nous sommes prêts à effectuer un travail réel avec Hibernate.

1.2.6. Charger et stocker des objets

Finalement nous pouvons utiliser Hibernate pour charger et stocker des objets. Nous écrivons une classe `EventManager` avec une méthode `main()` :

```
package events;
import org.hibernate.Session;

import java.util.Date;

import util.HibernateUtil;

public class EventManager {

    public static void main(String[] args) {
        EventManager mgr = new EventManager();

        if (args[0].equals("store")) {
            mgr.createAndStoreEvent("My Event", new Date());
        }

        HibernateUtil.getSessionFactory().close();
    }

    private void createAndStoreEvent(String title, Date theDate) {

        Session session = HibernateUtil.getSessionFactory().getCurrentSession();

        session.beginTransaction();

        Event theEvent = new Event();
        theEvent.setTitle(title);
        theEvent.setDate(theDate);

        session.save(theEvent);
```

```
        session.getTransaction().commit();
    }

}
```

Nous créons un nouvel objet `Event`, et le remettons à Hibernate. Hibernate s'occupe maintenant du SQL et exécute les `INSERT`s dans la base de données. Regardons le code de gestion de la `Session` et de la `Transaction` avant de lancer ça.

Une `Session` est une unité de travail. Pour le moment, nous allons faire les choses simplement et assumer une granularité un-un entre une `Session` hibernate et une transaction à la base de données. Pour isoler notre code du système de transaction sous-jacent (dans notre cas, du pure JDBC, mais cela pourrait être JTA), nous utilisons l'API `Transaction` qui est disponible depuis la `Session` Hibernate.

Que fait `sessionFactory.getCurrentSession()` ? Premièrement, vous pouvez l'invoquer autant de fois que vous le voulez et n'importe où du moment que vous avez votre `SessionFactory` (facile grâce à `HibernateUtil`). La méthode `getCurrentSession()` renvoie toujours l'unité de travail courante. Souvenez vous que nous avons basculé notre option de configuration au mécanisme basé sur le "thread" dans `hibernate.cfg.xml`. Par conséquent, l'unité de travail courante est liée au thread Java courant qui exécute notre application. Cependant, ce n'est pas tout, vous devez aussi considérer le scope, quand une unité de travail commence et quand elle finit.

Une `Session` commence lorsqu'elle est vraiment utilisée la première fois, lorsque nous appelons `getCurrentSession()` pour la première fois. Ensuite, elle est attachée par Hibernate au thread courant. Lorsque la transaction s'achève, par commit ou par rollback, Hibernate détache automatiquement la `Session` du thread et la ferme pour vous. Si vous invoquez `getCurrentSession()` une nouvelle fois, vous obtenez une nouvelle `Session` et pouvez entamer une nouvelle unité de travail. Ce modèle de programmation "*thread-bound*" est le moyen le plus populaire d'utiliser Hibernate, puisqu'il permet un découpage flexible de votre code (le code délimitant les transactions peut être séparé du code accédant aux données, nous verrons cela plus loin dans ce tutorial).

A propos du scope de l'unité de travail, la `Session` Hibernate devrait-elle être utilisée pour exécuter une ou plusieurs opérations en base de données ? L'exemple ci-dessus utilise une `Session` pour une opération. C'est une pure coïncidence, l'exemple est n'est seulement pas assez complexe pour montrer d'autres approches. Le scope d'une `Session` Hibernate est flexible mais vous ne devriez jamais concevoir votre application de manière à utiliser une nouvelle `Session` Hibernate pour *chaque* opération en base de données. Donc même si vous le voyez quelques fois dans les exemples (très simplistes) suivants, considérez *une session par operation* comme un anti-pattern. Une véritable application (web) est montrée plus loin dans ce tutorial.

Lisez Chapitre 11, *Transactions et accès concurrents* pour plus d'informations sur la gestion des transactions et leur démarcations. Nous n'avons pas géré les erreurs et rollback sur l'exemple précédent.

Pour lancer cette première routine, nous devons ajouter une cible appelable dans le fichier de construction de Ant :

```
<target name="run" depends="compile">
    <java fork="true" classname="events.EventManager" classpathref="libraries">
        <classpath path="${targetdir}"/>
        <arg value="${action}"/>
    </java>
</target>
```

La valeur de l'argument `action` correspond à la ligne de commande qui appelle la cible :

```
C:\hibernateTutorial\>ant run -Daction=store
```

Vous devriez voir, après la compilation, Hibernate démarrer et, en fonction de votre configuration, beaucoup de traces sur la sortie. À la fin vous trouverez la ligne suivante :

```
[java] Hibernate: insert into EVENTS (EVENT_DATE, title, EVENT_ID) values (?, ?, ?)
```

C'est l'`INSERT` exécuté par Hibernate, les points d'interrogation représentent les paramètres JDBC liés. Pour voir les valeurs liées aux arguments, ou pour réduire la verbosité des traces, vérifier votre `log4j.properties`.

Maintenant nous aimerions aussi lister les événements stockés, donc nous ajoutons une option à la méthode principale :

```
if (args[0].equals("store")) {
    mgr.createAndStoreEvent("My Event", new Date());
}
else if (args[0].equals("list")) {
    List events = mgr.listEvents();
    for (int i = 0; i < events.size(); i++) {
        Event theEvent = (Event) events.get(i);
        System.out.println("Event: " + theEvent.getTitle() +
                           " Time: " + theEvent.getDate());
    }
}
```

Nous ajoutons aussi une nouvelle méthode `listEvents()` :

```
private List listEvents() {

    Session session = HibernateUtil.getSessionFactory().getCurrentSession();

    session.beginTransaction();

    List result = session.createQuery("from Event").list();

    session.getTransaction().commit();

    return result;
}
```

Ce que nous faisons ici c'est utiliser une requête HQL (Hibernate Query Language) pour charger tous les objets `Event` existants de la base de données. Hibernate générera le SQL approprié, l'enverra à la base de données et peuplera des objets `Event` avec les données. Vous pouvez créer des requêtes plus complexes avec HQL, bien sûr.

Maintenant, pour exécuter et tester tout ça, suivez ces étapes :

- Exécutez `ant run -Daction=store` pour stocker quelque chose dans la base de données et, bien sûr, pour générer, avant, le schéma de la base de données grâce à hbm2ddl.
- Maintenant désactivez hbm2ddl en commentant la propriété dans votre fichier `hibernate.cfg.xml`. Généralement vous la laissez seulement activée dans des tests unitaires en continu, mais une autre exécution de hbm2ddl *effacerait* tout ce que vous avez stocké - le paramètre de configuration `create` se traduit en fait par "supprimer toutes les tables du schéma, puis re-créer toutes les tables, lorsque la SessionFactory est construite".

Si maintenant vous appelez Ant avec `-Daction=list`, vous devriez voir les événements que vous avez stockés jusque là. Vous pouvez bien sûr aussi appeler l'action `store` plusieurs fois.

1.3. Partie 2 - Mapper des associations

Nous avons mappé une classe d'une entité persistante vers une table. Partons de là et ajoutons quelques associations de classe. D'abord nous ajouterons des gens à notre application, et stockerons une liste d'événements auxquels ils participent.

1.3.1. Mapper la classe Person

La première version de la classe `Person` est simple :

```
package events;

public class Person {

    private Long id;
    private int age;
    private String firstname;
    private String lastname;

    public Person() {}

    // Accessor methods for all properties, private setter for 'id'

}
```

Créez un nouveau fichier de mapping appelé `Person.hbm.xml` (n'oubliez pas la référence à la DTD)

```
<hibernate-mapping>

    <class name="events.Person" table="PERSON">
        <id name="id" column="PERSON_ID">
            <generator class="native"/>
        </id>
        <property name="age"/>
        <property name="firstname"/>
        <property name="lastname"/>
    </class>

</hibernate-mapping>
```

Finalement, ajoutez la nouveau mapping à la configuration d'Hibernate :

```
<mapping resource="events/Event.hbm.xml"/>
<mapping resource="events/Person.hbm.xml"/>
```

Nous allons maintenant créer une association entre ces deux entités. Évidemment, des personnes peuvent participer aux événements, et des événements ont des participants. Les questions de conception que nous devons traiter sont : direction, cardinalité et comportement de la collection.

1.3.2. Une association unidirectionnelle basée sur Set

Nous allons ajouter une collection d'événements à la classe `Person`. De cette manière nous pouvons facilement naviguer dans les événements d'une personne particulière, sans exécuter une requête explicite - en appelant `aPerson.getEvents()`. Nous utilisons une collection Java, un `Set`, parce que la collection ne contiendra pas d'éléments dupliqués et l'ordre ne nous importe pas.

Nous avons besoin d'une association unidirectionnelle, pluri-valuée, implémentée avec un `Set`. Écrivons le code pour ça dans les classes Java et mappons les :

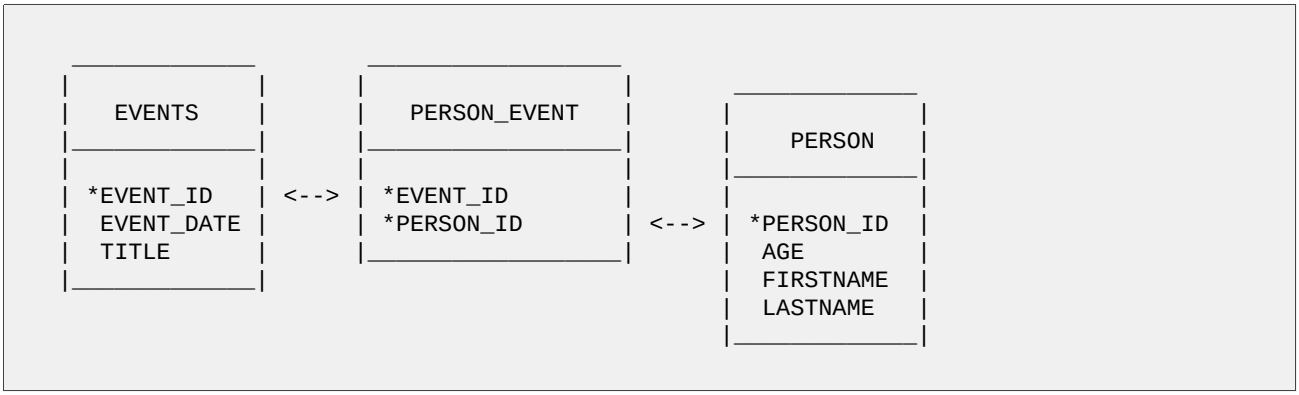
```
public class Person {

    private Set events = new HashSet();

    public Set getEvents() {
        return events;
    }

    public void setEvents(Set events) {
        this.events = events;
    }
}
```

D'abord nous mappons cette association, mais pensez à l'autre côté. Clairement, nous pouvons la laisser unidirectionnelle. Ou alors, nous pourrions créer une autre collection sur `Event`, si nous voulons être capable de la parcourir de manière bidirectionnelle, c'est-à-dire avoir `anEvent.getParticipants()`. Ce n'est pas nécessaire d'un point de vue fonctionnel. Vous pourrez toujours exécuter une requête explicite pour récupérer les participants d'un "event" particulier. Ce choix de conception vous est laissé, mais ce qui reste certains est la cardinalité de l'association: "plusieurs" des deux côtés, nous appelons cela une association *many-to-many*. Par conséquent nous utilisons un mapping Hibernate many-to-many:

```
<class name="events.Person" table="PERSON">
    <id name="id" column="PERSON_ID">
        <generator class="native"/>
    </id>
    <property name="age"/>
    <property name="firstname"/>
    <property name="lastname"/>

    <set name="events" table="PERSON_EVENT">
        <key column="PERSON_ID"/>
        <many-to-many column="EVENT_ID" class="events.Event"/>
    </set>

</class>
```

Hibernate supporte toutes sortes de mapping de collection, un `<set>` étant le plus commun. Pour une association many-to-many (ou une relation d'entité *n:m*), une table d'association est requise. Chaque ligne dans cette table représente un lien entre une personne et un événement. Le nom de la table est configuré avec l'attribut `table` de l'élément `set`. Le nom de la colonne identifiant dans l'association, du côté de la personne, est défini avec l'élément `<key>`, et le nom de la colonne pour l'événement dans l'attribut `column` de `<many-to-many>`. Vous devez aussi donner à Hibernate la classe des objets de votre collection (c'est-à-dire : la classe de l'autre côté de la collection).

Le schéma de base de données pour ce mapping est donc :



1.3.3. Travailler avec l'association

Réunissons quelques personnes et quelques événements dans une nouvelle méthode dans `EventManager` :

```
private void addPersonToEvent(Long personId, Long eventId) {

    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();

    Person aPerson = (Person) session.load(Person.class, personId);
    Event anEvent = (Event) session.load(Event.class, eventId);

    aPerson.getEvents().add(anEvent);

    session.getTransaction().commit();
}
```

Après le chargement d'une `Person` et d'un `Event`, modifiez simplement la collection en utilisant les méthodes normales de la collection. Comme vous pouvez le voir, il n'y a pas d'appel explicite à `update()` ou `save()`, Hibernate détecte automatiquement que la collection a été modifiée et a besoin d'être mise à jour. Ceci est appelé *la vérification sale automatique* (NdT : "automatic dirty checking"), et vous pouvez aussi l'essayer en modifiant le nom ou la propriété date de n'importe lequel de vos objets. Tant qu'ils sont dans un état *persistant*, c'est-à-dire, liés à une `Session` Hibernate particulière (c-à-d qu'ils ont juste été chargés ou sauvegardés dans une unité de travail), Hibernate surveille les changements et exécute le SQL correspondant. Le processus de synchronisation de l'état de la mémoire avec la base de données, généralement seulement à la fin d'une unité de travail, est appelé *flushing*. Dans notre code, l'unité de travail s'achève par un commit (ou rollback) de la transaction avec la base de données - comme défini par notre option `thread` de configuration pour la classe `CurrentSessionContext`.

Vous pourriez bien sûr charger une personne et un événement dans différentes unités de travail. Ou vous modifiez un objet à l'extérieur d'une `Session`, s'il n'est pas dans un état persistant (s'il était persistant avant, nous appelons cet état *détaché*). Vous pouvez même modifier une collection lorsqu'elle est détachée:

```
private void addPersonToEvent(Long personId, Long eventId) {

    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();

    Person aPerson = (Person) session
            .createQuery("select p from Person p left join fetch p.events where p.id = :pid")
            .setParameter("pid", personId)
            .uniqueResult(); // Eager fetch the collection so we can use it detached

    Event anEvent = (Event) session.load(Event.class, eventId);

    session.getTransaction().commit();

    // End of first unit of work

    aPerson.getEvents().add(anEvent); // aPerson (and its collection) is detached

    // Begin second unit of work

    Session session2 = HibernateUtil.getSessionFactory().getCurrentSession();
    session2.beginTransaction();

    session2.update(aPerson); // Reattachment of aPerson

    session2.getTransaction().commit();
}
```

L'appel à `update` rend un objet détaché à nouveau persistant, vous pourriez dire qu'il le lie à une unité de travail, ainsi toutes les modifications (ajout, suppression) que vous avez faites pendant qu'il était détaché peuvent être sauvegardées dans la base de données (il se peut que vous ayez besoin de modifier quelques unes

des méthodes précédentes pour retourner cet identifiant).

Cela n'a pas grand intérêt dans notre situation, mais c'est un concept important qu'il vous faut concevoir dans votre application. Pour le moment, complétez cet excercice en ajoutant une nouvelle action à la méthode principale de l'`EventManager` et invoquez la depuis la ligne de commande. Si vous avez besoin des identifiants d'un client et d'un évènement - la méthode `save()` vous les retourne (vous devrez peut être modifier certaines méthodes précédentes pour retourner ces identifiants):

```
else if (args[0].equals("addpersontoevent")) {
    Long eventId = mgr.createAndStoreEvent("My Event", new Date());
    Long personId = mgr.createAndStorePerson("Foo", "Bar");
    mgr.addPersonToEvent(personId, eventId);
    System.out.println("Added person " + personId + " to event " + eventId);
```

C'était un exemple d'une association entre deux classes de même importance, deux entités. Comme mentionné plus tôt, il y a d'autres classes et d'autres types dans un modèle typique, généralement "moins importants". Vous en avez déjà vu certains, comme un `int` ou une `String`. Nous appelons ces classes des *types de valeur*, et leurs instances *dépendent* d'une entité particulière. Des instances de ces types n'ont pas leur propre identité, elles ne sont pas non plus partagées entre des entités (deux personnes ne référencent pas le même objet `firstname`, même si elles ont le même prénom). Bien sûr, des types de valeur ne peuvent pas seulement être trouvés dans le JDK (en fait, dans une application Hibernate toutes les classes du JDK sont considérées comme des types de valeur), vous pouvez aussi écrire vous-même des classes dépendantes, `Address` ou `MonetaryAmount`, par exemple.

Vous pouvez aussi concevoir une collection de types de valeur. C'est conceptuellement très différent d'une collection de références vers d'autres entités, mais très ressemblant en Java.

1.3.4. Collection de valeurs

Nous ajoutons une collection d'objets de type de valeur à l'entité `Person`. Nous voulons stocker des adresses email, donc le type que nous utilisons est `String`, et la collection est encore un `Set` :

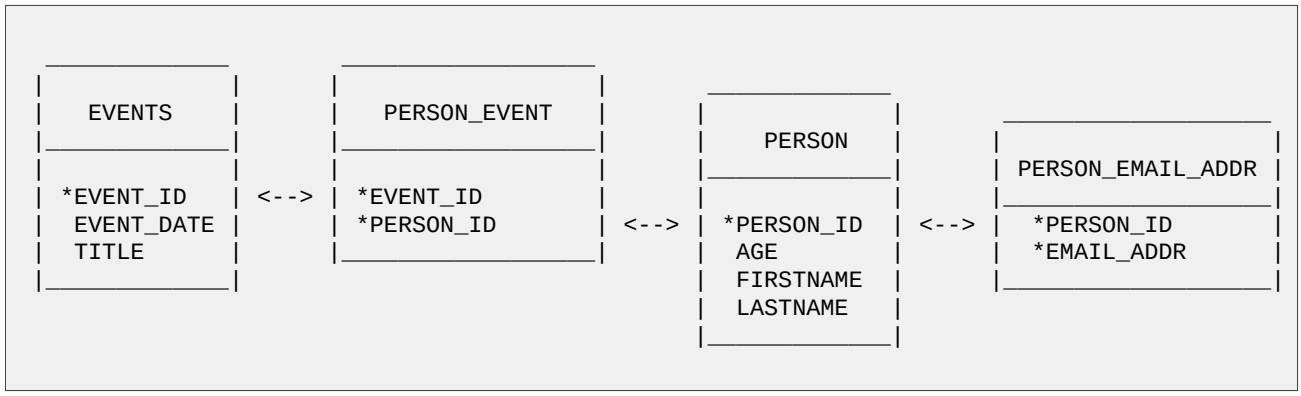
```
private Set emailAddresses = new HashSet();

public Set getEmailAddresses() {
    return emailAddresses;
}

public void setEmailAddresses(Set emailAddresses) {
    this.emailAddresses = emailAddresses;
}
```

Le mapping de ce `Set` :

```
<set name="emailAddresses" table="PERSON_EMAIL_ADDR">
    <key column="PERSON_ID"/>
    <element type="string" column="EMAIL_ADDR"/>
</set>
```

La différence comparée au mapping vu plus tôt est la partie `element`, laquelle dit à Hibernate que la collection ne contient pas de références vers une autre entité, mais une collection d'éléments de type `String` (le nom en minuscule vous indique que c'est un type/convertisseur du mapping Hibernate). Une fois encore, l'attribut `table` de l'élément `set` détermine le nom de la table pour la collection. L'élément `key` définit le nom de la colonne de la clef étrangère dans la table de la collection. L'attribut `column` dans l'élément `element` définit le nom de la colonne où les valeurs de `String` seront réellement stockées.

Regardons le schéma mis à jour :



Vous pouvez voir que la clef primaire de la table de la collection est en fait une clef composée, utilisant deux colonnes. Ceci implique aussi qu'il ne peut pas y avoir d'adresses email dupliquées par personne, ce qui est exactement la sémantique dont nous avons besoin pour un ensemble en Java.

Vous pouvez maintenant tester et ajouter des éléments à cette collection, juste comme nous l'avons fait avant en liant des personnes et des événements. C'est le même code en Java.

```
private void addEmailToPerson(Long personId, String emailAddress) {

    Session session = HibernateUtil.getSessionFactory().getCurrentSession();
    session.beginTransaction();

    Person aPerson = (Person) session.load(Person.class, personId);

    // The getEmailAddresses() might trigger a lazy load of the collection
    aPerson.getEmailAddresses().add(emailAddress);

    session.getTransaction().commit();
}
```

Cette fois ci, nous n'avons pas utilisé une requête de chargement agressif (*fetch*) pour initialiser la collection. Par conséquent, l'invocation du getter déclenchera un select supplémentaire pour l'initialiser. Traquez les logs SQL et tentez d'optimiser ce cas avec un chargement aggressif.

1.3.5. Associations bidirectionnelles

Ensuite nous allons mapper une association bidirectionnelle - faire fonctionner l'association entre une personne et un événement à partir des deux côtés en Java. Bien sûr, le schéma de la base de données ne change pas, nous avons toujours une pluralité many-to-many. Une base de données relationnelle est plus flexible qu'un langage de programmation réseau, donc elle n'a pas besoin de direction de navigation - les données peuvent être vues et récupérées de toutes les manières possibles.

D'abord, ajouter une collection de participants à la classe `Event` :

```
private Set participants = new HashSet();

public Set getParticipants() {
    return participants;
}

public void setParticipants(Set participants) {
    this.participants = participants;
}
```

Maintenant mapper ce côté de l'association aussi, dans `Event.hbm.xml`.

```
<set name="participants" table="PERSON_EVENT" inverse="true">
```

```
    <key column="EVENT_ID"/>
    <many-to-many column="PERSON_ID" class="events.Person"/>
</set>
```

Comme vous le voyez, ce sont des mappings de `sets` normaux dans les deux documents de mapping. Notez que les noms de colonne dans `key` et `many-to-many` sont inversés dans les 2 documents de mapping. L'ajout le plus important ici est l'attribut `inverse="true"` dans l'élément `set` du mapping de la collection des `Events`.

Ce que signifie qu'Hibernate devrait prendre l'autre côté - la classe `Person` - s'il a besoin de renseigner des informations à propos du lien entre les deux. Ce sera beaucoup plus facile à comprendre une fois que vous verrez comment le lien bidirectionnel entre les deux entités est créé.

1.3.6. Travailler avec des liens bidirectionnels

Premièrement, gardez à l'esprit qu'Hibernate n'affecte pas la sémantique normale de Java. Comment avons-nous créé un lien entre une `Person` et un `Event` dans l'exemple unidirectionnel ? Nous avons ajouté une instance de `Event` à la collection des références d'événement d'une instance de `Person`. Donc, évidemment, si vous voulons rendre ce lien bidirectionnel, nous devons faire la même chose de l'autre côté - ajouter une référence de `Person` à la collection d'un `Event`. Cette "configuration du lien des deux côtés" est absolument nécessaire et vous ne devriez jamais oublier de le faire.

Beaucoup de développeurs programment de manière défensive et créent des méthodes de gestion de lien pour affecter correctement les deux côtés, par exemple dans `Person` :

```
protected Set getEvents() {
    return events;
}

protected void setEvents(Set events) {
    this.events = events;
}

public void addToEvent(Event event) {
    this.getEvents().add(event);
    event.getParticipants().add(this);
}

public void removeFromEvent(Event event) {
    this.getEvents().remove(event);
    event.getParticipants().remove(this);
}
```

Notez que les méthodes get et set pour la collection sont maintenant protégées - ceci permet à des classes du même paquet et aux sous-classes d'accéder encore aux méthodes, mais empêche n'importe qui d'autre de mettre le désordre directement dans les collections (enfin, presque). Vous devriez probablement faire de même avec la collection de l'autre côté.

Et à propos de l'attribut de mapping `inverse` ? Pour vous, et pour Java, un lien bidirectionnel est simplement une manière de configurer correctement les références des deux côtés. Hibernate n'a cependant pas assez d'informations pour ordonner correctement les expressions SQL `INSERT` et `UPDATE` (pour éviter les violations de contrainte), et a besoin d'aide pour gérer proprement les associations bidirectionnelles. Rendre `inverse` un côté d'une assocation dit à Hibernate de l'ignorer essentiellement, pour le considérer comme un *miroir* de l'autre côté. C'est tout ce qui est nécessaire à Hibernate pour découvrir tout des problèmes de transformation d'un modèle de navigation directionnelle vers un schéma SQL de base de données. Les règles dont vous devez vous souvenir sont : toutes les associations bidirectionnelles ont besoin d'un côté marqué `inverse`. Dans une association un-vers-plusieurs vous pouvez choisir n'importe quel côté, il n'y a pas de différence.

1.4. Part 3 - L'application web EventManager

Une application web Hibernate utilise la `Session` et `Transaction` comme une application standalone. Cependant, quelques patterns sont utiles. Nous allons coder une `EventManagerServlet`. Cette servlet peut lister tous les évènements stockés dans la base de données, et fournir une formulaire HTML pour saisir d'autres évènements.

1.4.1. Ecrire la servlet de base

Créons une nouvelle classe dans notre répertoire source, dans le package `events` :

```
package events;

// Imports

public class EventManagerServlet extends HttpServlet {

    // Servlet code
}
```

La servlet n'accepte que les requêtes HTTP `GET`, la méthode à implémenter est donc `doGet()` :

```
protected void doGet(HttpServletRequest request,
                     HttpServletResponse response)
        throws ServletException, IOException {

    SimpleDateFormat dateFormatter = new SimpleDateFormat("dd.MM.yyyy");

    try {
        // Début de l'unité de travail
        HibernateUtil.getSessionFactory()
                .getCurrentSession().beginTransaction();

        // Traitement de la requête et rendu de la page...

        // Fin de l'unité de travail
        HibernateUtil.getSessionFactory()
                .getCurrentSession().getTransaction().commit();

    } catch (Exception ex) {
        HibernateUtil.getSessionFactory()
                .getCurrentSession().getTransaction().rollback();
        throw new ServletException(ex);
    }

}
```

Le pattern que nous utilisons ici est appelé *session-per-request*. Lorsqu'une requête appelle la servlet, une nouvelle `Session` Hibernate est ouverte à l'invocation de `getCurrentSession()` sur la `SessionFactory`. Ensuite, une transaction avec la base de données est démarrée — tous les accès à la base de données interviennent au sein de la transaction, peu importe que les données soient lues ou écrites (nous n'utilisons pas le mode auto-commit dans les applications).

N'utilisez pas une nouvelle `Session` Hibernate pour chaque opération en base de données. Utilisez une `Session` Hibernate qui porte sur l'ensemble de la requête. Utlisez `getCurrentSession()`, ainsi elle est automatiquement attachée au thread Java courant.

Ensuite, les actions possibles de la requêtes sont exécutées et la réponse HTML est rendue. Nous en parlerons plus tard.

Enfin, l'unité de travail s'achève lorsque l'exécution et le rendu sont achevés. Si un problème survient lors de ces deux phases, une exception est soulevée et la transaction avec la base de données subit un rollback. Voila pour le pattern `session-per-request`. Au lieu d'avoir un code de délimitant les transactions au sein de chaque servlet, vous pouvez écrire un filtre de servlet. Voir le site Hibernate et le Wiki pour plus d'information sur ce pattern, appelé *Open Session in View* — vous en aurez besoin dès que vous utiliserez des JSPs et non plus des servlets pour le rendu de vos vues.

1.4.2. Procéder et rendre

Implémentons l'exécution de la requête et le rendu de la page.

```
// Write HTML header
PrintWriter out = response.getWriter();
out.println("<html><head><title>Event Manager</title></head><body>");

// Handle actions
if ( "store".equals(request.getParameter("action")) ) {

    String eventTitle = request.getParameter("eventTitle");
    String eventDate = request.getParameter("eventDate");

    if ( "".equals(eventTitle) || "".equals(eventDate) ) {
        out.println("<b><i>Please enter event title and date.</i></b>");
    } else {
        createAndStoreEvent(eventTitle, dateFormatter.parse(eventDate));
        out.println("<b><i>Added event.</i></b>");
    }
}

// Print page
printEventForm(out);
listEvents(out);

// Write HTML footer
out.println("</body></html>");
out.flush();
out.close();
```

Ce style de code avec un mix de Java et d'HTML ne serait pas scalable dans une application plus complexe—gardez à l'esprit que nous ne faisons qu'illustrer les concepts basiques d'Hibernate dans ce tutoriel. Ce code affiche une en tête et un pied de page HTML. Dans cette page, sont affichés un formulaire pour la saisie d'évènements ainsi qu'une liste de tous les évènements de la base de données. La première méthode est triviale est ne fait que sortir de l'HTML:

```
private void printEventForm(PrintWriter out) {
    out.println("<h2>Add new event:</h2>");
    out.println("<form>");
    out.println("Title: <input name='eventTitle' length='50'/><br/>");
    out.println("Date (e.g. 24.12.2009): <input name='eventDate' length='10'/><br/>");
    out.println("<input type='submit' name='action' value='store'/>");
    out.println("</form>");
}
```

La méthode `listEvents()` utilise la `Session` Hibernate liée au thread courant pour exécuter la requête:

```
private void listEvents(PrintWriter out, SimpleDateFormat dateFormatter) {

    List result = HibernateUtil.getSessionFactory()
                    .getCurrentSession().createCriteria(Event.class).list();
    if (result.size() > 0) {
        out.println("<h2>Events in database:</h2>");
        out.println("<table border='1'>");
        out.println("<tr>");
```

```
        out.println("<th>Event title</th>");
        out.println("<th>Event date</th>");
        out.println("</tr>");
        for (Iterator it = result.iterator(); it.hasNext();) {
            Event event = (Event) it.next();
            out.println("<tr>");
            out.println("<td>" + event.getTitle() + "</td>");
            out.println("<td>" + dateFormatter.format(event.getDate()) + "</td>");
            out.println("</tr>");
        }
        out.println("</table>");
    }
}
```

FEnfin, l'action `store` renvoie à la méthode `createAndStoreEvent()`, qui utilise aussi la `Session` du thread courant:

```
protected void createAndStoreEvent(String title, Date theDate) {
    Event theEvent = new Event();
    theEvent.setTitle(title);
    theEvent.setDate(theDate);

    HibernateUtil.getSessionFactory()
                    .getCurrentSession().save(theEvent);
}
```

La servlet est faite. Une requête à la servlet sera exécutée par une seule `Session` et `Transaction`. Comme pour une application standalone, Hibernate peut automatiquement lier ces objets au thread courant d'exécution. Cela vous laisse la liberté de séparer votre code en couches et d'accéder à la `SessionFactory` par le moyen que vous voulez. Généralement, vous utiliserez des conceptions plus sophistiquées et déplacerez le code d'accès aux données dans une couche DAO. Voir le wiki Hibernate pour plus d'exemples.

1.4.3. Déployer et tester

Pour déployer cette application, vous devez créer une archive Web, un War. Ajoutez la cible Ant suivante dans votre `build.xml`:

```
<target name="war" depends="compile">
    <war destfile="hibernate-tutorial.war" webxml="web.xml">
        <lib dir="${librarydir}">
          <exclude name="jsdk*.jar"/>
        </lib>

        <classes dir="${targetdir}"/>
    </war>
</target>
```

Cette cible créé un fichier nommé `hibernate-tutorial.war` dans le répertoire de votre projet. Elle package les bibliothèques et le descripteur `web.xml` qui est attendu dans le répertoire racine de votre projet:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4"
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web-app_2_4.xs

    <servlet>
        <servlet-name>Event Manager</servlet-name>
        <servlet-class>events.EventManagerServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>Event Manager</servlet-name>
```

```
        <url-pattern>/eventmanager</url-pattern>
    </servlet-mapping>
</web-app>
```

Avant de compiler et déployer l'application web, notez qu'une bibliothèque supplémentaire est requise: `jsdk.jar`. C'est le kit de développement de Servlet Java, si vous ne disposez pas de cette bibliothèque, prenez la sur le site de Sun et copiez la dans votre répertoire des bibliothèques. Cependant, elle ne sera utilisée uniquement pour la compilation et sera exclue du paackage WAR.

Pour construire et déployer, appelez `ant war` dans votre projet et copier le fichier `hibernate-tutorial.war` dans le répertoire `webapp` de tomcat Si vous n'avez pas installé Tomcat, téléchargez le et suivez la notice d'installation. Vous n'avez pas à modifier la configuration Tomcat pour déployer cette application.

Une fois l'application déployée et Tomcat lancé, accédez à l'application via `http://localhost:8080/hibernate-tutorial/eventmanager`. Assurez vous de consulter les traces tomcat pour observer l'initialisation d'Hibernate à la première requête touchant votre servlet (l'initialisation statique dans `HibernateUtil` est invoquée) et pour vérifier qu'aucune exception ne survienne.

1.5. Résumé

Ce didacticiel a couvert les bases de l'écriture d'une simple application Hibernate ainsi qu'une petite application web.

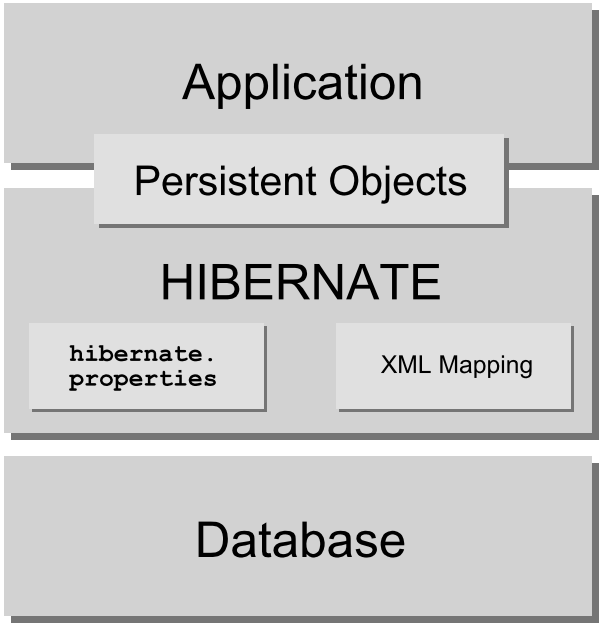
Si vous êtes déjà confiants avec Hibernate, continuez à parcourir les sujets que vous trouvez intéressants à travers la table des matières de la documentation de référence - les plus demandés sont le traitement transactionnel (Chapitre 11, *Transactions et accès concurrents*), la performance des récupérations d'information (Chapitre 19, *Améliorer les performances*), ou l'utilisation de l'API (Chapitre 10, *Travailler avec des objets*) et les fonctionnalités des requêtes (Section 10.4, « Requêtage »).

N'oubliez pas de vérifier le site web d'Hibernate pour d'autres didacticiels (plus spécialisés).

Chapitre 2. Architecture

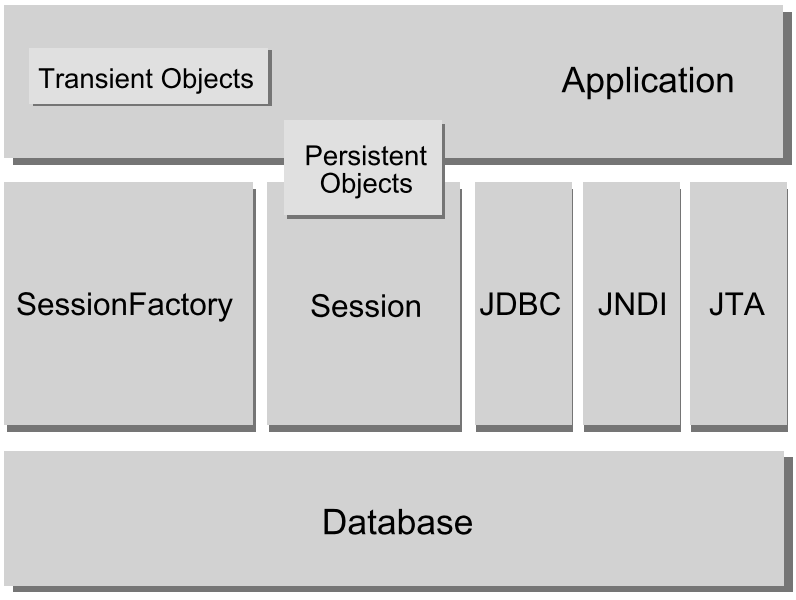
2.1. Généralités

Voici une vue (très) haut niveau de l'architecture d'Hibernate :



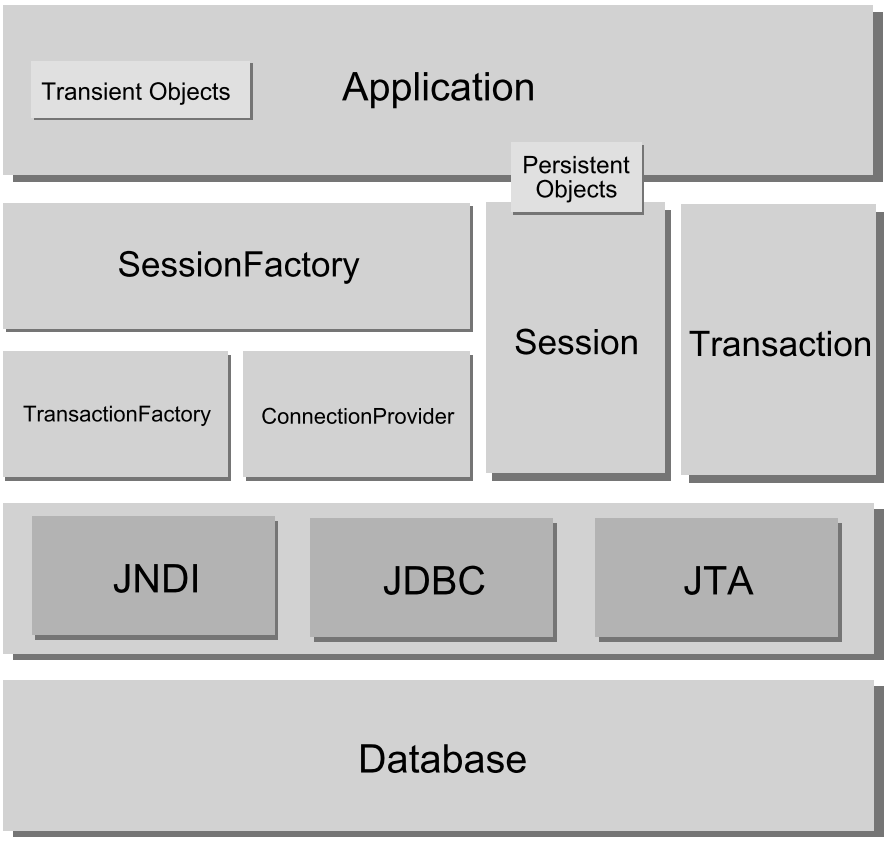
Ce diagramme montre Hibernate utilisant une base de données et des données de configuration pour fournir un service de persistance (et des objets persistants) à l'application.

Nous aimerions décrire une vue plus détaillée de l'architecture. Malheureusement, Hibernate est flexible et supporte différentes approches. Nous allons en montrer les deux extrêmes. L'architecture légère laisse l'application fournir ses propres connexions JDBC et gérer ses propres transactions. Cette approche utilise le minimum des APIs Hibernate :



L'architecture la plus complète abstrait l'application des APIs JDBC/JTA sous-jacentes et laisse Hibernate

s'occuper des détails.



Voici quelques définitions des objets des diagrammes :

SessionFactory (`org.hibernate.SessionFactory`)
: Un cache threadsafe (immuable) des mappings vers une (et une seule) base de données. Une factory (fabrique) de `Session` et un client de `ConnectionProvider`. Peut contenir un cache optionnel de données (de second niveau) qui est réutilisable entre les différentes transactions que cela soit au sein du même processus (JVLM) ou par plusieurs n½uds d'un cluster.

Session (`org.hibernate.Session`)
: Un objet mono-threadé, à durée de vie courte, qui représente une conversation entre l'application et l'entrepôt de persistance. Encapsule une connexion JDBC. Factory (fabrique) des objets `Transaction`. Contient un cache (de premier niveau) des objets persistants, ce cache est obligatoire. Il est utilisé lors de la navigation dans le graphe d'objets ou lors de la récupération d'objets par leur identifiant.

Objets et Collections persistants
: Objets mono-threadés à vie courte contenant l'état de persistance et la fonction métier. Ceux-ci sont en général les objets de type JavaBean (ou POJOs) ; la seule particularité est qu'ils sont associés avec une (et une seule) `Session`. Dès que la `Session` est fermée, ils seront détachés et libres d'être utilisés par n'importe laquelle des couches de l'application (ie. de et vers la présentation en tant que Data Transfer Objects - DTO : objet de transfert de données).

Objets et collections transients
: Instances de classes persistantes qui ne sont actuellement pas associées à une `Session`. Elles ont pu être instanciées par l'application et ne pas avoir (encore) été persistées ou elle ont pu être instanciées par une `Session` fermée.

Transaction (`org.hibernate.Transaction`)

(Optionnel) Un objet mono-threadé à vie courte utilisé par l'application pour définir une unité de travail atomique. Abstrait l'application des transactions sous-jacentes qu'elles soient JDBC, JTA ou CORBA. Une `Session` peut fournir plusieurs `Transaction`s dans certains cas. Toutefois, la délimitation des transactions, via l'API d'Hibernate ou par la `Transaction` sous-jacente, n'est jamais optionnelle!

ConnectionProvider (`org.hibernate.connection.ConnectionProvider`)
(Optionnel) Une fabrique de (pool de) connexions JDBC. Abstrait l'application de la `Datasource` ou du `DriverManager` sous-jacent. Non exposé à l'application, mais peut être étendu/implémenté par le développeur.

TransactionFactory (`org.hibernate.TransactionFactory`)
(Optionnel) Une fabrique d'instances de `Transaction`. Non exposé à l'application, mais peut être étendu/implémenté par le développeur.

Interfaces d'extension
Hibernate fournit de nombreuses interfaces d'extensions optionnelles que vous pouvez implémenter pour personnaliser le comportement de votre couche de persistance. Reportez vous à la documentation de l'API pour plus de détails.

Dans une architecture légère, l'application n'aura pas à utiliser les APIs `Transaction`/`TransactionFactory` et/ou n'utilisera pas les APIs `ConnectionProvider` pour utiliser JTA ou JDBC.

2.2. Etats des instances

Une instance d'une classe persistante peut être dans l'un des trois états suivants, définis par rapport à un *contexte de persistance*. L'objet `Session` d'hibernate correspond à ce concept de contexte de persistance :

passager (transient)
L'instance n'est pas et n'a jamais été associée à un contexte de persistance. Elle ne possède pas d'identité persistante (valeur de clé primaire)

persistant
L'instance est associée au contexte de persistance. Elle possède une identité persistante (valeur de clé primaire) et, peut-être, un enregistrement correspondant dans la base. Pour un contexte de persistance particulier, Hibernate *garantit* que l'identité persistante est équivalente à l'identité Java (emplacement mémoire de l'objet)

détaché
L'instance a été associée au contexte de persistance mais ce contexte a été fermé, ou l'instance a été sérialisée vers un autre processus. Elle possède une identité persistante et peut-être un enregistrement correspondant dans la base. Pour des instances détachées, Hibernate ne donne aucune garantie sur la relation entre l'identité persistante et l'identité Java.

2.3. Intégration JMX

JMX est le standard J2EE de gestion des composants Java. Hibernate peut être géré via un service JMX standard. Nous fournissons une implémentation d'un MBean dans la distribution : `org.hibernate.jmx.HibernateService`.

Pour avoir un exemple sur la manière de déployer Hibernate en tant que service JMX dans le serveur d'application JBoss Application Server, référez vous au guide utilisateur JBoss (JBoss User Guide). Si vous

déployez Hibernate via JMX sur JBoss AS, vous aurez également les bénéfices suivants :

- *Gestion de la session :* Le cycle de vie de la `Session` Hibernate peut être automatiquement limitée à la portée d'une transaction JTA. Cela signifie que vous n'avez plus besoin d'ouvrir et de fermer la `Session` manuellement, cela devient le travail de l'intercepteur EJB de JBoss. Vous n'avez pas non plus à vous occuper des démarcations des transactions dans votre code (sauf si vous voulez écrire une couche de persistance qui soit portable, dans ce cas vous pouvez utiliser l'API optionnelle `Transaction` d'Hibernate). Vous appelez l'`HibernateContext` pour accéder à la `Session`.
- *Déploiement HAR :* Habituellement vous déployez le service JMX Hibernate en utilisant le descripteur de déploiement de JBoss (dans un fichier EAR et/ou un SAR), il supporte toutes les options de configuration usuelles d'une `SessionFactory` Hibernate. Cependant, vous devez toujours nommer tous vos fichiers de mapping dans le descripteur de déploiement. Si vous décidez d'utiliser le déploiement optionnel sous forme de HAR, JBoss détectera automatiquement tous vos fichiers de mapping dans votre fichier HAR.

Consultez le guide d'utilisation de JBoss AS pour plus d'informations sur ces options.

Les statistiques pendant l'exécution d'Hibernate (au runtime) sont une autre fonctionnalité disponible en tant que service JMX. Voyez pour cela Section 3.4.6, « Statistiques Hibernate ».

2.4. Support JCA

Hibernate peut aussi être configuré en tant que connecteur JCA. Référez-vous au site web pour de plus amples détails. Il est important de noter que le support JCA d'Hibernate est encore considéré comme expérimental.

2.5. Sessions Contextuelles

Certaines applications utilisant Hibernate ont besoin d'une sorte de session "contextuelle", où une session est liée à la portée d'un contexte particulier. Cependant, les applications ne définissent pas toutes la notion de contexte de la même manière, et différents contextes définissent différentes portées à la notion de "courant". Les applications à base d'Hibernate, versions précédentes à la 3.0 utilisaient généralement un principe maison de sessions contextuelles basées sur le `ThreadLocal`, ainsi que sur des classes utilitaires comme `HibernateUtil`, ou utilisaient des framework tiers (comme Spring ou Pico) qui fournissaient des sessions contextuelles basées sur l'utilisation de proxy/interception.

A partir de la version 3.0.1, Hibernate a ajouté la méthode `SessionFactory.getCurrentSession()`. Initialement, cela demandait l'usage de transactions `JTA`, où la transaction `JTA` définissait la portée et le contexte de la session courante. L'équipe Hibernate pense que, étant donnée la maturité des implémentations de `JTA TransactionManager` , la plupart (sinon toutes) des applications devraient utiliser la gestion des transactions par `JTA` qu'elles soient ou non déployées dans un conteneur `J2EE`. Par conséquent, vous devriez toujours contextualiser vos sessions, si vous en avez besoin, via la méthode basée sur JTA.

Cependant, depuis la version 3.1, la logique derrière `SessionFactory.getCurrentSession()` est désormais branchable. A cette fin, une nouvelle interface d'extension (`org.hibernate.context.CurrentSessionContext`) et un nouveau paramètre de configuration (`hibernate.current_session_context_class`) ont été ajoutés pour permettre de configurer d'autres moyens de définir la portée et le contexte des sessions courantes.

Allez voir les Javadocs de l'interface `org.hibernate.context.CurrentSessionContext` pour une description détaillée de son contrat. Elle définit une seule méthode, `currentSession()`, depuis laquelle l'implémentation est responsable de traquer la session courante du contexte. Hibernate fournit deux implémentation de cette

interface.

- `org.hibernate.context.JTASessionContext` - les sessions courantes sont associées à une transaction `JTA`. La logique est la même que l'ancienne approche basée sur JTA. Voir les javadocs pour les détails.
- `org.hibernate.context.ThreadLocalSessionContext` - les sessions courantes sont associées au thread d'exécution. Voir les javadocs pour les détails.
- `org.hibernate.context.ManagedSessionContext` - les sessions courantes sont traquées par l'exécution du thread. Toutefois, vous êtes responsable de lier et délier une instance de `Session` avec les méthodes statiques de cette classes, qui n'ouvre, ne flush ou ne ferme jamais de `Session`.

Les deux implémentations fournissent un modèle de programmation de type "une session - une transaction à la base de données", aussi connu sous le nom de *session-per-request*. Le début et la fin d'une session Hibernate sont définis par la durée d'une transaction de base de données. Si vous utilisez une démarcation programmatique de la transaction (par exemple sous J2SE ou JTA/UserTransaction/BMT), nous vous conseillons d'utiliser l'API Hibernate `Transaction` pour masquer le système de transaction utilisé. Si vous exécutez sous un conteneur EJB qui supporte CMT, vous n'avez besoin d'aucune opérations de démarcations de session ou transaction dans votre code puisque tout est géré de manière déclarative. Référez vous à Chapitre 11, *Transactions et accès concurrents* pour plus d'informations et des exemples de code.

Le paramètre de configuration `hibernate.current_session_context_class` définit quelle implémentation de `org.hibernate.context.CurrentSessionContext` doit être utilisée. Notez que pour assurer la compatibilité avec les versions précédentes, si ce paramètre n'est pas défini mais qu'un `org.hibernate.transaction.TransactionManagerLookup` est configuré, Hibernate utilisera le `org.hibernate.context.JTASessionContext`. La valeur de ce paramètre devrait juste nommer la classe d'implémentation à utiliser, pour les deux implémentations fournies, il y a cependant deux alias correspondant: "jta" et "thread".

Chapitre 3. Configuration

Parce qu'Hibernate est conçu pour fonctionner dans différents environnements, il existe beaucoup de paramètres de configuration. Heureusement, la plupart ont des valeurs par défaut appropriées et la distribution d'Hibernate contient un exemple de fichier `hibernate.properties` dans le répertoire `etc/` qui montre les différentes options. Vous n'avez qu'à placer ce fichier dans votre classpath et à l'adapter.

3.1. Configuration par programmation

Une instance de `org.hibernate.cfg.Configuration` représente un ensemble de mappings des classes Java d'une application vers la base de données SQL. La `Configuration` est utilisée pour construire un objet (immuable) `SessionFactory`. Les mappings sont constitués d'un ensemble de fichiers de mapping XML.

Vous pouvez obtenir une instance de `Configuration` en l'instanciant directement et en spécifiant la liste des documents XML de mapping. Si les fichiers de mapping sont dans le classpath, vous pouvez le faire à l'aide de la méthode `addResource()` :

```
Configuration cfg = new Configuration()
    .addResource("Item.hbm.xml")
    .addResource("Bid.hbm.xml");
```

Une alternative (parfois meilleure) est de spécifier les classes mappées et de laisser Hibernate trouver les documents de mapping pour vous :

```
Configuration cfg = new Configuration()
    .addClass(org.hibernate.auction.Item.class)
    .addClass(org.hibernate.auction.Bid.class);
```

Hibernate va rechercher les fichiers de mappings `/org/hibernate/auction/Item.hbm.xml` et `/org/hibernate/auction/Bid.hbm.xml` dans le classpath. Cette approche élimine les noms de fichiers en dur.

Une `Configuration` vous permet également de préciser des propriétés de configuration :

```
Configuration cfg = new Configuration()
    .addClass(org.hibernate.auction.Item.class)
    .addClass(org.hibernate.auction.Bid.class)
    .setProperty("hibernate.dialect", "org.hibernate.dialect.MySQLInnoDBDialect")
    .setProperty("hibernate.connection.datasource", "java:comp/env/jdbc/test")
    .setProperty("hibernate.order_updates", "true");
```

Ce n'est pas le seul moyen de passer des propriétés de configuration à Hibernate. Les différentes options sont :

1. Passer une instance de `java.util.Properties` à `Configuration.setProperties()`.
2. Placer `hibernate.properties` dans un répertoire racine du classpath
3. Positionner les propriétés `System` en utilisant `java -Dproperty=value`.
4. Inclure des éléments `<property>` dans le fichier `hibernate.cfg.xml` (voir plus loin).

L'utilisation d'`hibernate.properties` est l'approche la plus simple si vous voulez démarrer rapidement

La `Configuration` est un objet de démarrage qui sera supprimé une fois qu'une `SessionFactory` aura été créée.

3.2. Obtenir une SessionFactory

Une fois que tous les mappings ont été parsés par la `Configuration`, l'application doit obtenir une fabrique d'instances de `Session`. Cette fabrique sera partagée entre tous les threads de l'application :

```
SessionFactory sessions = cfg.buildSessionFactory();
```

Hibernate permet à votre application d'instancier plus d'une `SessionFactory`. Cela est pratique lorsque vous utilisez plus d'une base de données.

3.3. Connexions JDBC

Habituellement, vous voulez que la `SessionFactory` crée les connexions JDBC et les mette dans un pool pour vous. Si vous suivez cette approche, ouvrir une `Session` est aussi simple que :

```
Session session = sessions.openSession(); // open a new Session
```

Dès que vous ferez quelquechose qui requiert un accès à la base de données, une connexion JDBC sera récupérée dans le pool.

Pour faire cela, il faut passer les propriétés de la connexion JDBC à Hibernate. Tous les noms des propriétés Hibernate et leur signification sont définies dans la classe `org.hibernate.cfg.Environment`. Nous allons maintenant décrire les paramètres de configuration des connexions JDBC les plus importants.

Hibernate obtiendra des connexions (et les mettra dans un pool) en utilisant `java.sql.DriverManager` si vous positionnez les paramètres de la manière suivante :

Tableau 3.1. Propriétés JDBC d'Hibernate

Nom de la propriété	Fonction
`hibernate.connection.driver_class`	*Classe du driver jdbc*
`hibernate.connection.url`	*URL jdbc*
`hibernate.connection.username`	*utilisateur de la base de données*
`hibernate.connection.password`	*mot de passe de la base de données*
`hibernate.connection.pool_size`	*nombre maximum de connexions dans le pool*

L'algorithme natif de pool de connexions d'Hibernate est plutôt rudimentaire. Il a été fait dans le but de vous aider à démarrer et *n'est pas prévu pour un système en production* ou même pour un test de peformance. Utilisez plutôt un pool tiers pour de meilleures performances et une meilleure stabilité : pour cela, remplacez la propriété `hibernate.connection.pool_size` avec les propriétés spécifique au pool de connexions que vous avez choisi. Cela désactivera le pool de connexions interne d'Hibernate. Vous pouvez par exemple utiliser C3P0.

C3P0 est un pool de connexions JDBC open source distribué avec Hibernate dans le répertoire `lib`. Hibernate utilisera son provider `C3P0ConnectionProvider` pour le pool de connexions si vous positionnez les propriétés `hibernate.c3p0.*`. Si vous voulez utiliser Proxool, référez vous au groupe de propriétés d'`hibernate.properties` correspondant et regardez sur le site web d'Hibernate pour plus d'informations.

Voici un exemple de fichier `hibernate.properties` pour C3P0:

```
hibernate.connection.driver_class = org.postgresql.Driver
hibernate.connection.url = jdbc:postgresql://localhost/mydatabase
hibernate.connection.username = myuser
hibernate.connection.password = secret
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=1800
hibernate.c3p0.max_statement=50
hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
```

Dans le cadre de l'utilisation au sein d'un serveur d'applications, vous devriez quasiment toujours configurer Hibernate pour qu'il obtienne ses connexions de la `DataSource` du serveur d'application enregistrée dans le JNDI. Pour cela vous devrez définir au moins une des propriétés suivantes :

Tableau 3.2. Propriété d'une Datasource Hibernate

Nom d'une propriété	fonction
`hibernate.connection.datasource`	*Nom JNDI de la datasource*
`hibernate.jndi.url`	*URL du fournisseur JNDI* (optionnelle)
`hibernate.jndi.class`	*Classe de l'`InitialContextFactory` du JNDI* (optionnelle)
`hibernate.connection.username`	*utilisateur de la base de données* (optionnelle)
`hibernate.connection.password`	*mot de passe de la base de données* (optionnelle)

Voici un exemple de fichier `hibernate.properties` pour l'utilisation d'une datasource JNDI fournie par un serveur d'applications :

```
hibernate.connection.datasource = java:/comp/env/jdbc/test
hibernate.transaction.factory_class = \
    org.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class = \
    org.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
```

Les connexions JDBC obtenues à partir d'une datasource JNDI participeront automatiquement aux transactions gérées par le conteneur du serveur d'applications.

Des propriétés supplémentaires de connexion peuvent être passées en préfixant le nom de la propriété par "`hibernate.connnection`". Par exemple, vous pouvez spécifier un jeu de caractères en utilisant `hibernate.connection.charSet`.

Vous pouvez fournir votre propre stratégie d'obtention des connexions JDBC en implémentant l'interface `org.hibernate.connection.ConnectionProvider`. Vous pouvez sélectionner une implémentation spécifique en positionnant `hibernate.connection.provider_class`.

3.4. Propriétés de configuration optionnelles

Il y a un certain nombre d'autres propriétés qui contrôlent le fonctionnement d'Hibernate à l'exécution. Toutes sont optionnelles et ont comme valeurs par défaut des valeurs "raisonnables" pour un fonctionnement nominal.

Attention : Certaines de ces propriétés sont uniquement de niveau System. Les propriétés de niveau System ne

peuvent être positionnées que via la ligne de commande (`java -Dproperty=value`) ou être définies dans `hibernate.properties`. Elle *ne peuvent pas* l'être via une des autres techniques décrites ci-dessus.

Tableau 3.3. Propriétés de configuration d'Hibernate

Nom de la propriété	Fonction
`hibernate.dialect`	Le nom de la classe du `Dialect` Hibernate. qui permet à Hibernate de générer du SQL optimisé pour une base de données relationnelle particulière. *ex.* `nom.complet.de.ma.classe.de.Dialect`
`hibernate.show_sql`	Ecrit toutes les requêtes SQL sur la console. Il s'agit d'une alternative au positionnement de la catégorie de log `org.hibernate.SQL` au niveau `debug`. *ex.* `true` | `false`
`hibernate.format_sql`	Formate et indente le sql dans la console et dans le log *ex.* `true` | `false`
`hibernate.default_schema`	Positionne dans le SQL généré un schéma/tablespace par défaut pour les noms de table ne l'ayant pas surchargé. *ex.* `MON_SCHEMA`
`hibernate.default_catalog`	Qualifie les noms de tables non qualifiées avec ce catalogue dans le SQL généré. *ex.* `CATALOG_NAME`
`hibernate.session_factory_name`	La `SessionFactory` sera automatiquement liée à ce nom dans le JNDI après sa création. *ex.* `jndi/nom/hierarchique`
`hibernate.max_fetch_depth`	Définit la profondeur maximale d'un arbre de chargement par jointures ouvertes pour les associations à cardinalité unitaire (un-à-un, plusieurs-à-un). Un `0` désactive le chargement par jointure ouverte. *ex.* valeurs recommandées entre `0` et `3`
`hibernate.default_batch_fetch_size`	Définit une taille par défaut pour le chargement par lot des associations *ex.* Valeurs recommandées : `4`, `8`, `16`
`hibernate.default_entity_mode`	Définit un mode de représentation par défaut des entités pour toutes les sessions ouvertes depuis cette `SessionFactory` `dynamic-map`, `dom4j`, `pojo`

Nom de la propriété	Fonction
`hibernate.order_updates`	Force Hibernate à trier les updates SQL par la valeur de la clé primaire des éléments qui sont mis à jour. Cela permet de limiter les deadlocks de transaction dans les systèmes hautement concurents. *ex.* `true` | `false`
`hibernate.generate_statistics`	Si activé, Hibernate va collecter des statistiques utiles pour le réglage des performances. *ex.* `true` | `false`
`hibernate.use_identifer_rollback`	Si activé, les propriétés correspondant à l'identifiant des objets vont être remises aux valeurs par défaut lorsque les objets seront supprimés. *ex.* `true` | `false`
`hibernate.use_sql_comments`	Si activé, Hibernate va générer des commentaires à l'intérieur des requêtes SQL pour faciliter le debogage., par défaut à `false`. *ex.* `true` | `false`

Tableau 3.4. Propriétés Hibernate liées à JDBC et aux connexions

Nom de la propriété	Fonction
`hibernate.jdbc.fetch_size`	Une valeur non nulle détermine la taille de chargement des statements JDBC (appelle `Statement.setFetchSize()`).
`hibernate.jdbc.batch_size`	Une valeur non nulle active l'utilisation par Hibernate des mises à jour par batch de JDBC2. *ex.* les valeurs recommandées entre `5` et `30`
`hibernate.jdbc.batch_versioned_data`	Paramétrez cette propriété à `true` si votre pilote JDBC retourne des row counts corrects depuis `executeBatch()` (il est souvent approprié d'activer cette option). Hibernate utilisera alors le "batched DML" pour versionner automatiquement les données. Par défaut = `false`. *eg.* `true` | `false`
`hibernate.jdbc.factory_class`	Sélectionne un `Batcher` personnalisé. La plupart des applications n'auront pas besoin de cette propriété de configuration *ex.* `classname.of.Batcher`
`hibernate.jdbc.use_scrollable_resultset`	Active l'utilisation par Hibernate des resultsets scrollables de JDBC2. Cette propriété est seulement nécessaire lorsque l'on utilise une connexion JDBC

Nom de la propriété	Fonction
	fournie par l'utilisateur. Autrement, Hibernate utilise les métadonnées de la connexion. *ex.* `true` | `false`
`hibernate.jdbc.use_streams_for_binary`	Utilise des flux lorsque l'on écrit/lit des types `binary` ou `serializable` vers et à partir de JDBC (propriété de niveau système). *ex.* `true` | `false`
`hibernate.jdbc.use_get_generated_keys`	Active l'utilisation de `PreparedStatement.getGeneratedKeys()` de JDBC3 pour récupérer nativement les clés générées après insertion. Nécessite un pilote JDBC3+, le mettre à false si votre pilote a des problèmes avec les générateurs d'identifiant Hibernate. Par défaut, essaie de déterminer les possibilités du pilote en utilisant les meta données de connexion. *eg.* `true|false`
`hibernate.connection.provider_class`	Le nom de la classe d'un `ConnectionProvider` personnalisé qui fournit des connexions JDBC à Hibernate *ex.* `classname.of.ConnectionProvider`
`hibernate.connection.isolation`	Définit le niveau d'isolation des transactions JDBC. Regardez `java.sql.Connection` pour connaître le sens des différentes valeurs mais notez également que la plupart des bases de données ne supportent pas tous les niveaux d'isolation. *ex.* `1, 2, 4, 8`
`hibernate.connection.autocommit`	Active le mode de commit automatique (autocommit) pour les connexions JDBC du pool (non recommandé). *ex.* `true` | `false`
`hibernate.connection.release_mode`	Spécifie à quel moment Hibernate doit relacher les connexion JDBC. Par défaut une connexion JDBC est conservée jusqu'à ce que la session soit explicitement fermée ou déconnectée. Pour une source de données JTA d'un serveur d'application, vous devriez utiliser `after_statement` pour libérer les connexions de manière plus agressive après chaque appel JDBC. Pour une connexion non JTA, il est souvent préférable de libérer la connexion à la fin de chaque transaction en utilisant `after_transaction`. `auto` choisira `after_statement` pour des transactions JTA et CMT et `after_transaction` pour des transactions JDBC.

Nom de la propriété	Fonction
	ex. `on_close` (default) | `after_transaction` | `after_statement` | `auto`
`hibernate.connection.<propertyName>`	Passe la propriété JDBC`propertyName` à `DriverManager.getConnection()`.
`hibernate.jndi.<propertyName>`	Passe la propriété `propertyName` à l'`InitialContextFactory` de JNDI.

Tableau 3.5. Propriétés du Cache d'Hibernate

Nom de la propriété	Fonction
`hibernate.cache.provider_class`	Le nom de classe d'un `CacheProvider` spécifique. *ex.* `nom.de.classe.du.CacheProvider`
`hibernate.cache.use_minimal_puts`	Optimise le cache de second niveau en minimisant les écritures, au prix de plus de lectures. Ce paramètre est surtout utile pour les caches en cluster et est activé par défaut dans hibernate3 pour les implémentations de cache en cluster. *ex.* `true|false`
`hibernate.cache.use_query_cache`	Activer le cache de requête, les requêtes individuelles doivent tout de même être déclarées comme pouvant être mise en cache. *ex.* `true|false`
`hibernate.cache.use_second_level_cache`	Peut être utilisé pour désactiver complètement le cache de second niveau qui est activé par défaut pour les classes qui spécifient un élément `<cache>` dans leur mapping. *ex.* `true|false`
`hibernate.cache.query_cache_factory`	Le nom de classe d'une interface `QueryCacheFactory`, par défaut = built-in `StandardQueryCacheFactory`. *ex.* `nom.de.la.classe.de.QueryCacheFactory`
`hibernate.cache.region_prefix`	Un préfixe à utiliser pour le nom des régions du cache de second niveau. *ex.* `prefix`
`hibernate.cache.use_structured_entries`	Force Hibernate à stocker les données dans le cache de second niveau dans un format plus adapté à la visualisation par un humain. *ex.* `true|false`

Tableau 3.6. Propriétés des transactions Hibernate

Nom de la propriété	Fonction
`hibernate.transaction.factory_class`	Le nom de classe d'une `TransactionFactory` qui sera utilisée par l'API `Transaction` d'Hibernate (la valeur par défaut est `JDBCTransactionFactory`). *ex.* `nom.de.classe.d.une.TransactionFactory`
`jta.UserTransaction`	Le nom JNDI utilisé par la `JTATransactionFactory` pour obtenir la `UserTransaction` JTA du serveur d'applications. *eg.* `jndi/nom/compose`
`hibernate.transaction.manager_lookup_class`	Le nom de la classe du `TransactionManagerLookup` - requis lorsque le cache de niveau JVM est activé ou lorsque l'on utilise un générateur hilo dans un environnement JTA. *ex.* `nom.de.classe.du.TransactionManagerLookup`
`hibernate.transaction.flush_before_completion`	Si activé, la session sera automatiquement vidée durant la phase qui précède la fin de la transaction (before completion). La gestion automatique de contexte fourni par Hibernate est recommandée, voir Section 2.5, « Sessions Contextuelles ». *ex.* `true` | `false`
`hibernate.transaction.auto_close_session`	Si activé, la session sera automatiquement fermé pendant la phase qui suit la fin de la transaction (after completion). La gestion automatique de contexte fourni par Hibernate est recommandée, voir *ex.* `true` | `false`

Tableau 3.7. Propriétés diverses

Nom de la propriété	Fonction
`hibernate.current_session_context_class`	Fournit une stratégie particulière pour contextualiser la `Session` courante. Voir Section 2.5, « Sessions Contextuelles » pour plus d'informations sur les stratégies fournies. *eg.* `jta` | `thread` | `managed` | `custom.Class`

`hibernate.query.factory_class`	Choisi l'implémentation du parseur de requête
	
	ex.
	`org.hibernate.hql.ast.ASTQueryTranslatorFactory`
	ou
	`org.hibernate.hql.classic.ClassicQueryTranslatorFactory`

Nom de la propriété	Fonction
`hibernate.query.substitutions`	Lien entre les tokens de requêtes Hibernate et les tokens SQL (les tokens peuvent être des fonctions ou des noms littéraux par exemple). *ex.* `hqlLiteral=SQL_LITERAL, hqlFunction=SQLFUNC`
`hibernate.hbm2ddl.auto`	Valide ou exporte automatiquement le schéma DDL vers la base de données lorsque la `SessionFactory` est créée. La valeur `create-drop` permet de supprimer le schéma de base de données lorsque la `SessionFactory` est fermée explicitement. *ex.* `validate` | `update` | `create` | `create-drop`
`hibernate.cglib.use_reflection_optimizer`	Active l'utilisation de CGLIB à la place de la réflexion à l'exécution (Propriété de niveau système). La réflexion peut parfois être utile pour résoudre des problèmes. Notez qu'Hibernate a tout de même toujours besoin de CGLIB même si l'optimiseur est désactivé. Cette optimisation ne peut être définie que dans le fichier `hibernate.cfg.xml`. *ex.* `true` | `false`

3.4.1. Dialectes SQL

Vous devriez toujours positionner la propriété `hibernate.dialect` à la sous-classe de `org.hibernate.dialect.Dialect` appropriée à votre base de données. Si vous spécifiez un dialecte, Hibernate utilisera des valeurs adaptées pour certaines autres propriétés listées ci-dessus, vous évitant l'effort de le faire à la main.

Tableau 3.8. Dialectes SQL d'Hibernate (`hibernate.dialect`)

SGBD	Dialecte
DB2	`org.hibernate.dialect.DB2Dialect`
DB2 AS/400	`org.hibernate.dialect.DB2400Dialect`
DB2 OS390	`org.hibernate.dialect.DB2390Dialect`
PostgreSQL	`org.hibernate.dialect.PostgreSQLDialect`
MySQL	`org.hibernate.dialect.MySQLDialect`
MySQL with InnoDB	`org.hibernate.dialect.MySQLInnoDBDialect`
MySQL with MyISAM	`org.hibernate.dialect.MySQLMyISAMDialect`
Oracle (any version)	`org.hibernate.dialect.OracleDialect`
Oracle 9i/10g	`org.hibernate.dialect.Oracle9Dialect`
Sybase	`org.hibernate.dialect.SybaseDialect`

SGBD	Dialecte
Sybase Anywhere	`org.hibernate.dialect.SybaseAnywhereDialect`
Microsoft SQL Server	`org.hibernate.dialect.SQLServerDialect`
SAP DB	`org.hibernate.dialect.SAPDBDialect`
Informix	`org.hibernate.dialect.InformixDialect`
HypersonicSQL	`org.hibernate.dialect.HSQLDialect`
Ingres	`org.hibernate.dialect.IngresDialect`
Progress	`org.hibernate.dialect.ProgressDialect`
Mckoi SQL	`org.hibernate.dialect.MckoiDialect`
Interbase	`org.hibernate.dialect.InterbaseDialect`
Pointbase	`org.hibernate.dialect.PointbaseDialect`
FrontBase	`org.hibernate.dialect.FrontbaseDialect`
Firebird	`org.hibernate.dialect.FirebirdDialect`

3.4.2. Chargement par Jointure Ouverte

Si votre base de données supporte les outer joins de type ANSI, Oracle ou Sybase, *le chargement par jointure ouverte* devrait améliorer les performances en limitant le nombre d'aller-retour avec la base de données (la base de données effectuant donc potentiellement plus de travail). Le chargement par jointure ouverte permet à un graphe entier d'objets connectés par une relation plusieurs-à-un, un-à-plusieurs ou un-à-un d'être chargé en un seul `SELECT` SQL.

Le chargement par jointure ouverte peut être désactiver *globalement* en mettant la propriété `hibernate.max_fetch_depth` à `0`. Une valeur de `1` ou plus active le chargement par jointure ouverte pour les associatiosn un-à-un et plusieurs-à-un qui ont été mappée avec `fetch="join"`.

Reportez vous à Section 19.1, « Stratégies de chargement » pour plus d'information.

3.4.3. Flux binaires

Oracle limite la taille d'un tableau de `byte` qui peuvent être passées à et vers son pilote JDBC. Si vous souhaitez utiliser des instances larges de type `binary` ou `serializable`, vous devez activer la propriété `hibernate.jdbc.use_streams_for_binary`. *C'est une fonctionalité de niveau système uniquement.*

3.4.4. Cache de second niveau et cache de requêtes

Les propriétés préfixées par `hibernate.cache` vous permettent d'utiliser un système de cache de second niveau. Ce cache peut avoir une portée dans le processus ou même être utilisable dans un système distribué. Référez vous au chapitre Section 19.2, « Le cache de second niveau » pour plus de détails.

3.4.5. Substitution dans le langage de requêtage

Vous pouvez définir de nouveaux tokens dans les requêtes Hibernate en utilisant la propriété

`hibernate.query.substitutions`. Par exemple :

```
hibernate.query.substitutions vrai=1, faux=0
```

remplacerait les tokens `vrai` et `faux` par des entiers dans le SQL généré.

```
hibernate.query.substitutions toLowercase=LOWER
```

permettrait de renommer la fonction SQL `LOWER` en `toLowercase`

3.4.6. Statistiques Hibernate

Si vous activez `hibernate.generate_statistics`, Hibernate va fournir un certains nombre de métriques utiles pour régler les performances d'une application qui tourne via `SessionFactory.getStatistics()`. Hibernate peut aussi être configuré pour exposer ces statistiques via JMX. Lisez les Javadoc des interfaces dans le package `org.hibernate.stats` pour plus d'informations.

3.5. Tracer

Hibernate trace divers évènements en utilisant Apache commons-logging.

Le service commons-logging délèguera directement à Apache Log4j (si vous incluez `log4j.jar` dans votre classpath) ou le système de trace du JDK 1.4 (si vous tournez sous le JDK 1.4 et supérieur). Vous pouvez télécharger Log4j à partir de `http://jakarta.apache.org`. Pour utiliser Log4j, vous devrez placer dans votre classpath un fichier `log4j.properties`. Un exemple de fichier est distribué avec Hibernate dans le répertoire `src/`.

Nous vous recommandons fortement de vous familiariser avec les messages des traces d'Hibernate. Beaucoup de soins a été apporté pour donner le plus de détails possible sans les rendre illisibles. C'est un outil essentiel en cas de soucis. Les catégories de trace les plus intéressantes sont les suivantes :

Tableau 3.9. Catégories de trace d'Hibernate

Catégorie	Fonction
`org.hibernate.SQL`	Trace toutes les requêts SQL de type DML (gestion des données) qui sont exécutées
`org.hibernate.type`	Trace tous les paramètres JDBC
`org.hibernate.tool.hbm2ddl`	Trace toutes les requêts SQL de type DDL (gestion de la structure de la base) qui sont exécutées
`org.hibernate.pretty`	Trace l'état de toutes les entités (20 entités maximum) qui sont associées avec la session hibernate au moment du flush
`org.hibernate.cache`	Trace toute l'activité du cache de second niveau
`org.hibernate.transaction`	Trace toute l'activité relative aux transactions
`org.hibernate.jdbc`	Trace toute acquisition de ressource JDBC
`org.hibernate.hql.ast.AST`	Trace l'arbre syntaxique des requêtes HQL et SQL durant l'analyse syntaxique des requêtes
`org.hibernate.secure`	Trace toutes les demandes d'autorisation JAAS

Catégorie	Fonction
org.hibernate	Trace tout (beaucoupe d'informations, mais très utile pour résoudre les problèmes).

Lorsque vous développez des applications avec Hibernate, vous devriez quasiment toujours travailler avec le niveau debug activé pour la catégorie org.hibernate.SQL, ou sinon avec la propriété hibernate.show_sql activée.

3.6. Implémenter une NamingStrategy

L'interface org.hibernate.cfg.NamingStrategy vous permet de spécifier une "stratégie de nommage" des objets et éléments de la base de données.

Vous pouvez fournir des règles pour automatiquement générer les identifiants de base de données à partir des identifiants Java, ou transformer une colonne ou table "logique" donnée dans le fichier de mapping en une colonne ou table "physique". Cette fonctionnalité aide à réduire la verbosité de documents de mapping, en éliminant le bruit répétitif (les préfixes TBL_ par exemple). La stratégie par défaut utilisée par Hibernate est minimale.

Vous pouvez définir une stratégie différente en appelant Configuration.setNamingStrategy() avant d'ajouter des mappings :

```
SessionFactory sf = new Configuration()
    .setNamingStrategy(ImprovedNamingStrategy.INSTANCE)
    .addFile("Item.hbm.xml")
    .addFile("Bid.hbm.xml")
    .buildSessionFactory();
```

net.sf.hibernate.cfg.ImprovedNamingStrategy est une stratégie fournie qui peut être utile comme point de départ de quelques applications.

3.7. Fichier de configuration XML

Une approche alternative est de spécifier toute la configuration dans un fichier nommé hibernate.cfg.xml. Ce fichier peut être utilisé à la place du fichier hibernate.properties, voire même peut servir à surcharger les propriétés si les deux fichiers sont présents.

Le fichier de configuration XML doit par défaut se placer à la racine du CLASSPATH. En voici un exemple :

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <!-- a SessionFactory instance listed as /jndi/name -->
    <session-factory
        name="java:hibernate/SessionFactory">

        <!-- properties -->
        <property name="connection.datasource">java:/comp/env/jdbc/MyDB</property>
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">false</property>
        <property name="transaction.factory_class">
```

```
            org.hibernate.transaction.JTATransactionFactory
        </property>
        <property name="jta.UserTransaction">java:comp/UserTransaction</property>

        <!-- mapping files -->
        <mapping resource="org/hibernate/auction/Item.hbm.xml"/>
        <mapping resource="org/hibernate/auction/Bid.hbm.xml"/>

        <!-- cache settings -->
        <class-cache class="org.hibernate.auction.Item" usage="read-write"/>
        <class-cache class="org.hibernate.auction.Bid" usage="read-only"/>
        <collection-cache collection="org.hibernate.auction.Item.bids" usage="read-write"/>

    </session-factory>

</hibernate-configuration>
```

Commme vous pouvez le voir, l'avantage de cette approche est l'externalisation des noms des fichiers de mapping de la configuration. Le fichier `hibernate.cfg.xml` est également plus pratique quand on commence à régler le cache d'Hibernate. Notez que vous pouvez choisir entre utiliser `hibernate.properties` ou `hibernate.cfg.xml`, les deux sont équivalents, sauf en ce qui concerne les bénéfices de l'utilisation de la syntaxe XML mentionnés ci-dessus.

Avec la configuration XML, démarrer Hibernate devient donc aussi simple que ceci :

```
SessionFactory sf = new Configuration().configure().buildSessionFactory();
```

3.8. Intégration à un serveur d'application J2EE

Hibernate possède les points suivants d'intégration à l'infrastructure J2EE :

- *Source de données gérée par le conteneur* : Hibernate peut utiliser des connexions JDBC gérées par le conteneur et fournie par l'intermédiaire de JNDI. Souvent, un `TransactionManager` compatible JTA et un `ResourceManager` s'occupent de la gestion des transactions (CMT). Ils sont particulièrement prévus pour pouvoir gérer des transactions distribuées sur plusieurs sources de données. Vous pouvez bien sûr également définir vos limites de transaction dans votre programme (BMT) ou vous pouvez sinon aussi utiliser l'API optionnelle `Transaction` d'Hibernate qui vous garantira la portabilité de votre code entre plusieurs serveurs d'application.

- *Association JNDI automatique*: Hibernate peut associer sa `SessionFactory` à JNDI après le démarrage.

- *Association de la Session à JTA:* La `Session` Hibernate peut être associée automatiquement à une transaction JTA si vous utilisez les EJBs. Vous avez juste à récupérer la `SessionFactory` depuis JNDI et à récupérer la `Session` courante. Hibernate s'occupe de vider et fermer la `Session` lorsque le transaction JTA se termine. La démarcation des transactions se fait de manière déclarative dans les descripteurs de déploiement.

- *Déploiement JMX :*Si vous avez un serveur d'application compatible JMX (JBoss AS par exemple), vous pouvez choisir de déployer Hibernate en temps que MBean géré par le serveur. Cela vous évite de coder la ligne de démarrage qui permet de construire la `SessionFactory` depuis la `Configuration`. Le conteneur va démarrer votre `HibernateService`, et va idéalement s'occuper des dépendances entre les services (la source de données doit être disponible avant qu'Hibernate ne démarre, etc).

En fonction de votre environnement, vous devrez peut être mettre l'option de configuration `hibernate.connection.aggressive_release` à vrai si le serveur d'application affiche des exceptions de type "connection containment".

3.8.1. Configuration de la stratégie transactionnelle

L'API de la `Session` Hibernate est indépendante de tout système de démarcation des transactions qui peut être présent dans votre architecture. Si vous laissez Hibernate utiliser l'API JDBC directement via un pool de connexion, vous devrez commencer et terminer vos transactions en utilisant l'API JDBC. Si votre application tourne à l'intérieur d'un serveur d'application J2EE, vous voudrez peut être utiliser les transactions gérées par les beans (BMT) et appeller l'API JTA et `UserTransaction` lorsque cela est nécessaire.

Pour conserver votre code portable entre ces deux environnements (et d'autres éventuels) nous vous recommandons d'utiliser l'API optionnelle `Transaction` d'Hibernate, qui va encapsuler et masquer le système de transaction sous-jacent. Pour cela, vous devez préciser une classe de fabrique d'instances de `Transaction` en positionnant la propriété `hibernate.transaction.factory_class`.

Il existe trois choix standards (fournis) :

`net.sf.hibernate.transaction.JDBCTransactionFactory`
: délègue aux transactions de la base de données (JDBC). Valeur par défaut.

`org.hibernate.transaction.JTATransactionFactory`
: délègue à CMT si une transaction existante est sous ce contexte (ex: méthode d'un EJB session), sinon une nouvelle transaction est entamée et une transaction gérée par le bean est utilisée.

`org.hibernate.transaction.CMTTransactionFactory`
: délègue à aux transactions JTA gérées par le conteneur

Vous pouvez également définir votre propre stratégie transactionnelle (pour un service de transaction CORBA par exemple).

Certaines fonctionnalités d'Hibernate (i.e. le cache de second niveau, l'association automatique des Session à JTA, etc.) nécessitent l'accès au `TransactionManager` JTA dans un environnement "managé". Dans un serveur d'application, vous devez indiquer comment Hibernate peut obtenir une référence vers le `TransactionManager`, car J2EE ne fournit pas un seul mécanisme standard.

Tableau 3.10. TransactionManagers JTA

Fabrique de Transaction	Serveur d'application
`org.hibernate.transaction.JBossTransactionManagerLookup`	JBoss
`org.hibernate.transaction.WeblogicTransactionManagerLookup`	Weblogic
`org.hibernate.transaction.WebSphereTransactionManagerLookup`	WebSphere
`org.hibernate.transaction.WebSphereExtendedJTATransactionLookup`	WebSphere 6
`org.hibernate.transaction.OrionTransactionManagerLookup`	Orion
`org.hibernate.transaction.ResinTransactionManagerLookup`	Resin
`org.hibernate.transaction.JOTMTransactionManagerLookup`	JOTM
`org.hibernate.transaction.JOnASTransactionManagerLookup`	JOnAS

Fabrique de Transaction	Serveur d'application
`org.hibernate.transaction.JRun4TransactionManagerLookup`	JRun4
`org.hibernate.transaction.BESTransactionManagerLookup`	Borland ES

3.8.2. `SessionFactory` associée au JNDI

Une `SessionFactory` Hibernate associée au JNDI peut simplifier l'accès à la fabrique et donc la création de nouvelles `Sessions`. Notez que cela n'est pas lié avec les `Datasource` associées au JNDI, elles utilisent juste le même registre.

Si vous désirez associer la `SessionFactory` à un nom JNDI, spécifiez un nom (ex. `java:hibernate/SessionFactory`) en utilisant la propriété `hibernate.session_factory_name`. Si cette propriété est omise, la `SessionFactory` ne sera pas associée au JNDI (c'est particulièrement pratique dans les environnements ayant une implémentation de JNDI en lecture seule, comme c'est le cas pour Tomcat).

Lorsqu'il associe la `SessionFactory` au JNDI, Hibernate utilisera les valeurs de `hibernate.jndi.url`, `hibernate.jndi.class` pour instancier un contexte d'initialisation. S'ils ne sont pas spécifiés, l'`InitialContext` par défaut sera utilisé.

Hibernate va automatiquement placer la `SessionFactory` dans JNDI après avoir appelé `cfg.buildSessionFactory()`. Cela signifie que vous devez avoir cet appel dans un code de démarrage (ou dans une classe utilitaire) dans votre application sauf si vous utilisez le déploiement JMX avec le service `HibernateService` présenté plus tard dans ce document.

Si vous utilisez `SessionFactory` JNDI, un EJB ou n'importe quelle autre classe peut obtenir la `SessionFactory` en utilisant un lookup JNDI.

Nous recommandons que vous liiez la `SessionFactory` à JNDI dans les environnements managés et que vous utilisiez un singleton `static` si ce n'est pas le cas. Pour isoler votre application de ces détails, nous vous recommandons aussi de masquer le code de lookup actuel pour une `SessionFactory` dans une classe helper, comme `HibernateUtil.getSessionFactory()`. Notez qu'une telle classe est aussi un moyen efficace de démarrer Hibernate—voir chapitre 1.

3.8.3. Association automatique de la Session à JTA

Le moyen le plus simple de gérer les `Sessions` et transactions est la gestion automatique de session "courante" offerte par Hibernate. Voir détail à Section 2.5, « Sessions Contextuelles ». En utilisant le contexte de session `"jta"` session context, s'il n'y a pas de `Session` associée à la transaction JTA courante, une session sera démarrée et associée à la transaction JTA courante la première fois que vous appelez `sessionFactory.getCurrentSession()`. Les `Sessions` obtenue via `getCurrentSession()` dans une contexte `"jta"` seront automatiquement flushées avant la validation de la transaction, fermées une fois la transaction complétée, et libéreront les connexions JDBC de manière aggressive après chaque statement. Ceci permet aux `Sessions` d'être gérées par le cycle de vie de la transaction JTA à la quelle est sont associées, laissant le code de l'utilisateur propre de ce type de gestion. Votre code peut soit utiliser JTA de manière programmatique via `UserTransaction`, ou (ce qui est recommandé pour la portabilité du code) utiliser l'API `Transaction` API pour marquer les limites. Si vous exécutez sous un conteneur EJB, la démarcation déclarative des transactions avec CMT est recommandée.

3.8.4. Déploiement JMX

La ligne `cfg.buildSessionFactory()` doit toujours être exécutée quelque part pour avoir une `SessionFactory` dans JNDI. Vous pouvez faire cela dans un bloc d'initialisation `static` (comme celui qui se trouve dans la classe `HibernateUtil`) ou vous pouvez déployer Hibernate en temps que *service managé*.

Hibernate est distribué avec `org.hibernate.jmx.HibernateService` pour le déploiement sur un serveur d'application avec le support de JMX comme JBoss AS. Le déploiement et la configuration sont spécifiques à chaque vendeur. Voici un fichier `jboss-service.xml` d'exemple pour JBoss 4.0.x:

```
<?xml version="1.0"?>
<server>

<mbean code="org.hibernate.jmx.HibernateService"
    name="jboss.jca:service=HibernateFactory,name=HibernateFactory">

    <!-- Required services -->
    <depends>jboss.jca:service=RARDeployer</depends>
    <depends>jboss.jca:service=LocalTxCM,name=HsqlDS</depends>

    <!-- Bind the Hibernate service to JNDI -->
    <attribute name="JndiName">java:/hibernate/SessionFactory</attribute>

    <!-- Datasource settings -->
    <attribute name="Datasource">java:HsqlDS</attribute>
    <attribute name="Dialect">org.hibernate.dialect.HSQLDialect</attribute>

    <!-- Transaction integration -->
    <attribute name="TransactionStrategy">
        org.hibernate.transaction.JTATransactionFactory</attribute>
    <attribute name="TransactionManagerLookupStrategy">
        org.hibernate.transaction.JBossTransactionManagerLookup</attribute>
    <attribute name="FlushBeforeCompletionEnabled">true</attribute>
    <attribute name="AutoCloseSessionEnabled">true</attribute>

    <!-- Fetching options -->
    <attribute name="MaximumFetchDepth">5</attribute>

    <!-- Second-level caching -->
    <attribute name="SecondLevelCacheEnabled">true</attribute>
    <attribute name="CacheProviderClass">org.hibernate.cache.EhCacheProvider</attribute>
    <attribute name="QueryCacheEnabled">true</attribute>

    <!-- Logging -->
    <attribute name="ShowSqlEnabled">true</attribute>

    <!-- Mapping files -->
    <attribute name="MapResources">auction/Item.hbm.xml,auction/Category.hbm.xml</attribute>

</mbean>

</server>
```

Ce fichier est déployé dans un répertoire `META-INF` et est packagé dans un fichier JAR avec l'extension `.sar` (service archive). Vous devez également packager Hibernate, les librairies tierces requises, vos classes persistantes compilées et vos fichiers de mapping dans la même archive. Vos beans entreprise (souvent des EJBs session) peuvent rester dans leur propre fichier JAR mais vous pouvez inclure ce fichier JAR dans le jar principal du service pour avoir une seule unité déployable à chaud. Vous pouvez consulter la documentation de JBoss AS pour plus d'information sur les services JMX et le déploiement des EJBs.

Chapitre 4. Classes persistantes

Les classes persistantes sont les classes d'une application qui implémentent les entités d'un problème métier (ex. Client et Commande dans une application de commerce électronique). Toutes les instances d'une classe persistante ne sont pas forcément dans l'état persistant - au lieu de cela, une instance peut être éphémère (NdT : transient) ou détachée.

Hibernate fonctionne de manière optimale lorsque ces classes suivent quelques règles simples, aussi connues comme le modèle de programmation Plain Old Java Object (POJO). Cependant, aucune de ces règles ne sont des besoins absolus. En effet, Hibernate3 suppose très peu de choses à propos de la nature de vos objets persistants. Vous pouvez exprimer un modèle de domaine par d'autres moyens : utiliser des arbres d'instances de `Map`, par exemple.

4.1. Un exemple simple de POJO

Toute bonne application Java nécessite une classe persistante représentant les félins.

```
package eg;
import java.util.Set;
import java.util.Date;

public class Cat {
    private Long id; // identifier

    private Date birthdate;
    private Color color;
    private char sex;
    private float weight;
    private int litterId;

    private Cat mother;
    private Set kittens = new HashSet();

    private void setId(Long id) {
        this.id=id;
    }
    public Long getId() {
        return id;
    }

    void setBirthdate(Date date) {
        birthdate = date;
    }
    public Date getBirthdate() {
        return birthdate;
    }

    void setWeight(float weight) {
        this.weight = weight;
    }
    public float getWeight() {
        return weight;
    }

    public Color getColor() {
        return color;
    }
    void setColor(Color color) {
        this.color = color;
    }

    void setSex(char sex) {
        this.sex=sex;
```

```
    }
    public char getSex() {
        return sex;
    }

    void setLitterId(int id) {
        this.litterId = id;
    }
    public int getLitterId() {
        return litterId;
    }

    void setMother(Cat mother) {
        this.mother = mother;
    }
    public Cat getMother() {
        return mother;
    }
    void setKittens(Set kittens) {
        this.kittens = kittens;
    }
    public Set getKittens() {
        return kittens;
    }

    // addKitten not needed by Hibernate
    public void addKitten(Cat kitten) {
        kitten.setMother(this);
    kitten.setLitterId( kittens.size() );
        kittens.add(kitten);
    }
}
```

Il y a quatre règles à suivre ici :

4.1.1. Implémenter un constructeur sans argument

`Cat` a un constructeur sans argument. Toutes les classes persistantes doivent avoir un constructeur par défaut (lequel peut ne pas être public) pour qu'Hibernate puissent les instancier en utilisant `Constructor.newInstance()`. Nous recommandons fortement d'avoir un constructeur par défaut avec au moins une visibilité *paquet* pour la génération du proxy à l'exécution dans Hibernate.

4.1.2. Fournir une propriété d'indentifiant (optionnel)

`Cat` possède une propriété appelée `id`. Cette propriété mappe la valeur de la colonne de clé primaire de la table d'une base de données.La propriété aurait pu s'appeler complètement autrement, et son type aurait pu être n'importe quel type primitif, n'importe quel "encapsuleur" de type primitif, `java.lang.String` ou `java.util.Date`. (Si votre base de données héritée possède des clés composites, elles peuvent être mappées en utilisant une classe définie par l'utilisateur et possédant les propriétés associées aux types de la clé composite - voir la section concernant les identifiants composites plus tard).

La propriété d'identifiant est strictement optionnelle. Vous pouver l'oublier et laisser Hibernate s'occuper des identifiants de l'objet en interne. Toutefois, nous ne le recommandons pas.

En fait, quelques fonctionnalités ne sont disponibles que pour les classes déclarant un identifiant de propriété :

- Les réattachements transitifs pour les objets détachés (mise à jour en cascade ou fusion en cascade) - voir Section 10.11, « Persistance transitive »
- `Session.saveOrUpdate()`
- `Session.merge()`

Nous recommandons que vous déclariez les propriétés d'identifiant de manière uniforme. Nous recommandons également que vous utilisiez un type nullable (ie. non primitif).

4.1.3. Favoriser les classes non finales (optionnel)

Une fonctionnalité clef d'Hibernate, les *proxies*, nécessitent que la classe persistente soit non finale ou qu'elle soit l'implémentation d'une interface qui déclare toutes les méthodes publiques.

Vous pouvez persister, grâce à Hibernate, les classes `final` qui n'implémentent pas d'interface, mais vous ne pourrez pas utiliser les proxies pour les chargements d'associations paresseuses - ce qui limitera vos possibilités d'ajustement des performances.

Vous devriez aussi éviter de déclarer des méthodes `public final` sur des classes non-finales. Si vous voulez utiliser une classe avec une méthode `public final`, vous devez explicitement désactiver les proxies en paramétrant `lazy="false"`.

4.1.4. Déclarer les accesseurs et mutateurs des attributs persistants (optionnel)

`Cat` déclare des mutateurs pour toutes ses champs persistants. Beaucoup d'autres solutions de mapping Objet/relationnel persistent directement les variables d'instance. Nous pensons qu'il est bien mieux de fournir une indirection entre le schéma relationnel et les structures de données internes de la classe. Par défaut, Hibernate persiste les propriétés suivant le style JavaBean, et reconnaît les noms de méthodes de la forme `getFoo`, `isFoo` et `setFoo`. Nous pouvons changer pour un accès direct aux champs pour des propriétés particulières, si besoin est.

Les propriétés *n'ont pas* à être déclarées publiques - Hibernate peut persister une propriété avec un paire de getter/setter de visibilité par défault, `protected` ou `private`.

4.2. Implémenter l'héritage

Une sous-classe doit également suivre la première et la seconde règle. Elle hérite sa propriété d'identifiant de `Cat`.

```
package eg;

public class DomesticCat extends Cat {
        private String name;

        public String getName() {
                return name;
        }
        protected void setName(String name) {
                this.name=name;
        }
}
```

4.3. Implémenter `equals()` et `hashCode()`

Vous devez surcharger les méthodes `equals()` et `hashCode()` si vous

- avez l'intention de mettre des instances de classes persistantes dans un `Set` (la manière recommandée pour

représenter des associations pluri-valuées) *et*
- avez l'intention d'utiliser le réattachement d'instances détachées

Hibernate garantit l'équivalence de l'identité persistante (ligne de base de données) et l'identité Java seulement à l'intérieur de la portée d'une session particulière. Donc dès que nous mélangeons des instances venant de différentes sessions, nous devons implémenter `equals()` et `hashCode()` si nous souhaitons avoir une sémantique correcte pour les `Sets`.

La manière la plus évidente est d'implémenter `equals()`/`hashCode()` en comparant la valeur de l'identifiant des deux objets. Si cette valeur est identique, les deux doivent représenter la même ligne de base de données, ils sont donc égaux (si les deux sont ajoutés à un `Set`, nous n'aurons qu'un seul élément dans le `Set`). Malheureusement, nous ne pouvons pas utiliser cette approche avec des identifiants générés ! Hibernate n'assignera de valeur d'identifiant qu'aux objets qui sont persistants, une instance nouvellement créée n'aura donc pas de valeur d'identifiant ! De plus, si une instance est non sauvegardée et actuellement dans un `Set`, le sauvegarder assignera une valeur d'identifiant à l'objet. Si `equals()` et `hashCode()` sont basées sur la valeur de l'identifiant, le code de hachage devrait changer, rompant le contrat du `Set`. Regardez sur le site web d'Hibernate pour une discussion complète de ce problème. Notez que ceci n'est pas un problème d'Hibernate, mais la sémantique normale de Java pour l'identité d'un objet et l'égalité.

Nous recommandons donc d'implémenter `equals()` et `hashCode()` en utilisant *l'égalité par clé métier*.L'égalité par clé métier signifie que la méthode `equals()` compare uniquement les propriétés qui forment une clé métier, une clé qui identifierait notre instance dans le monde réel (une clé candidate *naturelle*) :

```
public class Cat {

    ...
    public boolean equals(Object other) {
        if (this == other) return true;
        if ( !(other instanceof Cat) ) return false;

        final Cat cat = (Cat) other;

        if ( !cat.getLitterId().equals( getLitterId() ) ) return false;
        if ( !cat.getMother().equals( getMother() ) ) return false;

        return true;
    }

    public int hashCode() {
        int result;
        result = getMother().hashCode();
        result = 29 * result + getLitterId();
        return result;
    }

}
```

Notez qu'une clef métier ne doit pas être solide comme une clef primaire de base de données (voir Section 11.1.3, « L'identité des objets »). Les propriétés immuables ou uniques sont généralement de bonnes candidates pour une clef métier.

4.4. Modèles dynamiques

Notez que la fonctionnalités suivantes sont actuellement considérées comme expérimentales et peuvent changer dans un futur proche.

Les entités persistantes ne doivent pas nécessairement être représentées comme des classes POJO ou des objets JavaBean à l'exécution. Hibernate supporte aussi les modèles dynamiques (en utilisant des `Maps` de `Maps` à

l'exécution) et la représentation des entités comme des arbres DOM4J. Avec cette approche, vous n'écrivez pas de classes persistantes, seulement des fichiers de mapping.

Par défaut, Hibernate fonctionne en mode POJO normal. Vous pouvez paramétrer un mode de représentation d'entité par défaut pour une `SessionFactory` particulière en utilisant l'option de configuration `default_entity_mode` (voir Tableau 3.3, « Propriétés de configuration d'Hibernate »).

Les exemples suivants démontrent la représentation utilisant des `Maps`. D'abord, dans le fichier de mapping, un `entity-name` doit être déclaré au lieu (ou en plus) d'un nom de classe :

```
<hibernate-mapping>

    <class entity-name="Customer">

        <id name="id"
            type="long"
            column="ID">
            <generator class="sequence"/>
        </id>

        <property name="name"
            column="NAME"
            type="string"/>

        <property name="address"
            column="ADDRESS"
            type="string"/>

        <many-to-one name="organization"
            column="ORGANIZATION_ID"
            class="Organization"/>

        <bag name="orders"
            inverse="true"
            lazy="false"
            cascade="all">
            <key column="CUSTOMER_ID"/>
            <one-to-many class="Order"/>
        </bag>

    </class>

</hibernate-mapping>
```

Notez que même si des associations sont déclarées en utilisant des noms de classe cible, le type de cible d'une association peut aussi être une entité dynamique au lieu d'un POJO.

Après avoir configuré le mode d'entité par défaut à `dynamic-map` pour la `SessionFactory`, nous pouvons lors de l'exécution fonctionner avec des `Maps` de `Maps` :

```
Session s = openSession();
Transaction tx = s.beginTransaction();
Session s = openSession();

// Create a customer
Map david = new HashMap();
david.put("name", "David");

// Create an organization
Map foobar = new HashMap();
foobar.put("name", "Foobar Inc.");

// Link both
david.put("organization", foobar);

// Save both
s.save("Customer", david);
```

```
s.save("Organization", foobar);

tx.commit();
s.close();
```

Les avantages d'un mapping dynamique sont un gain de temps pour le prototypage sans la nécessité d'implémenter les classes d'entité. Pourtant, vous perdez la vérification du typage au moment de la compilation et aurez plus d'exceptions à gérer lors de l'exécution. Grâce au mapping d'Hibernate, le schéma de la base de données peut facilement être normalisé et solidifié, permettant de rajouter une implémentation propre du modèle de domaine plus tard.

Les modes de représentation d'une entité peut aussi être configuré par `Session` :

```
Session dynamicSession = pojoSession.getSession(EntityMode.MAP);

// Create a customer
Map david = new HashMap();
david.put("name", "David");
dynamicSession.save("Customer", david);
...
dynamicSession.flush();
dynamicSession.close()
...
// Continue on pojoSession
```

Veuillez noter que l'appel à `getSession()` en utilisant un `EntityMode` se fait sur l'API `Session`, pas `SessionFactory`. De cette manière, la nouvelle `Session` partage les connexions JDBC, transactions et autres informations de contexte sous-jacentes. Cela signifie que vous n'avez pas à appeler `flush()` et `close()` sur la `Session` secondaire, et laissez aussi la gestion de la transaction et de la connexion à l'unité de travail primaire.

Plus d'informations à propos de la représentation XML peuvent être trouvées dans Chapitre 18, *Mapping XML*.

4.5. Tuplizers

`org.hibernate.tuple.Tuplizer`, et ses sous-interfaces, sont responsables de la gestion d'une représentation particulière d'un morceau de données, en fonction du `org.hibernate.EntityMode` de réprésentation. Si un morceau donné de données est pensé comme une structure de données, alors un tuplizer est la chose qui sait comment créer une telle structure de données, comment extraire des valeurs et injecter des valeurs dans une telle structure de données. Par exemple, pour le mode d'entité POJO, le tuplizer correspondant sait comment créer le POJO à travers son constructeur et comment accéder aux propriétés du POJO utilisant les accesseurs de la propriété définie. Il y a deux types de Tuplizers haut niveau, représentés par les interfaces `org.hibernate.tuple.EntityTuplizer` et `org.hibernate.tuple.ComponentTuplizer`. Les `EntityTuplizers` sont responsables de la gestion des contrats mentionnés ci-dessus pour les entités, alors que les `ComponentTuplizers` s'occupent des composants.

Les utilisateurs peuvent aussi brancher leurs propres tuplizers. Peut-être vous est-il nécessaire qu'une implémentation de `java.util.Map` autre que `java.util.HashMap` soit utilisée dans le mode d'entité dynamic-map ; ou peut-être avez-vous besoin de définir une statégie de génération de proxy différente de celle utilisée par défaut. Les deux devraient être effectuées en définissant une implémentation de tuplizer utilisateur. Les définitions de tuplizers sont attachées au mapping de l'entité ou du composant qu'ils sont censés gérer. Retour à l'exemple de notre entité utilisateur :

```
<hibernate-mapping>
    <class entity-name="Customer">
        <!--
            Override the dynamic-map entity-mode
            tuplizer for the customer entity
```

```
        -->
        <tuplizer entity-mode="dynamic-map"
                class="CustomMapTuplizerImpl"/>

        <id name="id" type="long" column="ID">
            <generator class="sequence"/>
        </id>

        <!-- other properties -->
        ...
    </class>
</hibernate-mapping>


public class CustomMapTuplizerImpl
        extends org.hibernate.tuple.entity.DynamicMapEntityTuplizer {
    // override the buildInstantiator() method to plug in our custom map...
    protected final Instantiator buildInstantiator(
            org.hibernate.mapping.PersistentClass mappingInfo) {
        return new CustomMapInstantiator( mappingInfo );
    }

    private static final class CustomMapInstantiator
            extends org.hibernate.tuple.DynamicMapInstantitor {
        // override the generateMap() method to return our custom map...
        protected final Map generateMap() {
            return new CustomMap();
        }
    }
}
```

TODO: Document user-extension framework in the property and proxy packages

Chapitre 5. Mapping O/R basique

5.1. Déclaration de Mapping

Les mappings Objet/relationnel sont généralement définis dans un document XML. Le document de mapping est conçu pour être lisible et éditable à la main. Le langage de mapping est Java-centrique, c'est à dire que les mappings sont construits à partir des déclarations des classes persistantes et non des déclarations des tables.

Remarquez que même si beaucoup d'utilisateurs de Hibernate préfèrent écrire les fichiers de mappings à la main, plusieurs outils existent pour générer ce document, notamment XDoclet, Middlegen et AndroMDA.

Démarrons avec un exemple de mapping :

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
      "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="eg">

        <class name="Cat"
            table="cats"
            discriminator-value="C">

                <id name="id">
                        <generator class="native"/>
                </id>

                <discriminator column="subclass"
                     type="character"/>

                <property name="weight"/>

                <property name="birthdate"
                    type="date"
                    not-null="true"
                    update="false"/>

                <property name="color"
                    type="eg.types.ColorUserType"
                    not-null="true"
                    update="false"/>

                <property name="sex"
                    not-null="true"
                    update="false"/>

                <property name="litterId"
                    column="litterId"
                    update="false"/>

                <many-to-one name="mother"
                    column="mother_id"
                    update="false"/>

                <set name="kittens"
                    inverse="true"
                    order-by="litter_id">
                        <key column="mother_id"/>
                        <one-to-many class="Cat"/>
                </set>

                <subclass name="DomesticCat"
                    discriminator-value="D">
```

```
                        <property name="name"
                            type="string"/>

                </subclass>

        </class>

        <class name="Dog">
                <!-- mapping for Dog could go here -->
        </class>

</hibernate-mapping>
```

Etudions le contenu du document de mapping. Nous décrirons uniquement les éléments et attributs du document utilisés par Hibernate à l'exécution. Le document de mapping contient aussi des attributs et éléments optionnels qui agissent sur le schéma de base de données exporté par l'outil de génération de schéma. (Par exemple l'attribut `not-null`.)

5.1.1. Doctype

Tous les mappings XML devraient utiliser le doctype indiqué. Ce fichier est présent à l'URL ci-dessus, dans le répertoire `hibernate-x.x.x/src/org/hibernate` ou dans `hibernate3.jar`. Hibernate va toujours chercher la DTD dans son classpath en premier lieu. Si vous constatez des recherches de la DTD sur Internet, vérifiez votre déclaration de DTD par rapport au contenu de votre classpath.

5.1.1.1. EntityResolver

Comme cité précédemment, Hibernate tentera de trouver les DTDs d'abord dans son classpath. Il réussit à faire cela en utilisant une implémentation particulière de `org.xml.sax.EntityResolver` avec le SAXReader qu'il utilise pour lire les fichiers xml. Cet `EntityResolver` particulier reconnait deux espaces de nommage systemId différents.

- un `espace de nommage hibernate` est reconnu dès qu'un systemId commence par `http://hibernate.sourceforge.net/`; alors ces entités sont résolues via le classloader qui a chargé les classes Hibernate.
- un `espace de nommage utilisateur` est reconnu dès qu'un systemId utilise un protocol URL `classpath://`. Le résolveur tentera de résoudre ces entités via (1) le classloader du contexte du thread courant et (2) le classloader qui a chargé les classes Hibernate.

Un exemple d'utilisation de l'espace de nommage utilisateur:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd" [
    <!ENTITY types SYSTEM "classpath://your/domain/types.xml">
]>

<hibernate-mapping package="your.domain">
    <class name="MyEntity">
        <id name="id" type="my-custom-id-type">
            ...
        </id>
    <class>
    &types;
</hibernate-mapping>
```

Où `types.xml` est une ressource dans le package `your.domain` et qui contient un Section 5.2.3, « Types de valeur définis par l'utilisateur » particulier.

5.1.2. hibernate-mapping

Cet élément a plusieurs attributs optionnels. Les attributs `schema` et `catalog` indiquent que les tables référencées par ce mapping appartiennent au schéma nommé et/ou au catalogue. S'ils sont spécifiés, les noms de tables seront qualifiés par les noms de schéma et catalogue. L'attribut `default-cascade` indique quel type de cascade sera utlisé par défaut pour les propriétés et collections qui ne précisent pas l'attribut `cascade`. L'attribut `auto-import` nous permet d'utiliser par défaut des noms de classes non qualifiés dans le langage de requête.

```
<hibernate-mapping
        schema="schemaName"                         (1)
        catalog="catalogName"                       (2)
        default-cascade="cascade_style"             (3)
        default-access="field|property|ClassName"   (4)
        default-lazy="true|false"                   (5)
        auto-import="true|false"                    (6)
        package="package.name"                      (7)
 />
```

(1) `schema` (optionnel) : Le nom d'un schéma de base de données.

(2) `catalog` (optionnel) : Le nom d'un catalogue de base de données.

(3) `default-cascade` (optionnel - par défaut vaut : `none`) : Un type de cascade par défaut.

(4) `default-access` (optionnel - par défaut vaut : `property`) : Comment hibernate accèdera aux propriétés. On peut aussi redéfinir sa propre implémentation de `PropertyAccessor`.

(5) `default-lazy` (optionnel - par défaut vaut : `true`) : Valeur par défaut pour un attribut `lazy` non spécifié : celui des mappings de classes et de collection.

(6) `auto-import` (optionnel - par défaut vaut : `true`) : Spécifie si l'on peut utiliser des noms de classes non qualifiés (des classes de ce mapping) dans le langage de requête.

(7) `package` (optionnel) : Préfixe de package par défaut pour les noms de classe non qualifiés du document de mapping.

Si deux classes possèdent le même nom de classe (non qualifié), vous devez indiquer `auto-import="false"`. Hibernate lancera une exception si vous essayez d'assigner à deux classes le même nom importé.

Notez que l'élément `hibernate-mapping` vous permet d'imbriquer plusieurs mappings de `<class>` persistantes, comme dans l'exemple ci-dessus. Cependant la bonne pratique (ce qui est attendu par certains outils) est de mapper une seule classe (ou une seule hiérarchie de classes) par fichier de mapping et de nommer ce fichier d'après le nom de la superclasse, par exemple `Cat.hbm.xml`, `Dog.hbm.xml`, ou en cas d'héritage, `Animal.hbm.xml`.

5.1.3. class

Déclarez une classe persistante avec l'élément `class` :

```
<class
        name="ClassName"                            (1)
        table="tableName"                           (2)
        discriminator-value="discriminator_value"   (3)
        mutable="true|false"                        (4)
        schema="owner"                              (5)
        catalog="catalog"                           (6)
        proxy="ProxyInterface"                      (7)
        dynamic-update="true|false"                 (8)
        dynamic-insert="true|false"                 (9)
        select-before-update="true|false"           (10)
```

```
        polymorphism="implicit|explicit"              (11)
        where="arbitrary sql where condition"         (12)
        persister="PersisterClass"                    (13)
        batch-size="N"                                (14)
        optimistic-lock="none|version|dirty|all"      (15)
        lazy="true|false"                             (16)
        entity-name="EntityName"                      (17)
        catalog="catalog"                             (18)
        check="arbitrary sql check condition"         (19)
        rowid="rowid"                                 (20)
        subselect="SQL expression"                    (21)
        abstract="true|false"
        entity-name="EntityName"
/>
```

(1) `name` (optionnel) : Le nom Java complet de la classe (ou interface) persistante. Si cet attribut est absent, il est supposé que ce mapping ne se rapporte pas à une entité POJO.

(2) `table` (optionnel - par défaut le nom (non-qualifié) de la classe) : Le nom de sa table en base de données.

(3) `discriminator-value` (optionnel - par défaut le nom de la classe) : Une valeur permettant de distinguer les sous-classes dans le cas de l'utilisation du polymorphisme. Les valeurs `null` et `not null` sont autorisées.

(4) `mutable` (optionnel, vaut `true` par défaut) : Spécifie que des instances de la classe sont (ou non) immuables.

(5) `schema` (optionnel) : Surcharge le nom de schéma spécifié par l'élément racine `<hibernate-mapping>`.

(6) `catalog` (optionnel) : Surcharge le nom du catalogue spécifié par l'élément racine `<hibernate-mapping>`.

(7) `proxy` (optionnel) : Spécifie une interface à utiliser pour l'initialisation différée (lazy loading) des proxies. Vous pouvez indiquer le nom de la classe elle-même.

(8) `dynamic-update` (optionnel, par défaut à `false`) : Spécifie que les `UPDATE` SQL doivent être générés à l'exécution et contenir uniquement les colonnes dont les valeurs ont été modifiées.

(9) `dynamic-insert` (optionnel, par défaut à `false`): Spécifie que les `INSERT` SQL doivent être générés à l'exécution et ne contenir que les colonnes dont les valeurs sont non nulles.

(10) `select-before-update` (optionnel, par défaut à `false`): Spécifie que Hibernate ne doit *jamais* exécuter un `UPDATE` SQL sans être certain qu'un objet a été réellement modifié. Dans certains cas, (en réalité, seulement quand un objet transient a été associé à une nouvelle session par `update()`), cela signifie que Hibernate exécutera un `SELECT` SQL pour s'assurer qu'un `UPDATE` SQL est véritablement nécessaire.

(11) `polymorphism` (optionnel, vaut `implicit` par défaut) : Détermine si, pour cette classe, une requête polymorphique implicite ou explicite est utilisée.

(12) `where` (optionnel) spécifie une clause SQL `WHERE` à utiliser lorsque l'on récupère des objets de cette classe.

(13) `persister` (optionnel) : Spécifie un `ClassPersister` particulier.

(14) `batch-size` (optionnel, par défaut = `1`) : spécifie une taille de batch pour remplir les instances de cette classe par identifiant en une seule requête.

(15) `optimistic-lock` (optionnel, par défaut = `version`) : Détermine la stratégie de verrou optimiste.

(16) `lazy` (optionnel) : Déclarer `lazy="true"` est un raccourci pour spécifier le nom de la classe comme étant l'interface `proxy`.

(17) `entity-name` (optionnel) : Hibernate3 permet à une classe d'être mappée plusieurs fois (potentiellement à plusieurs tables), et permet aux mappings d'entité d'être représentés par des Maps ou du XML au niveau Java. Dans ces cas, vous devez indiquer un nom explicite arbitraire pour les entités. Voir Section 4.4, « Modèles dynamiques » et Chapitre 18, *Mapping XML* pour plus d'informations.

(18) `catalog` (optionnel) : The name of a database catalog used for this class and its table.

(19) `check` (optionnel) : expression SQL utilisée pour générer une contrainte de vérification multi-lignes pour la génération automatique de schéma.

(20) `rowid` (optionnel) : Hibernate peut utiliser des ROWID sur les bases de données qui utilisent ce mécanisme. Par exemple avec Oracle, Hibernate peut utiliser la colonne additionnelle `rowid` pour des mises à jour rapides si cette option vaut `rowid`. Un ROWID représente la localisation physique d'un tuple enregistré.

(21) `subselect` (optionnel) : Permet de mapper une entité immuable en lecture-seule sur un sous-select de base de données. Utile pour avoir une vue au lieu d'une table en base, mais à éviter. Voir plus bas pour plus d'information.

class.abstract (optionnel) : Utilisé pour marquer des superclasses abstraites dans des hiérarchies de ??? `<union-subclass>`.

Il est tout à fait possible d'utiliser une interface comme nom de classe persistante. Vous devez alors déclarer les classes implémentant cette interface en utilisant l'élément `<subclass>`. Vous pouvez faire persister toute classe interne *static*. Vous devez alors spécifier le nom de la classe par la notation habituelle des classes internes c'est à dire `eg.Foo$Bar`.

Les classes immuables, `mutable="false"`, ne peuvent pas être modifiées ou supprimées par l'application. Cela permet à Hibernate de faire quelques optimisations mineures sur les performances.

L'attribut optionnnel `proxy` permet les intialisations différées des instances persistantes de la classe. Hibernate retournera initialement des proxies CGLIB qui implémentent l'interface nommée. Le véritable objet persistant ne sera chargé que lorsque une méthode du proxy sera appelée. Voir plus bas le paragraphe abordant les proxies et le chargement différé (lazy initialization).

Le polymorphisme *implicite* signifie que les instances de la classe seront retournées par une requête qui utilise les noms de la classe ou de chacune de ses superclasses ou encore des interfaces implémentées par cette classe ou ses superclasses. Les instances des classes filles seront retournées par une requête qui utilise le nom de la classe elle même. Le polymorphisme *explicite* signifie que les instances de la classe ne seront retournées que par une requête qui utilise explicitement son nom et que seules les instances des classes filles déclarées dans les éléments `<subclass>` ou `<joined-subclass>` seront retournées. Dans la majorités des cas la valeur par défaut, `polymorphism="implicit"`, est appropriée. Le polymorphisme explicite est utile lorsque deux classes différentes sont mappées à la même table (ceci permet d'écrire une classe "légère" qui ne contient qu'une partie des colonnes de la table - voir la partie design pattern du site communautaire).

L'attribut `persister` vous permet de customiser la stratégie utilisée pour la classe. Vous pouvez, par exemple, spécifier votre propre sous-classe de `org.hibernate.persister.EntityPersister` ou vous pourriez aussi créer une nouvelle implémentation de l'interface `org.hibernate.persister.ClassPersister` qui proposerait une persistance via, par exemple, des appels de procédures stockées, de la sérialisation vers des fichiers plats ou un annuaire LDAP. Voir `org.hibernate.test.CustomPersister` pour un exemple simple (d'une "persistance" vers une `Hashtable`).

Notez que les paramètres `dynamic-update` et `dynamic-insert` ne sont pas hérités par les sous-classes et peuvent donc être spécifiés pour les éléments `<subclass>` ou `<joined-subclass>` Ces paramètres peuvent améliorer les performances dans certains cas, mais peuvent aussi les amoindrir. A utiliser en connaissance de causes.

L'utilisation de `select-before-update` va généralement faire baisser les performances. Ce paramètre est pratique pour prévenir l'appel inutile d'un trigger sur modification quand on réattache un graphe d'instances à une `Session`.

Si vous utilisez le `dynamic-update`, les différentes stratégies de verrouillage optimiste (optimistic locking) sont les suivantes:

- `version` vérifie les colonnes version/timestamp
- `all` vérifie toutes les colonnes
- `dirty` vérifie les colonnes modifiées, permettant des updates concurrents
- `none` pas de verrouillage optimiste

Nous encourageons *très* fortement l'utilisation de colonnes de version/timestamp pour le verrouillage optimiste avec Hibernate. C'est la meilleure stratégie en regard des performances et la seule qui gère correctement les modifications sur les objets détachés (c'est à dire lorsqu'on utilise `Session.merge()`).

Il n'y a pas de différence entre table et vue pour le mapping Hibernate, tant que c'est transparent au niveau base de données (remarquez que certaines BDD ne supportent pas les vues correctement, notamment pour les updates). Vous rencontrerez peut-être des cas où vous souhaitez utiliser une vue mais ne pouvez pas en créer sur votre BDD (par exemple à cause de schémas anciens et figés). Dans ces cas, vous pouvez mapper une entité immuable en lecture seule sur un sous-select SQL donné:

```
<class name="Summary">
    <subselect>
        select item.name, max(bid.amount), count(*)
        from item
        join bid on bid.item_id = item.id
        group by item.name
    </subselect>
    <synchronize table="item"/>
    <synchronize table="bid"/>
    <id name="name"/>
    ...
</class>
```

Déclarez les tables à synchroniser avec cette entité pour assurer que le flush automatique se produise correctement, et pour que les requêtes sur l'entité dérivée ne renvoient pas des données périmées. Le litéral `<subselect>` est disponible comme attribut ou comme élément de mapping.

5.1.4. id

Les classes mappées *doivent* déclarer la clef primaire de la table en base de données. La plupart des classes auront aussi une propriété de type javabean présentant l'identifiant unique d'une instance. L'élément `<id>` sert à définir le mapping entre cette propriété et la clef primaire en base.

```
<id
        name="propertyName"                                     (1)
        type="typename"                                         (2)
        column="column_name"                                    (3)
        unsaved-value="null|any|none|undefined|id_value"        (4)
        access="field|property|ClassName">                      (5)

        <generator class="generatorClass"/>
</id>
```

(1) `name` (optionnel) : Nom de la propriété qui sert d'identifiant.
(2) `type` (optionnel) : Nom indiquant le type Hibernate.
(3) `column` (optionnel - le nom de la propriété est pris par défaut) : Nom de la clef primaire.
(4) `unsaved-value` (optionnel - par défaut une valeur "bien choisie") : Une valeur de la propriété d'identifiant qui indique que l'instance est nouvellement instanciée (non sauvegardée), et qui la distingue des instances `transients` qui ont été sauvegardées ou chargées dans une session précédente.
(5) `access` (optionnel - par défaut `property`) : La stratégie que doit utiliser Hibernate pour accéder aux valeurs des propriétés.

Si l'attribut `name` est absent, Hibernate considère que la classe ne possède pas de propriété identifiant.

L'attribut `unsaved-value` est important ! Si l'identifiant de votre classe n'a pas une valeur par défaut compatible avec le comportement standard de Java (zéro ou null), vous devez alors préciser la valeur par défaut.

La déclaration alternative `<composite-id>` permet l'acccès aux données d'anciens systèmes qui utilisent des

clefs composées. Son utilisation est fortement déconseillée pour d'autres cas.

5.1.4.1. Generator

L'élément fils `<generator>` nomme une classe Java utilisée pour générer les identifiants uniques pour les instances des classes persistantes. Si des paramètres sont requis pour configurer ou initialiser l'instance du générateur, ils sont passés en utilisant l'élément `<param>`.

```
<id name="id" type="long" column="cat_id">
        <generator class="org.hibernate.id.TableHiLoGenerator">
                <param name="table">uid_table</param>
                <param name="column">next_hi_value_column</param>
        </generator>
</id>
```

Tous les générateurs doivent implémenter l'interface `org.hibernate.id.IdentifierGenerator`. C'est une interface très simple ; certaines applications peuvent proposer leur propre implémentations spécialisées. Cependant, Hibernate propose une série d'implémentations intégrées. Il existe des noms raccourcis pour les générateurs intégrés :

`increment`
: Génère des identifiants de type `long`, `short` ou `int` qui ne sont uniques que si aucun autre processus n'insère de données dans la même table. *Ne pas utiliser en environnement clusterisé.*

`identity`
: Utilisation de la colonne identity de DB2, MySQL, MS SQL Server, Sybase et HypersonicSQL. L'identifiant renvoyé est de type `long`, `short` ou `int`.

`sequence`
: Utilisation des séquences dans DB2, PostgreSQL, Oracle, SAP DB, McKoi ou d'un générateur dans Interbase. L'identifiant renvoyé est de type `long`, `short` ou `int`

`hilo`
: Utilise un algorithme hi/lo pour générer de façon efficace des identifiants de type `long`, `short` ou `int`, en prenant comme source de valeur "hi" une table et une colonne (par défaut `hibernate_unique_key` et `next_hi` respectivement). L'algorithme hi/lo génère des identifiants uniques pour une base de données particulière seulement.

`seqhilo`
: Utilise un algorithme hi/lo pour générer efficacement des identifiants de type `long`, `short` ou `int`, étant donné un nom de séquence en base.

`uuid`
: Utilise un algorithme de type UUID 128 bits pour générer des identifiants de type string, unique au sein d'un réseau (l'adresse IP est utilisée). Le UUID en codé en une chaîne de nombre héxadécimaux de longueur 32.

`guid`
: Utilise une chaîne GUID générée par la base pour MS SQL Server et MySQL.

`native`
: Choisit `identity`, `sequence` ou `hilo` selon les possibilités offertes par la base de données sous-jacente.

`assigned`
: Laisse l'application affecter un identifiant à l'objet avant que la métode `save()` soit appelée. Il s'agit de la

stratégie par défaut si aucun `<generator>` n'est spécifié.

`select`
Récupère une clef primaire assignée par un trigger en sélectionnant la ligne par une clef unique quelconque.

`foreign`
Utilise l'identifiant d'un objet associé. Habituellement utilisé en conjonction avec une association `<one-to-one>` sur la clef primaire.

5.1.4.2. algorithme Hi/lo

Les générateurs `hilo` et `seqhilo` proposent deux implémentations alternatives de l'algorithme hi/lo, une approche largement utilisée pour générer des identifiants. La première implémentation nécessite une table "spéciale" en base pour héberger la prochaine valeur "hi" disponible. La seconde utilise une séquence de type Oracle (quand la base sous-jacente le propose).

```
<id name="id" type="long" column="cat_id">
        <generator class="hilo">
                <param name="table">hi_value</param>
                <param name="column">next_value</param>
                <param name="max_lo">100</param>
        </generator>
</id>
```

```
<id name="id" type="long" column="cat_id">
        <generator class="seqhilo">
                <param name="sequence">hi_value</param>
                <param name="max_lo">100</param>
        </generator>
</id>
```

Malheureusement, vous ne pouvez pas utilisez `hilo` quand vous apportez votre propre `Connection` à Hibernate. Quand Hibernate utilise une datasource du serveur d'application pour obtenir des connexions inscrites avec JTA, vous devez correctement configurer `hibernate.transaction.manager_lookup_class`.

5.1.4.3. UUID algorithm

Le contenu du UUID est : adresse IP, date de démarrage de la JVM (précis au quart de seconde), l'heure système et un compteur (unique au sein de la JVM). Il n'est pas possible d'obtenir l'adresse MAC ou une adresse mémoire à partir de Java, c'est donc le mieux que l'on puisse faire sans utiliser JNI.

5.1.4.4. Colonnes identifiantes et séquences

Pour les bases qui implémentent les colonnes "identité" (DB2, MySQL, Sybase, MS SQL), vous pouvez utiliser la génération de clef par `identity`. Pour les bases qui implémentent les séquences (DB2, Oracle, PostgreSQL, Interbase, McKoi, SAP DB) vous pouvez utiliser la génération de clef par `sequence`. Ces deux méthodes nécessitent deux requêtes SQL pour insérer un objet.

```
<id name="id" type="long" column="person_id">
        <generator class="sequence">
                <param name="sequence">person_id_sequence</param>
        </generator>
</id>
```

```
<id name="id" type="long" column="person_id" unsaved-value="0">
        <generator class="identity"/>
</id>
```

Pour le développement multi-plateformes, la stratégie `native` choisira entre les méthodes `identity`, `sequence` et `hilo`, selon les possibilités offertes par la base sous-jacente.

5.1.4.5. Identifiants assignés

Si vous souhaitez que l'application assigne des identifiants (par opposition à la génération par Hibernate), vous pouvez utiliser le générateur `assigned`. Ce générateur spécial utilisera une valeur d'identifiant déjà utilisé par la propriété identifiant l'objet. Ce générateur est utilisé quand la clef primaire est une clef naturelle plutôt qu'une clef secondaire. C'est le comportement par défaut si vous ne précisez pas d'élément `<generator>`.

Choisir le générateur `assigned` fait utiliser `unsaved-value="undefined"` par Hibernate, le forçant à interroger la base pour déterminer si l'instance est transiente ou détachée, à moins d'utiliser une propriété version ou timestamp, ou alors de définir `Interceptor.isUnsaved()`.

5.1.4.6. Clefs primaires assignées par trigger

Pour les schémas de base hérités d'anciens systèmes uniquement (Hibernate ne génère pas de DDL avec des triggers)

```
<id name="id" type="long" column="person_id">
        <generator class="select">
                <param name="key">socialSecurityNumber</param>
        </generator>
</id>
```

Dans l'exemple ci-dessus, `socialSecurityNumber` a une valeur unique définie par la classe en tant que clef naturelle et `person_id` est une clef secondaire dont la valeur est générée par trigger.

5.1.5. composite-id

```
<composite-id
        name="propertyName"
        class="ClassName"
        mapped="true|false"
        access="field|property|ClassName">
        node="element-name|."

        <key-property name="propertyName" type="typename" column="column_name"/>
        <key-many-to-one name="propertyName class="ClassName" column="column_name"/>
        ......
</composite-id>
```

Pour une table avec clef composée, vous pouvez mapper plusieurs attributs de la classe comme propriétés identifiantes. L'élement `<composite-id>` accepte les mappings de propriétés `<key-property>` et les mappings `<key-many-to-one>` comme fils.

```
<composite-id>
        <key-property name="medicareNumber"/>
        <key-property name="dependent"/>
</composite-id>
```

Vos classes persistantes *doivent* surcharger les méthodes `equals()` et `hashCode()` pour implémenter l'égalité d'identifiant composé. Elles doivent aussi implenter l'interface `Serializable`.

Malheureusement cette approche sur les identifiants composés signifie qu'un objet persistant est son propre identifiant. Il n'y a pas d'autre moyen pratique de manipuler l'objet que par l'objet lui-même. Vous devez instancier une instance de la classe persistante elle-même et peupler ses attributs identifiants avant de pouvoir

appeler la méthode `load()` pour charger son état persistant associé à une clef composée. Nous appelons cette approche "identifiant composé *embarqué*" et ne la recommandons pas pour des applications complexes.

Une seconde approche, appelée identifiant composé *mappé*, consiste à encapsuler les propriétés identifiantes (celles contenues dans `<composite-id>`) dans une classe particulière.

```
<composite-id class="MedicareId" mapped="true">
        <key-property name="medicareNumber"/>
        <key-property name="dependent"/>
</composite-id>
```

Dans cet exemple, la classe d'identifiant composée,`MedicareId` et la classe mappée elle-même, possèdent les propriétés `medicareNumber` et `dependent`. La classe identifiante doit redéfinir `equals()` et `hashCode()` et implémenter `Serializable`. Le désavantage de cette approche est la duplication du code.

Les attributs suivants servent à configurer un identifiant composé mappé :

- `mapped` (optionnel, défaut à `false`) : indique qu'un identifiant composé mappé est utilisé, et que les propriétés contenues font référence aux deux classes (celle mappée et la classe identifiante).
- `class` (optionnel, mais requis pour un identifiant composé mappé) : La classe composant utilisée comme identifiant composé.

Nous décrirons une troisième approche beaucoup plus efficace ou l'identifiant composé est implémenté comme une classe composant dans Section 8.4, « Utiliser un composant comme identifiant ». Les attributs décrits ci dessous, ne s'appliquent que pour cette dernière approche :

- `name` (optionnel, requis pour cette approche) : une propriété de type composant qui contient l'identifiant composé (voir chapitre 9).
- `access` (optionnel - défaut à `property`) : La stratégie qu'Hibernate utilisera pour accéder à la valeur de la propriété.
- `class` (optionnel - défaut au type de la propriété déterminé par réflexion) : La classe composant utilisée comme identifiant (voir prochaine section).

Cette dernière approche est celle que nous recommandons pour toutes vos applications.

5.1.6. discriminator

L'élément `<discriminator>` est nécessaire pour la persistance polymorphique qui utilise la stratégie de mapping de table par hiérarchie de classe. La colonne discriminante contient une valeur marqueur qui permet à la couche de persistance de savoir quelle sous-classe instancier pour une ligne particulière de table en base. Un nombre restreint de types peuvent être utilisés : `string`, `character`, `integer`, `byte`, `short`, `boolean`, `yes_no`, `true_false`.

```
<discriminator
        column="discriminator_column"                    (1)
        type="discriminator_type"                        (2)
        force="true|false"                               (3)
        insert="true|false"                              (4)
        formula="arbitrary sql expression"               (5)
/>
```

(1) `column` (optionnel - par défaut à `class`) le nom de la colonne discriminante.

(2) `type` (optionnel - par défaut à `string`) un nom indiquant le type Hibernate.

(3) `force` (optionnel - par défaut à `false`) "oblige" Hibernate à spécifier une valeur discriminante autorisée même quand on récupère toutes les instances de la classe de base.

(4) `insert` (optionnel - par défaut à `true`) à passer à `false` si la colonne discriminante fait aussi partie d'un identifiant composé mappé (Indique à Hibernate de ne pas inclure la colonne dans les `INSERT` SQL).

(5) `formula` (optionnel) une expression SQL arbitraire qui est exécutée quand un type doit être évalué. Permet la discrimination basée sur le contenu.

Les véritables valeurs de la colonne discriminante sont spécifiées par l'attribut `discriminator-value` des éléments `<class>` et `<subclass>`.

L'attribut `force` n'est utile que si la table contient des lignes avec des valeurs "extra" discriminantes qui ne sont pas mappées à une classe persistante. Ce ne sera généralement pas le cas.

En utilisant l'attribut `formula` vous pouvez déclarer une expression SQL arbitraire qui sera utilisée pour évaluer le type d'une ligne :

```
<discriminator
    formula="case when CLASS_TYPE in ('a', 'b', 'c') then 0 else 1 end"
    type="integer"/>
```

5.1.7. version (optionnel)

L'élément `<version>` est optionnel et indique que la table contient des données versionnées. C'est particulièrement utile si vous avez l'intention d'utiliser des *transactions longues* (voir plus-bas).

```
<version
        column="version_column"                         (1)
        name="propertyName"                             (2)
        type="typename"                                 (3)
        access="field|property|ClassName"               (4)
        unsaved-value="null|negative|undefined"         (5)
        generated="never|always"                                  (6)
        insert="true|false"                                       (7)
        node="element-name|@attribute-name|element/@attribute|."
/>
```

(1) `column` (optionnel - par défaut égal au nom de la propriété) : Le nom de la colonne contenant le numéro de version.

(2) `name` : Le nom d'un attribut de la classe persistante.

(3) `type` (optionnel - par défaut à `integer`) : Le type du numéro de version.

(4) `access` (optionnel - par défaut à `property`) : La stratégie à utiliser par Hibernate pour accéder à la valeur de la propriété.

(5) `unsaved-value` (optionnel - par défaut à `undefined`) : Une valeur de la propriété d'identifiant qui indique que l'instance est nouvellement instanciée (non sauvegardée), et qui la distingue des instances détachées qui ont été sauvegardées ou chargées dans une session précédente (`undefined` indique que la valeur de l'atribut identifiant devrait être utilisé).

(6) `generated` (optional - défaut à `never`) : Indique que la valeur de la propriété version est générée par la base de données cf. Section 5.6, « Propriétés générées ».

(7) `insert` (optionnel - défaut à `true`) : Indique si la colonne de version doit être incluse dans les ordres insert. Peut être à `false` si et seulement si la colonne de la base de données est définie avec une valeur par défaut à `0`.

Les numéros de version doivent avoir les types Hibernate `long`, `integer`, `short`, `timestamp` ou `calendar`.

Une propriété de version ou un timestamp ne doit jamais être null pour une instance détachée, ainsi Hibernate pourra détecter toute instance ayant une version ou un timestamp null comme transient, quelles que soient les stratégies `unsaved-value` spécifiées. *Déclarer un numéro de version ou un timestamp "nullable" est un moyen pratique d'éviter tout problème avec les réattachements transitifs dans Hibernate, particulièrement utile pour*

ceux qui utilisent des identifiants assignés ou des clefs composées !

5.1.8. timestamp (optionnel)

L'élément optionnel `<timestamp>` indique que la table contient des données horodatées (timestamp). Cela sert d'alternative à l'utilisation de numéros de version. Les timestamps (ou horodatage) sont par nature une implémentation moins fiable pour l'optimistic locking. Cependant, l'application peut parfois utiliser l'horodatage à d'autres fins.

```
<timestamp
        column="timestamp_column"           (1)
        name="propertyName"                 (2)
        access="field|property|ClassName"   (3)
        unsaved-value="null|undefined"      (4)
        source="vm|db"                                          (5)
        generated="never|always"                                (6)
        node="element-name|@attribute-name|element/@attribute|."
/>
```

(1) `column` (optionnel - par défaut à le nom de la propriété) : Le nom d'une colonne contenant le timestamp.

(2) `name` : Le nom d'une propriété au sens JavaBean de type `Date` ou `Timestamp` de la classe persistante.

(3) `access` (optionnel - par défaut à `property`) : La stratégie à utiliser par Hibernate pour accéder à la valeur de la propriété.

(4) `unsaved-value` (optionnel - par défaut à `null`) : Propriété dont la valeur est un numéro de version qui indique que l'instance est nouvellement instanciée (non sauvegardée), et qui la distingue des instances détachées qui ont été sauvegardées ou chargées dans une session précédente (`undefined` indique que la valeur de l'attribut identifiant devrait être utilisée).

(5) `source` (optionnel - par défaut à `vm`) : D'où Hibernate doit-il récupérer la valeur du timestamp? Depuis la base de données ou depuis la JVM d'exécution? Les valeurs de timestamp de la base de données provoquent une surcharge puisque Hibernate doit interroger la base pour déterminer la prochaine valeur mais cela est plus sûr lorsque vous fonctionnez dans un cluster. Remarquez aussi que certains des dialectes ne supportent pas cette fonction, et que d'autres l'implémentent mal, provoquant des erreurs de précision (Oracle 8 par exemple).

(6) `generated` (optional - défaut à `never`) : Indique que la valeur de ce timestamp est générée par la base de données cf. Section 5.6, « Propriétés générées ».

Notez que `<timestamp>` est équivalent à `<version type="timestamp">` et `<timestamp source="db">` équivaut à `<version type="dbtimestamp">`.

5.1.9. property

L'élément `<property>` déclare une propriété de la classe au sens JavaBean.

```
<property
        name="propertyName"                 (1)
        column="column_name"                (2)
        type="typename"                     (3)
        update="true|false"                 (4)
        insert="true|false"                 (4)
        formula="arbitrary SQL expression"  (5)
        access="field|property|ClassName"   (6)
        lazy="true|false"                   (7)
        unique="true|false"                 (8)
        not-null="true|false"               (9)
        optimistic-lock="true|false"        (10)
        generated="never|insert|always"                         (11)
        node="element-name|@attribute-name|element/@attribute|."
        index="index_name"
        unique_key="unique_key_id"
```

```
        length="L"
        precision="P"
        scale="S"
/>
```

(1) `name` : nom de la propriété, avec une lettre initiale en minuscule.
(2) `column` (optionnel - par défaut au nom de la propriété) : le nom de la colonne mappée. Cela peut aussi être indiqué dans le(s) sous-élément(s) `<column>`.
(3) `type` (optionnel) : nom indiquant le type Hibernate.
(4) `update, insert` (optionnel - par défaut à `true`) : indique que les colonnes mappées devraient être incluses dans des `UPDATE` SQL et/ou des `INSERT`. Mettre les deux à `false` empêche la propagation en base de données (utile si vous savez qu'un trigger affectera la valeur à la colonne).
(5) `formula` (optionnel) : une expression SQL qui définit la valeur pour une propriété *calculée*. Les propriétés calculées ne possède pas leur propre mapping.
(6) `access` (optionnel - par défaut à `property`): Stratégie que Hibernate doit utiliser pour accéder à cette valeur.
(7) `lazy` (optionnel - par défaut à `false`): Indique que cette propriété devrait être chargée en différé (lazy loading) quand on accède à la variable d'instance pour la première fois.
(8) `unique` (optionnel): Génère le DDL d'une contrainte d'unicité pour les colonnes. Permet aussi d'en faire la cible d'un `property-ref`.
(9) `not-null` (optionnel): Génère le DDL d'une contrainte de non nullité pour les colonnes.
(10) `optimistic-lock` (optionnel - par défaut à `true`): Indique que les mises à jour de cette propriété peuvent ou non nécessiter l'acquisition d'un verrou optimiste. En d'autres termes, cela détermine s'il est nécessaire d'incrémenter un numéro de version quand cette propriété est marquée obsolète (dirty).
(11) `generated` (optional - défaut à`never`): Indique que la valeur de ce timestamp est générée par la base de données cf. Section 5.6, « Propriétés générées ».

typename peut être:

1. Nom d'un type basique Hibernate (ex: `integer, string, character, date, timestamp, float, binary, serializable, object, blob`).
2. Nom d'une classe Java avec un type basique par défaut (ex: `int, float, char, java.lang.String, java.util.Date, java.lang.Integer, java.sql.Clob`).
3. Nom d'une classe Java sérialisable.
4. Nom d'une classe ayant un type spécifique (ex: `com.illflow.type.MyCustomType`).

Si vous n'indiquez pas un type, Hibernate utlisera la réflexion sur le nom de la propriété pour tenter de trouver le type Hibernate correct. Hibernate essayera d'interprêter le nom de la classe retournée par le getter de la propriété en utilisant les régles 2, 3, 4 dans cet ordre. Cependant, ce n'est pas toujours suffisant. Dans certains cas vous aurez encore besoin de l'attribut `type` (Par exemple, pour distinguer `Hibernate.DATE` et `Hibernate.TIMESTAMP`, ou pour préciser un type spécifique).

L'attribut `access` permet de contrôler comment Hibernate accèdera à la propriété à l'exécution. Par défaut, Hibernate utilisera les méthodes set/get. Si vous indiquez `access="field"`, Hibernate ignorera les getter/setter et accèdera à la propriété directement en utilisant la réflexion. Vous pouvez spécifier votre propre stratégie d'accès aux propriété en donnant une classe qui implémente l'interface `org.hibernate.property.PropertyAccessor`.

Une fonctionnalité particulièrement intéressante est les propriétés dérivées. Ces propriétés sont par définition en lecture seule, la valeur de la propriété est calculée au chargement. Le calcul est déclaré comme une expression SQL, qui se traduit par une sous-requête `SELECT` dans la requête SQL qui charge une instance :

```
<property name="totalPrice"
    formula="( SELECT SUM (li.quantity*p.price) FROM LineItem li, Product p
```

```
                WHERE li.productId = p.productId
                AND li.customerId = customerId
                AND li.orderNumber = orderNumber )"/>
```

Remarquez que vous pouvez référencer la propre table des entités en ne déclarant pas un alias sur une colonne particulière (`customerId` dans l'exemple donné). Notez aussi que vous pouvez utiliser le sous-élément de mapping `<formula>` plutôt que d'utiliser l'attribut si vous le souhaitez.

5.1.10. many-to-one

Une association ordinaire vers une autre classe persistante est déclarée en utilisant un élément `many-to-one`. Le modèle relationnel est une association de type many-to-one : une clef étrangère dans une table référence la ou les clef(s) primaire(s) dans la table cible.

```
<many-to-one
        name="propertyName"                                 (1)
        column="column_name"                                (2)
        class="ClassName"                                   (3)
        cascade="cascade_style"                             (4)
        fetch="join|select"                                 (5)
        update="true|false"                                 (6)
        insert="true|false"                                 (6)
        property-ref="propertyNameFromAssociatedClass"      (7)
        access="field|property|ClassName"                   (8)
        unique="true|false"                                 (9)
        not-null="true|false"                               (10)
        optimistic-lock="true|false"                        (11)
        lazy="proxy|no-proxy|false"                         (12)
        not-found="ignore|exception"                        (13)          (14)
        entity-name="EntityName"                                    (15)
        formula="arbitrary SQL expression"                          (16)
        node="element-name|@attribute-name|element/@attribute|."
        embed-xml="true|false"
        index="index_name"
        unique_key="unique_key_id"
        foreign-key="foreign_key_name"
/>
```

(1) `name` : Nom de la propriété.

(2) `column` (optionnel) : Le nom de la clef étrangère. Cela peut être aussi indiqué avec le sous-élément `<column>`.

(3) `class` (optionnel - par défaut le type de la propriété déterminé par réflexion) : Le nom de la classe associée.

(4) `cascade` (optionnel) : Indique quelles opérations doivent être propagées de l'objet père vers les objets associés.

(5) `fetch` (optionnel - par défaut à `select`) : Choisit entre le chargement de type outer-join ou le chargement par select successifs.

(6) `update, insert` (optionnel - par défaut à `true`) : indique que les colonnes mappées devraient être incluses dans des `UPDATE` SQL et/ou des `INSERT`. Mettre les deux à `false` empêche la propagation en base de données (utile si vous savez qu'un trigger affectera la valeur à la colonne).

(7) `property-ref` : (optionnel) Le nom d'une propriété de la classe associée qui est liée à cette clef étrangère. Si ce n'est pas spécifié, la clef primaire de la classe associée est utilisée.

(8) `access` (optionnel - par défaut à `property`) : La stratégie à utiliser par Hibernate pour accéder à la valeur de cette propriété.

(9) `unique` (optionnel) : Génère le DDL d'une contrainte d'unicité pour la clef étrangère. Permet aussi d'en faire la cible d'un `property-ref`. Cela permet de créer une véritable association one-to-one.

(10) `not-null` (optionnel) : Génère le DDL pour une contrainte de non nullité pour la clef étrangère.

(11) `optimistic-lock` (optionnel - par défaut à `true`) : Indique que les mises à jour de cette propriété

requièrent ou non l'acquisition d'un verrou optimiste. En d'autres termes, détermine si un incrément de version doit avoir lieu quand la propriété est marquée obsolète (dirty).

(12) `lazy` (optionnel - par défaut à `false`) : Indique que cette propriété doit être chargée en différé (lazy loading) au premier accès à la variable d'instance (nécessite une instrumentation du bytecode lors de la phase de construction). Remarquez que cela n'influence pas le comportement du proxy Hibernate - comme l'attribut `lazy` sur des classes ou des mappings de collections, mais utilise l'interception pour le chargement différé. `lazy="false"` indique que l'association sera toujours chargée.

(13) `not-found` (optionnel - par défaut à `exception`) : Indique comment les clefs étrangères qui référencent des lignes manquantes doivent être manipulées : `ignore` traitera une ligne manquante comme une association nulle.

(15) `entity-name` (optionnel) : Le nom de l'entité de la classe associée.

(16) `formula` (optionnel) : une expression SQL qui définit la valeur pour une clé étrangère calculée.

Donner une valeur significative à l'attribut `cascade` autre que `none` propagera certaines opérations à l'objet associé. Les valeurs significatives sont les noms des opérations Hibernate basiques, `persist, merge, delete, save-update, evict, replicate, lock, refresh`, ainsi que les valeurs spéciales `delete-orphan` et `all` et des combinaisons de noms d'opérations séparées par des virgules, comme par exemple `cascade="persist,merge,evict"` ou `cascade="all,delete-orphan"`. Voir Section 10.11, « Persistance transitive » pour une explication complète. Notez que les assocations many-to-one et one-to-one ne supportent pas orphan delete.

Une déclaration `many-to-one` typique est aussi simple que :

```
<many-to-one name="product" class="Product" column="PRODUCT_ID"/>
```

L'attribut `property-ref` devrait être utilisé pour mapper seulement des données provenant d'un ancien système où les clefs étrangères font référence à une clef unique de la table associée et qui n'est pas la clef primaire. C'est un cas de mauvaise conception relationnelle. Par exemple, supposez que la classe `Product` a un numéro de série unique qui n'est pas la clef primaire. (L'attribut `unique` contrôle la génération DDL par Hibernate avec l'outil SchemaExport.)

```
<property name="serialNumber" unique="true" type="string" column="SERIAL_NUMBER"/>
```

Ainsi le mapping pour `OrderItem` peut utiliser :

```
<many-to-one name="product" property-ref="serialNumber" column="PRODUCT_SERIAL_NUMBER"/>
```

bien que ce ne soit certainement pas encouragé.

Si la clef unique référencée comprend des propriétés multiples de l'entité associée, vous devez mapper ces propriétés à l'intérieur d'un élément `<properties>`.

5.1.11. one-to-one

Une association one-to-one vers une autre classe persistante est déclarée avec l'élément `one-to-one`.

```
<one-to-one
        name="propertyName"                                 (1)
        class="ClassName"                                   (2)
        cascade="cascade_style"                             (3)
        constrained="true|false"                            (4)
        fetch="join|select"                                 (5)
        property-ref="propertyNameFromAssociatedClass"      (6)
        access="field|property|ClassName"                   (7)
        formula="any SQL expression"                        (8)
        entity-name="EntityName"                                        (9)
```

```
/>
```

(1) `name` : Le nom de la propriété.
(2) `class` (optionnel - par défaut du type de la propriété déterminé par réflexion) : Le nom de la classe associée.
(3) `cascade` (optionnel) : Indique quelles opérations doivent être cascadées de l'objet père vers l'objet associé.
(4) `constrained` (optionnel) : Indique qu'une contrainte de clef étrangère sur la clef primaire de la table mappée référence la table de la classe associée. Cette option affecte l'ordre dans lequel chaque `save()` et chaque `delete()` sont cascadés et détermine si l'association peut utiliser un proxy (aussi utilisé par l'outil d'export de schéma).
(5) `fetch` (optionnel - par défaut à `select`) : Choisit entre récupération par jointure externe ou select séquentiel.
(6) `property-ref` (optionnel) : Le nom de la propriété de la classe associée qui est jointe à la clef primaire de cette classe. Si ce n'est pas spécifié, la clef primaire de la classe associée est utilisée.
(7) `access` (optionnel - par défaut à `property`) : La stratégie à utiliser par Hibernate pour accéder à la valeur de la propriété.
(8) `formula` (optionnel) : Presque toutes les associations one-to-one pointent sur la clef primaire de l'entité propriétaire. Dans les rares cas différents, vous devez donner une ou plusieurs autres colonnes ou expression à joindre par une formule SQL (voir `org.hibernate.test.onetooneformula` pour un exemple).
(9) `lazy` (optionnel - par défaut `proxy`) : Par défaut, les associations simples sont soumise à proxy. `lazy="no-proxy"` spécifie que la propriété doit être chargée à la demande au premier accès à l'instance. (nécessite l'intrumentation du bytecode à la construction). `lazy="false"` indique que l'association sera toujours chargée agressivement. *Notez que si `constrained="false"`, l'utilisation de proxy est impossible et Hibernate chargera automatiquement l'association !*
(10) `entity-name` (optional) : The entity name of the associated class.

Il existe deux types d'associations one-to-one :

- associations par clef primaire
- association par clef étrangère unique

Les associations par clef primaire ne nécessitent pas une colonne supplémentaire en table ; si deux lignes sont liés par l'association alors les deux lignes de la table partagent la même valeur de clef primaire. Donc si vous voulez que deux objets soient liés par une association par clef primaire, vous devez faire en sorte qu'on leur assigne la même valeur d'identifiant !

Pour une association par clef primaire, ajoutez les mappings suivants à `Employee` et `Person`, respectivement.

```
<one-to-one name="person" class="Person"/>
```

```
<one-to-one name="employee" class="Employee" constrained="true"/>
```

Maintenant, vous devez faire en sorte que les clefs primaires des lignes liées dans les tables PERSON et EMPLOYEE sont égales. On utilise une stratégie Hibernate spéciale de génération d'identifiants appelée `foreign` :

```
<class name="person" table="PERSON">
    <id name="id" column="PERSON_ID">
        <generator class="foreign">
            <param name="property">employee</param>
        </generator>
    </id>
```

```
    ...
    <one-to-one name="employee"
        class="Employee"
        constrained="true"/>
</class>
```

Une instance fraîchement enregistrée de `Person` se voit alors assignée la même valeur de clef primaire que l'instance de `Employee` référencée par la propriété `employee` de cette `Person`.

Alternativement, une clef étrangère avec contrainte d'unicité de `Employee` vers `Person` peut être indiquée ainsi :

```
<many-to-one name="person" class="Person" column="PERSON_ID" unique="true"/>
```

Et cette association peut être rendue bidirectionnelle en ajoutant ceci au mapping de `Person` :

```
<one-to-one name"employee" class="Employee" property-ref="person"/>
```

5.1.12. natural-id

```
<natural-id mutable="true|false"/>
        <property ... />
        <many-to-one ... />
        ......
</natural-id>
```

Bien que nous recommandions l'utilisation de clé primaire générée, vous devriez toujours essayer d'identifier des clé métier (naturelles) pour toutes vos entités. Une clé naturelle est une propriété ou une combinaison de propriétés uniques et non nulles. Si elle est aussi immuable, c'est encore mieux. Mappez les propriétés de la clé naturelle dans l'élément `<natural-id>`. Hibernate générera la clé unique nécessaire et les contraintes de non-nullité, et votre mapping s'auto-documentera.

Nous vous recommandons fortement d'implémenter `equals()` et `hashCode()` pour comparer les clés naturelles de l'entité.

Ce mapping n'est pas destiné à être utilisé avec des entités qui ont des clés naturelles.

- `mutable` (optionel, par défaut à `false`) : Par défaut, les identifiants naturels sont supposés être immuable (constants).

5.1.13. component, dynamic-component

L'élément `<component>` mappe les propriétés d'un objet fils aux colonnes d'une classe parente. Les composants peuvent en retour déclarer leurs propres propriétés, composants ou collections. Voir "Components" plus bas.

```
<component
        name="propertyName"                 (1)
        class="className"                   (2)
        insert="true|false"                 (3)
        update="true|false"                 (4)
        access="field|property|ClassName"   (5)
        lazy="true|false"                   (6)
        optimistic-lock="true|false"        (7)
        unique="true|false"                 (8)
>

        <property ...../>
        <many-to-one .... />
        ........
```

```
</component>
```

(1) `name` : Nom de la propriété
(2) `class` (optionnel - par défaut au type de la propriété déterminé par réflexion) : le nom de la classe (fille) du composant.
(3) `insert` : Est ce que les colonnes mappées apparaissent dans les `INSERT`s ?
(4) `update`: Est ce que les colonnes mappées apparaissent dans les `UPDATE`s ?
(5) `access` (optionnel - par défaut à `property`) : La stratégie que Hibernate doit utiliser pour accéder à la valeur de cette propriété.
(6) `lazy` (optionnel - par défaut à `false`) : Indique que ce composant doit être chargé au premier accès à la variable d'instance (nécessite une instrumentation du bytecode au moment du build).
(7) `optimistic-lock` (optionnel - par défaut à `true`) : Indique que les mises à jour sur ce composant nécessitent ou non l'acquisition d'un verrou optimiste. En d'autres termes, cela détermine si une incrémentation de version doit avoir lieu quand la propriété est marquée obsolète (dirty).
(8) `unique` (optionnel - par défaut à `false`) : Indique qu'une contrainte d'unicité existe sur toutes les colonnes mappées de ce composant.

Les tags fils `<property>` mappent les propriétés de la classe fille sur les colonnes de la table.

L'élément `<component>` permet de déclarer sous-élément `<parent>` qui associe une propriété de la classe composant comme une référence arrière vers l'entité contenante.

L'élément `<dynamic-component>` permet à une `Map` d'être mappée comme un composant, quand les noms de la propriété font référence aux clefs de cette Map, voir Section 8.5, « Composant Dynamique ».

5.1.14. properties

L'élément `<properties>` permet la définition d'un groupement logique nommé des propriétés d'une classe. L'utilisation la plus importante de cette construction est la possibilité pour une combinaison de propriétés d'être la cible d'un `property-ref`. C'est aussi un moyen pratique de définir une contrainte d'unicité multi-colonnes.

```
<properties
        name="logicalName"                 (1)
        insert="true|false"                (2)
        update="true|false"                (3)
        optimistic-lock="true|false"       (4)
        unique="true|false"                (5)
>

        <property ...../>
        <many-to-one .... />
        ........
</properties>
```

(1) `name` : Le nom logique d'un regroupement et *non* le véritable nom d'une propriété.
(2) `insert` : Est-ce que les colonnes mappées apparaissent dans les `INSERT`s ?
(3) `update` : Est-ce que les colonnes mappées apparaissent dans les `UPDATE`s ?
(4) `optimistic-lock` (optionnel - par défaut à `true`) : Indique que les mises à jour sur ce composant nécessitent ou non l'acquisition d'un verrou optimiste. En d'autres termes, cela détermine si une incrémentation de version doit avoir lieu quand la propriété est marquée obsolète (dirty).
(5) `unique` (optionnel - par défaut à `false`) : Indique qu'une contrainte d'unicité existe sur toutes les colonnes mappées de ce composant.

Par exemple, si nous avons le mapping de `<properties>` suivant :

```
<class name="Person">
```

```
    <id name="personNumber"/>
    ...
    <properties name="name"
            unique="true" update="false">
        <property name="firstName"/>
        <property name="initial"/>
        <property name="lastName"/>
    </properties>
</class>
```

Alors nous pourrions avoir une association sur des données d'un ancien système (legacy) qui font référence à cette clef unique de la table `Person` au lieu de la clef primaire :

```
<many-to-one name="person"
         class="Person" property-ref="name">
    <column name="firstName"/>
    <column name="initial"/>
    <column name="lastName"/>
</many-to-one>
```

Nous ne recommandons pas l'utilisation de ce genre de chose en dehors du contexte de mapping de données héritées d'anciens systèmes.

5.1.15. subclass

Pour finir, la persistance polymorphique nécessite la déclaration de chaque sous-classe de la classe persistante de base. pour la stratégie de mapping de type table-per-class-hierarchy, on utilise la déclaration `<subclass>`.

```
<subclass
        name="ClassName"                                (1)
        discriminator-value="discriminator_value"       (2)
        proxy="ProxyInterface"                          (3)
        lazy="true|false"                               (4)
        dynamic-update="true|false"
        dynamic-insert="true|false"
        entity-name="EntityName"
        node="element-name"
        extends="SuperclassName">

        <property .... />
        .....
</subclass>
```

(1) `name` : Le nom complet de la sous-classe.

(2) `discriminator-value` (optionnel - par défaut le nom de la classe) : une valeur qui distingue les différentes sous-classes.

(3) `proxy` (optionnel) : Indique une classe ou interface à utiliser pour les chargements à la demande des proxies (lazy).

(4) `lazy` (optionnel, par défaut à `true`) : Spécifier `lazy="false"` désactive l'utilisation du chargement à la demande (lazy).

Chaque sous-classe devrait déclarer ses propres propriétés persistantes et sous-classes. Les propriétés `<version>` et `<id>` sont implicitement hérités de la classe de base. Chaque sous-classe dans une hiérarchie doit définir une unique `discriminator-value`. Si aucune n'est spécifiée, le nom complet de la classe Java est utilisé.

Pour plus d'infos sur le mapping d'héritage, voir Chapitre 9, *Mapping d'héritage de classe*.

```

<hibernate-mapping>
    <subclass name="DomesticCat" extends="Cat" discriminator-value="D">
```

```
        <property name="name" type="string"/>
    </subclass>
</hibernate-mapping>
```

Pour des informations sur les mappings d'héritage, voir Chapitre 9, *Mapping d'héritage de classe*.

5.1.16. joined-subclass

Une autre façon possible de faire est la suivante, chaque sous-classe peut être mappée vers sa propre table (stratégie de mapping de type table-per-subclass). L'état hérité est récupéré en joignant la table de la super-classe. L'élément `<joined-subclass>` est utilisé.

```
<joined-subclass
        name="ClassName"                    (1)
        table="tablename"                   (2)
        proxy="ProxyInterface"              (3)
        lazy="true|false"                   (4)
        dynamic-update="true|false"
        dynamic-insert="true|false"
        schema="schema"
        catalog="catalog"
        extends="SuperclassName"
        persister="ClassName"
        subselect="SQL expression"
        entity-name="EntityName">

        <key .... >

        <property .... />
        .....
</joined-subclass>
```

(1) `name` : Le nom Java complet de la sous-classe.
(2) `table` : Le nom de la table de la sous-classe.
(3) `proxy` (optionnel) : Indique une classe ou interface pour le chargement différé des proxies.
(4) `lazy` (optionnel, par défaut à `true`) : Indiquer `lazy="false"` désactive l'utilisation du chargement à la demande.

Aucune colonne discriminante n'est nécessaire pour cette stratégie de mapping. Cependant, chaque sous-classe doit déclarer une colonne de table contenant l'objet identifiant qui utilise l'élément `<key>`. Le mapping au début de ce chapitre serait ré-écrit ainsi :

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="eg">

        <class name="Cat" table="CATS">
                <id name="id" column="uid" type="long">
                        <generator class="hilo"/>
                </id>
                <property name="birthdate" type="date"/>
                <property name="color" not-null="true"/>
                <property name="sex" not-null="true"/>
                <property name="weight"/>
                <many-to-one name="mate"/>
                <set name="kittens">
                        <key column="MOTHER"/>
                        <one-to-many class="Cat"/>
                </set>
                <joined-subclass name="DomesticCat" table="DOMESTIC_CATS">
```

```
                    <key column="CAT"/>
                    <property name="name" type="string"/>
                </joined-subclass>
        </class>

        <class name="eg.Dog">
                <!-- mapping for Dog could go here -->
        </class>

</hibernate-mapping>
```

Pour des informations sur les mappings d'héritage, voir Chapitre 9, *Mapping d'héritage de classe*.

5.1.17. union-subclass

Une troisième option est de seulement mapper vers des tables les classes concrètes d'une hiérarchie d'héritage, (stratégie de type table-per-concrete-class) où chaque table définit tous les états persistants de la classe, y compris les états hérités. Dans Hibernate il n'est absolument pas nécessaire de mapper explicitement de telles hiérarchies d'héritage. Vous pouvez simplement mapper chaque classe avec une déclaration `<class>` différente. Cependant, si vous souhaitez utiliser des associations polymorphiques (càd une association vers la superclasse de la hiérarchie), vous devez utiliser le mapping `<union-subclass>`.

```
<union-subclass
        name="ClassName"                    (1)
        table="tablename"                   (2)
        proxy="ProxyInterface"              (3)
        lazy="true|false"                   (4)
        dynamic-update="true|false"
        dynamic-insert="true|false"
        schema="schema"
        catalog="catalog"
        extends="SuperclassName"
        abstract="true|false"
        persister="ClassName"
        subselect="SQL expression"
        entity-name="EntityName">

        <property .... />
        .....
</union-subclass>
```

(1) `name` : Le nom Java complet de la sous-classe.
(2) `table` : nom de la table de la sous-classe.
(3) `proxy` (optionnel) : Indique une classe ou interface pour le chargement différé des proxies.
(4) `lazy` (optionnel, par défaut à `true`) : Indiquer `lazy="false"` désactive l'utilisation du chargement à la demande.

Aucune colonne discriminante ou colonne clef n'est requise pour cette stratégie de mapping.

Pour des informations sur les mappings d'héritage, voir Chapitre 9, *Mapping d'héritage de classe*.

5.1.18. join

En utilisant l'élément `<join>`, il est possible de mapper des propriétés d'une classe sur plusieurs tables.

```
<join
        table="tablename"                       (1)
        schema="owner"                          (2)
        catalog="catalog"                       (3)
        fetch="join|select"                     (4)
```

```
        inverse="true|false"                    (5)
        optionnel="true|false">                 (6)

        <key ... />

        <property ... />
        ...
</join>
```

(1) `table` : Le nom de la table jointe.

(2) `schema` (optionnel) : court-circuite le nom de schéma spécifié par l'élément de base `<hibernate-mapping>`.

(3) `catalog` (optionnel) : court-circuite le nom de catalogue spécifié par l'élément de base `<hibernate-mapping>`.

(4) `fetch` (optionnel - par défaut à `join`) : Si positionné à `join`, Hibernate utilisera une jointure interne pour charger une `jointure` définie par une classe ou ses super-classes et une jointure externe pour une `<jointure>` définie par une sous-classe. Si positionné à `select` alors Hibernate utilisera un select séquentiel pour une `<jointure>` définie sur une sous-classe, qui ne sera délivrée que si une ligne se représente une instance de la sous-classe. Les jointures internes seront quand même utilisées pour charger une `<jointure>` définie par une classe et ses super-classes.

(5) `inverse` (optionnel - par défaut à `false`) : Si positionné à true, Hibernate n'essaiera pas d'insérer ou de mettre à jour les propriétés définies par cette jointure.

(6) `optionnel` (optionnel - par défaut à `false`) : Si positionné à true, Hibernate insèrera une ligne seulement si les propriétés définies par cette jointure sont non-nulles et utilisera toujours une jointure externe pour charger les propriétés.

Par exemple, les informations d'adresse pour une personne peuvent être mappées vers une table séparée (tout en préservant des sémantiques de type valeur pour toutes ses propriétés) :

```
<class name="Person"
    table="PERSON">

    <id name="id" column="PERSON_ID">...</id>

    <join table="ADDRESS">
        <key column="ADDRESS_ID"/>
        <property name="address"/>
        <property name="zip"/>
        <property name="country"/>
    </join>
    ...
```

Cette fonctionnalité est souvent seulement utile pour les modèles de données hérités d'anciens systèmes (legacy), nous recommandons d'utiliser moins de tables que de classes et un modèle de domaine à granularité fine. Cependant, c'est utile pour passer d'une stratégie de mapping d'héritage à une autre dans une hiérarchie simple ainsi qu'il est expliqué plus tard.

5.1.19. key

Nous avons rencontré l'élément `<key>` à plusieurs reprises maintenant. Il apparaît partout que l'élément de mapping parent définit une jointure sur une nouvele table, et définit la clef étrangère dans la table jointe, ce qui référence la clef primaire de la table d'origine.

```
<key
        column="columnname"                     (1)
        on-delete="noaction|cascade"            (2)
        property-ref="propertyName"             (3)
        not-null="true|false"                   (4)
```

```
        update="true|false"                    (5)
        unique="true|false"                    (6)
/>
```

(1) `column` (optionnel) : Le nom de la colonne de la clef étrangère Cela peut aussi être spécifié par l'élément(s) intégré(s) `<column>`.

(2) `on-delete` (optionnel, par défaut à `noaction`) : Indique si la contrainte de clef étrangère possède la possibilité au niveau base de données de suppression en cascade.

(3) `property-ref` (optionnel) : Indique que la clef étrangère fait référence à des colonnes qui ne sont pas la clef primaire de la table d'origine (Pour les données de systèmes legacy).

(4) `not-null` (optionnel) : Indique que les colonnes des clefs étrangères ne peuvent pas être nulles (c'est implicite si la clef étrangère fait partie de la clef primaire).

(5) `update` (optionnel) : Indique que la clef étrangère ne devrait jamais être mise à jour (implicite si celle-ci fait partie de la clef primaire).

(6) `unique` (optionnel) : Indique que la clef étrangère doit posséder une contrainte d'unicité (implicite si la clef étrangère est aussi la clef primaire).

Nous recommandons pour les systèmes où les suppressions doivent être performantes de définir toutes les clefs `on-delete="cascade"`, ainsi Hibernate utilisera une contrainte `ON CASCADE DELETE` au niveau base de données, plutôt que de nombreux `DELETE` individuels. Attention, cette fonctionnalité court-circuite la stratégie habituelle de verrou optimiste pour les données versionnées.

Les attributs `not-null` et `update` sont utiles pour mapper une association one-to-many unidirectionnelle. Si vous mappez un one-to-many unidirectionnel vers une clef étrangère non nulle, vous *devez* déclarer la colonne de la clef en utilisant `<key not-null="true">`.

5.1.20. éléments column et formula

Tout élément de mapping qui accepte un attribut `column` acceptera alternativement un sous-élément `<column>`. De façon identique, `<formula>` est une alternative à l'attribut `formula`.

```
<column
        name="column_name"
        length="N"
        precision="N"
        scale="N"
        not-null="true|false"
        unique="true|false"
        unique-key="multicolumn_unique_key_name"
        index="index_name"
        sql-type="sql_type_name"
        check="SQL expression"/>
```

```
<formula>SQL expression</formula>
```

Les attributs `column` et `formula` peuvent même être combinés au sein d'une même propriété ou mapping d'association pour exprimer, par exemple, des conditions de jointure exotiques.

```
<many-to-one name="homeAddress" class="Address"
        insert="false" update="false">
    <column name="person_id" not-null="true" length="10"/>
    <formula>'MAILING'</formula>
</many-to-one>
```

5.1.21. import

Supposez que votre application possède deux classes persistantes du même nom, et vous ne voulez pas préciser le nom Java complet (packages inclus) dans les queries Hibernate. Les classes peuvent alors être "importées" explicitement plutôt que de compter sur `auto-import="true"`.Vous pouvez même importer des classes et interfaces qui ne sont pas mappées explicitement.

```
<import class="java.lang.Object" rename="Universe"/>
```

```
<import
        class="ClassName"             (1)
        rename="ShortName"            (2)
/>
```

(1) `class` : Nom Java complet de la classe.

(2) `rename` (optionnel - par défaut vaut le nom de la classe Java (sans package)) : Nom pouvant être utilisé dans le langage de requête.

5.1.22. any

Il existe encore un type de mapping de propriété. L'élément de mapping `<any>` définit une association polymorphique vers des classes de tables multiples. Ce type de mapping requiert toujours plus d'une colonne. La première colonne contient le type de l'entité associée. Les colonnes restantes contiennent l'identifiant. il est impossible de spécifier une contrainte de clef étrangère pour ce type d'association, donc ce n'est certainement pas considéré comme le moyen habituel de mapper des associations (polymorphiques). Vous devriez utiliser cela uniquement dans des cas particuliers (par exemple des logs d'audit, des données de session utilisateur, etc...).

L'attribut `meta-type` permet à l'application de spécifier un type personnalisé qui mappe des valeurs de colonnes de le base de données sur des classes persistantes qui ont un attribut identifiant du type spécifié par `id-type`. Vous devez spécifier le mapping à partir de valeurs du méta-type sur les noms des classes.

```
<any name="being" id-type="long" meta-type="string">
    <meta-value value="TBL_ANIMAL" class="Animal"/>
    <meta-value value="TBL_HUMAN" class="Human"/>
    <meta-value value="TBL_ALIEN" class="Alien"/>
    <column name="table_name"/>
    <column name="id"/>
</any>
```

```
<any
        name="propertyName"                    (1)
        id-type="idtypename"                   (2)
        meta-type="metatypename"               (3)
        cascade="cascade_style"                (4)
        access="field|property|ClassName"      (5)
        optimistic-lock="true|false"           (6)
>
        <meta-value ... />
        <meta-value ... />
        .....
        <column .... />
        <column .... />
        .....
</any>
```

(1) `name` : le nom de la propriété.

(2) `id-type` : le type identifiant.

(3) `meta-type` (optionnel - par défaut à `string`) : Tout type permis pour un mapping par discriminateur.

(4) `cascade` (optionnel - par défaut à `none`) : le style de cascade.

(5) `access` (optionnel - par défaut à `property`) : La stratégie à utiliser par Hibernate pour accéder à cette propriété.

(6) `optimistic-lock` (optionnel - par défaut à `true`) : Indique que les mises à jour sur cette propriété nécessitent ou non l'acquisition d'un verrou optimiste. En d'autres termes, définit si un incrément de version doit avoir lieu quand cette propriété est marquée dirty.

5.2. Hibernate Types

5.2.1. Entités et valeurs

Pour comprendre le comportement des différents objets Java par rapport au service de persistance, nous avons besoin de les classer en deux groupes :

Une *entité* existe indépendamment de tout autre objet possédant une référence vers l'entité. Comparez cela avec le modèle Java habituel où un objet est supprimé par le garbage collector dès qu'il n'est plus référencé. Les entités doivent être explicitement enregistrées et supprimées (sauf dans les cas où sauvegardes et suppressions sont *cascadées* d'une entité mère vers ses enfants). C'est différent du modèle ODMG de persistance par atteignabilité - et correspond mieux à la façon dont les objets sont habituellement utilisés dans des grands systèmes. Les entités permettent les références circulaires et partagées. Elles peuvent aussi être versionnées.

L'état persistant d'une entité consiste en des références vers d'autres entités et instances de types *valeurs*. Ces valeurs sont des types primitifs, des collections (et non le contenu d'une collection), des composants de certains objets immuables. Contrairement aux entités, les valeurs (et en particulier les collections et composants) *sont* persistés par atteignabiliité. Comme les valeurs (et types primitifs) sont persistés et supprimés avec l'entité qui les contient, ils ne peuvent pas posséder leurs propres versions. Les valeurs n'ont pas d'identité indépendantes, ainsi elles ne peuvent pas être partagées par deux entités ou collections.

Jusqu'à présent nous avons utilisé le terme "classe persistante" pour parler d'entités. Nous allons continuer à faire ainsi. Cependant, au sens strict, toutes les classes définies par un utilisateur possédant un état persistant ne sont pas des entités. Un *composant* est une classe définie par un utilisateur avec les caractéristiques d'une valeur. Une propriété Java de type `java.lang.String` a aussi les caractéristiques d'une valeur. Given this definition, we can say that all types (classes) provided by the JDK have value type semantics in Java, while user-defined types may be mapped with entity or value type semantics. This decision is up to the application developer. A good hint for an entity class in a domain model are shared references to a single instance of that class, while composition or aggregation usually translates to a value type.

Nous nous pencherons sur ces deux concepts tout au long de la documentation.

Le défi est de mapper les type Javas (et la définition des développeurs des entités et valeurs types) sur les types du SQL ou des bases de données. Le pont entre les deux systèmes est proposé par Hibernate : pour les entités nous utilisons `<class>`, `<subclass>` et ainsi de suite. Pour les types valeurs nous utilisons `<property>`, `<component>`, etc., habituellement avec un attribut `type`. La valeur de cet attribut est le nom d'un *type de mapping* Hibernate. Hibernate propose de base de nombreux mappings (pour les types de valeurs standards du JDK). Vous pouvez écrire vos propres types de mappings et implémenter aussi vos propres stratégies de conversion, nous le verrons plus tard.

Tous les types proposés de base par Hibernate à part les collections autorisent la valeur null.

5.2.2. Basic value types

Les *types basiques de mapping* proposés de base peuvent grossièrement être rangés dans les catégories

suivantes :

`integer, long, short, float, double, character, byte, boolean, yes_no, true_false`
: Les mappings de type des primitives Java ou leurs classes wrappers (ex: Integer pour int) vers les types SQL (propriétaires) appropriés. `boolean`, `yes_no`et `true_false` sont tous des alternatives pour les types Java `boolean` ou `java.lang.Boolean`.

`string`
: Mapping de type de `java.lang.String` vers `VARCHAR` (ou le `VARCHAR2` Oracle).

`date, time, timestamp`
: Mappings de type pour `java.util.Date` et ses sous-classes vers les types SQL `DATE`, `TIME` et `TIMESTAMP` (ou équivalent).

`calendar, calendar_date`
: Mappings de type pour `java.util.Calendar` vers les types SQL `TIMESTAMP` et `DATE` (ou équivalent).

`big_decimal, big_integer`
: Mappings de type pour `java.math.BigDecimal` et `java.math.BigInteger` vers `NUMERIC` (ou le `NUMBER` Oracle).

`locale, timezone, currency`
: Mappings de type pour `java.util.Locale`, `java.util.TimeZone` et `java.util.Currency` vers `VARCHAR` (ou le `VARCHAR2` Oracle). Les instances de `Locale` et `Currency` sont mappées sur leurs codes ISO. Les instances de `TimeZone` sont mappées sur leur `ID`.

`class`
: Un type de mapping pour `java.lang.Class` vers `VARCHAR` (ou le `VARCHAR2` Oracle). Un objet `Class` est mappé sur son nom Java complet.

`binary`
: Mappe les tableaux de bytes vers le type binaire SQL approprié.

`text`
: Mappe les longues chaînes de caractères Java vers les types SQL `CLOB` ou `TEXT`.

`serializable`
: Mappe les types Java sérialisables vers le type SQL binaire approprié. Vous pouvez aussi indiquer le type Hibernate `serializable` avec le nom d'une classe Java sérialisable ou une interface qui ne soit pas par défaut un type de base.

`clob, blob`
: Mappings de type pour les classes JDBC `java.sql.Clob` and `java.sql.Blob`. Ces types peuvent ne pas convenir pour certaines applications car un objet blob ou clob peut ne pas être réutilisable en dehors d'une transaction (de plus l'implémentation par les pilotes est moyennement bonne).

`imm_date, imm_time, imm_timestamp, imm_calendar, imm_calendar_date, imm_serializable, imm_binary`
: Mappings de type pour ceux qui sont habituellement modifiable, pour lesquels Hibernate effectue certains optimisations convenant seulement aux types Java immuables, et l'application les traite comme immuable. Par exemple, vous ne devriez pas appeler `Date.setTime()` sur une instance mappée sur un `imm_timestamp`. Pour changer la valeur de la propriété, et faire que cette modification soit persistée, l'application doit assigner un nouvel (non identique) objet à la propriété.

Les identifiants uniques des entités et collections peuvent être de n'importe quel type de base excepté `binary`,

`blob` et `clob` (les identifiants composites sont aussi permis, voir plus bas).

Les types de base des valeurs ont des `Type` constants correspondants définis dans `org.hibernate.Hibernate`. Par exemple, `Hibernate.STRING` représenté le type `string`.

5.2.3. Types de valeur définis par l'utilisateur

Il est assez facile pour les développeurs de créer leurs propres types de valeurs. Par exemple, vous pourriez vouloir persister des propriétés du type `java.lang.BigInteger` dans des colonnnes `VARCHAR`. Hibernate ne procure pas par défaut un type pour cela. Mais les types que vous pouvez créer ne se limitent pas à mapper des propriétés (ou élément collection) à une simple colonne d'une table. Donc, par exemple, vous pourriez avoir une propriété Java `getName()`/`setName()` de type `java.lang.String` persistée dans les colonnes `FIRST_NAME`, `INITIAL`, `SURNAME`.

Pour implémenter votre propre type, vous pouvez soit implémenter `org.hibernate.UserType` soit `org.hibernate.CompositeUserType` et déclarer des propriétés utilisant des noms de classes complets du type. Regardez `org.hibernate.test.DoubleStringType` pour voir ce qu'il est possible de faire.

```
<property name="twoStrings" type="org.hibernate.test.DoubleStringType">
    <column name="first_string"/>
    <column name="second_string"/>
</property>
```

Remarquez l'utilisation des tags `<column>` pour mapper une propriété sur des colonnes multiples.

Les interfaces `CompositeUserType`, `EnhancedUserType`, `UserCollectionType`, et `UserVersionType` permettent des utilisations plus spécialisées.

Vous pouvez même donner des paramètres en indiquant `UserType` dans le fichier de mapping ; Pour cela, votre `UserType` doit implémenter l'interface `org.hibernate.usertype.ParameterizedType`. Pour spécifier des paramètres dans votre type propre, vous pouvez utiliser l'élément `<type>` dans vos fichiers de mapping.

```
<property name="priority">
    <type name="com.mycompany.usertypes.DefaultValueIntegerType">
        <param name="default">0</param>
    </type>
</property>
```

Le `UserType` permet maintenant de récupérer la valeur pour le paramètre nommé `default` à partir de l'objet `Properties` qui lui est passé.

Si vous utilisez fréquemment un `UserType`, cela peut être utile de lui définir un nom plus court. Vous pouvez faire cela en utilisant l'élément `<typedef>`. Les typedefs permettent d'assigner un nom à votre type propre et peuvent aussi contenir une liste de valeurs de paramètres par défaut si ce type est paramétré.

```
<typedef class="com.mycompany.usertypes.DefaultValueIntegerType" name="default_zero">
    <param name="default">0</param>
</typedef>
```

```
<property name="priority" type="default_zero"/>
```

Il est aussi possible de redéfinir les paramètres par défaut du typedef au cas par cas en utilisant des paramètres type sur le mapping de la propriété.

Bien que le fait que Hibernate propose de base une riche variété de types, et qu'il supporte les composants signifie que vous aurez très rarement *besoin* d'utiliser un nouveau type propre, il est néanmoins de bonne

pratique d'utiliser des types propres pour les classes (non entités) qui apparaissent fréquemment dans votre application. Par exemple une classe `MonetaryAmount` est un bon candidat pour un `CompositeUserType` même s'il pourrait facilement être mappé comme un composant. Une motivation pour cela est l'abstraction. Avec un type propre vos documents de mapping sont à l'abri des changements futurs dans votre façon de représenter des valeurs monétaires.

5.3. Mapper une classe plus d'une fois

Il est possible de proposer plus d'un mapping par classe persistante. Dans ce cas, vous devez spécifier un *nom d'entité* pour lever l'ambiguité entre les instances des entités mappées (par défaut, le nom de l'entité est celui de la classe). Hibernate vous permet de spécifier le nom de l'entité lorsque vous utilisez des objets persistants, lorsque vous écrivez des requêtes ou quand vous mappez des associations vers les entités nommées.

```
<class name="Contract" table="Contracts"
        entity-name="CurrentContract">
    ...
    <set name="history" inverse="true"
            order-by="effectiveEndDate desc">
        <key column="currentContractId"/>
        <one-to-many entity-name="HistoricalContract"/>
    </set>
</class>

<class name="Contract" table="ContractHistory"
        entity-name="HistoricalContract">
    ...
    <many-to-one name="currentContract"
            column="currentContractId"
            entity-name="CurrentContract"/>
</class>
```

Remarquez comment les associations sont désormais spécifiées en utilisant `entity-name` au lieu de `class`.

5.4. SQL quoted identifiers

Vous pouvez forcer Hibernate à mettre un identifiant entre quotes dans le SQL généré en mettant le nom de la table ou de la colonne entre backticks dans le document de mapping. Hibernate utilisera les bons styles de quotes pour le `Dialect` SQL (habituellement des doubles quotes, mais des parenthèses pour SQL server et des backticks pour MySQL).

```
<class name="LineItem" table="`Line Item`">
    <id name="id" column="`Item Id`"/><generator class="assigned"/></id>
    <property name="itemNumber" column="`Item #`"/>
    ...
</class>
```

5.5. alternatives Metadata

XML ne convient pas à tout le monde, il y a donc des moyens alternatifs pour définir des metatda de mappings O/R dans Hibernate.

5.5.1. utilisation de XDoclet

De nombreux utilisateurs de Hibernate préfèrent embarquer les informations de mappings directement au sein

du code source en utilisant les tags XDoclet `@hibernate.tags`. Nous ne couvrons pas cette approche dans ce document cependant, puisque c'est considéré comme faisant partie de XDoclet. Cependant, nous présentons l'exemple suivant de la classe `Cat` avec des mappings XDoclet.

```
package eg;
import java.util.Set;
import java.util.Date;

/**
 * @hibernate.class
 *  table="CATS"
 */
public class Cat {
    private Long id; // identifier
    private Date birthdate;
    private Cat mother;
    private Set kittens
    private Color color;
    private char sex;
    private float weight;

    /*
     * @hibernate.id
     *  generator-class="native"
     *  column="CAT_ID"
     */
    public Long getId() {
        return id;
    }
    private void setId(Long id) {
        this.id=id;
    }

    /**
     * @hibernate.many-to-one
     *  column="PARENT_ID"
     */
    public Cat getMother() {
        return mother;
    }
    void setMother(Cat mother) {
        this.mother = mother;
    }

    /**
     * @hibernate.property
     *  column="BIRTH_DATE"
     */
    public Date getBirthdate() {
        return birthdate;
    }
    void setBirthdate(Date date) {
        birthdate = date;
    }
    /**
     * @hibernate.property
     *  column="WEIGHT"
     */
    public float getWeight() {
        return weight;
    }
    void setWeight(float weight) {
        this.weight = weight;
    }

    /**
     * @hibernate.property
     *  column="COLOR"
     *  not-null="true"
     */
```

```
    public Color getColor() {
        return color;
    }
    void setColor(Color color) {
        this.color = color;
    }
    /**
     * @hibernate.set
     *  inverse="true"
     *  order-by="BIRTH_DATE"
     * @hibernate.collection-key
     *  column="PARENT_ID"
     * @hibernate.collection-one-to-many
     */
    public Set getKittens() {
        return kittens;
    }
    void setKittens(Set kittens) {
        this.kittens = kittens;
    }
    // addKitten not needed by Hibernate
    public void addKitten(Cat kitten) {
        kittens.add(kitten);
    }

    /**
     * @hibernate.property
     *  column="SEX"
     *  not-null="true"
     *  update="false"
     */
    public char getSex() {
        return sex;
    }
    void setSex(char sex) {
        this.sex=sex;
    }
}
```

Voyez le site web de Hibernate pour plus d'exemples sur XDoclet et Hibernate.

5.5.2. Utilisation des annotations JDK 5.0

Le JDK 5.0 introduit des annotations proches de celles de XDoclet au niveau java, qui sont type-safe et vérifiées à la compilation. Ce mécanisme est plus puissant que XDoclet et mieux supporté par les outils et IDE. IntelliJ IDEA, par exemple, supporte l'auto-complétion et le surlignement syntaxique des annotations JDK 5.0. La nouvelle révision des spécifications des EJB (JSR-220) utilise les annotations JDK 5.0 comme mécanisme primaire pour les meta-données des beans entités. Hibernate3 implémente l'`EntityManager` de la JSR-220 (API de persistance), le support du mapping de meta-données est disponible via le package *Hibernate Annotations*, en tant que module séparé à télécharger. EJB3 (JSR-220) et les métadata Hibernate3 sont supportés.

Ceci est un exemple d'une classe POJO annotée comme un EJB entité :

```
@Entity(access = AccessType.FIELD)
public class Customer implements Serializable {

    @Id;
    Long id;

    String firstName;
    String lastName;
    Date birthday;

    @Transient
    Integer age;
```

```
    @Embedded
    private Address homeAddress;

    @OneToMany(cascade=CascadeType.ALL)
    @JoinColumn(name="CUSTOMER_ID")
    Set<Order> orders;

    // Getter/setter and business methods
}
```

Notez que le support des annotations JDK 5.0 (et de la JSR-220) est encore en cours et n'est pas terminé. Référez vous au module Hibernate Annotation pour plus de détails.

5.6. Propriétés générées

Les propriétés générées sont des propriétés dont les valeurs sont générées par la base de données. Typiquement, les applications Hibernate avaient besoin d'invoquer `refresh` sur les instances qui contenaient des propriétés pour lesquelles la base de données générait des valeurs. Marquer les propriétés comme générées permet à l'application de déléguer cette responsabilité à Hibernate. Principalement, à chaque fois qu'Hibernate réalise une insertion ou une mise à jour en base de données pour une entité marquée comme telle, cela provoque immédiatement un select pour récupérer les valeurs générées.

Les propriétés marquées comme générées doivent de plus ne pas être insérables et modifiables Seuls Section 5.1.7, « version (optionnel) », Section 5.1.8, « timestamp (optionnel) », et Section 5.1.9, « property » peuvent être marqués comme générées.

`never` (par défaut) - indique la valeur de la propriété n'est pas générée dans la base de données.

`insert` - indique que la valeur de la propriété donnée est générée à l'insertion mais pas lors des futures mises à jour de l'enregistrement. Les colonnes de type "date de création" sont le cas d'utilisation typique de cette option. Notez que même les propriétés Section 5.1.7, « version (optionnel) » et Section 5.1.8, « timestamp (optionnel) » peuvent être déclarées comme générées, cette option n'est pas disponible à cet endroit...

`always` - indique que la valeur de la propriété est générée à l'insert comme aux updates.

5.7. Objets auxiliaires de la base de données

Permettent les ordres CREATE et DROP d'objets arbitraire de la base de donnéées, en conjonction avec les outils Hibernate d'évolutions de schéma, pour permettre de définir complètement un schéma utilisateur au sein des fichiers de mapping Hibernate. Bien que conçu spécifiquement pour créer et supprimer des objets tels que des triggers et des procédures stockées, ou toute commande pouvant être exécutée via une méthode de `java.sql.Statement.execute()` (ALTERs, INSERTS, etc). Il y a principalement deux modes pour définir les objets auxiliaires de base de données...

Le premier mode est de lister explicitement les commandes CREATE et DROP dans le fichier de mapping:

```
<hibernate-mapping>
    ...
    <database-object>
        <create>CREATE TRIGGER my_trigger ...</create>
        <drop>DROP TRIGGER my_trigger</drop>
    </database-object>
</hibernate-mapping>
```

Le second mode est de fournir une classe particulière qui connait comment construire les commandes CREATE et DROP. Cette classe particulière doit implémenter l'interface `org.hibernate.mapping.AuxiliaryDatabaseObject`.

```
<hibernate-mapping>
    ...
    <database-object>
        <definition class="MyTriggerDefinition"/>
    </database-object>
</hibernate-mapping>
```

Additionnellement, ces objets de base de données peuvent être optionnellement traités selon l'utilisation de dialectes particuliers..

```
<hibernate-mapping>
    ...
    <database-object>
        <definition class="MyTriggerDefinition"/>
        <dialect-scope name="org.hibernate.dialect.Oracle9Dialect"/>
        <dialect-scope name="org.hibernate.dialect.OracleDialect"/>
    </database-object>
</hibernate-mapping>
```

Chapitre 6. Mapping des collections

6.1. Collections persistantes

Hibernate requiert que les champs contenant des collections persistantes soient déclarés comme des types d'interface, par exemple :

```
public class Product {
    private String serialNumber;
    private Set parts = new HashSet();

    public Set getParts() { return parts; }
    void setParts(Set parts) { this.parts = parts; }
    public String getSerialNumber() { return serialNumber; }
    void setSerialNumber(String sn) { serialNumber = sn; }
}
```

L'interface réelle devrait être `java.util.Set`, `java.util.Collection`, `java.util.List`, `java.util.Map`, `java.util.SortedSet`, `java.util.SortedMap` ou ... n'importe quoi d'autre ! (Où "n'importe quoi d'autre" signifie que vous devrez écrire une implémentation de `org.hibernate.usertype.UserCollectionType`.)

Notez comment nous avons initialisé les variables d'instance avec une instance de `HashSet`. C'est le meilleur moyen pour initialiser les collections d'instances nouvellement créées (non persistantes). Quand nous fabriquons l'instance persistante - en appelant `persist()`, par exemple - Hibernate remplacera réellement le `HashSet` avec une instance d'une implémentation propre à Hibernate de `Set`. Prenez garde aux erreurs :

```
Cat cat = new DomesticCat();
Cat kitten = new DomesticCat();
....
Set kittens = new HashSet();
kittens.add(kitten);
cat.setKittens(kittens);
session.persist(cat);
kittens = cat.getKittens(); // Ok, la collection kittens est un Set
(HashSet) cat.getKittens(); // Erreur !
```

Les collections persistantes injectées par Hibernate se comportent de la même manière que `HashMap`, `HashSet`, `TreeMap`, `TreeSet` ou `ArrayList`, selon le type de l'interface.

Les instances des collections ont le comportement habituel des types des valeurs. Elles sont automatiquement persistées quand elles sont référencées par un objet persistant et automatiquement effacées quand elles sont déréférencées. Si une collection est passée d'un objet persistant à un autre, ses éléments pourraient être déplacés d'une table à une autre. Deux entités ne peuvent pas partager une référence vers une même instance d'une collection. Dû au modèle relationnel sous-jacent, les propriétés contenant des collections ne supportent pas la sémantique de la valeur null ; Hibernate ne distingue pas une référence vers une collection nulle d'une collection vide.

Vous ne devriez pas vous préoccuper trop de ça. Utilisez les collections persistantes de la même manière que vous utilisez des collections Java ordinaires. Assurez-vous de comprendre la sémantique des associations bidirectionnelles (traitée plus loin).

6.2. Mapper une collection

L'élément de mapping d'Hibernate utilisé pour mapper une collection dépend du type de l'interface. Par

exemple, un élément <set> est utilisé pour mapper des propriétés de type Set.

```
<class name="Product">
    <id name="serialNumber" column="productSerialNumber"/>
    <set name="parts">
        <key column="productSerialNumber" not-null="true"/>
        <one-to-many class="Part"/>
    </set>
</class>
```

À part <set>, il y aussi les éléments de mapping <list>, <map>, <bag>, <array> et <primitive-array>. L'élément <map> est représentatif :

```
<map
    name="nomDePropriete"                                          (1)
    table="nom_de_table"                                           (2)
    schema="nom_du_schema"                                         (3)
    lazy="true|extra|false"                                        (4)
    inverse="true|false"                                           (5)
    cascade="all|none|save-update|delete|all-delete-orphan"        (6)
    sort="unsorted|natural|ClasseDeComparateur"                    (7)
    order-by="nom_de_column asc|desc"                              (8)
    where="condition sql where quelcconque"                        (9)
    fetch="join|select|subselect"                                  (10)
    batch-size="N"                                                 (11)
    access="field|property|NomDeClasse"                            (12)
    optimistic-lock="true|false"                                   (13)
    mutable="true|false"                                           (14)
    node="nom-d-element|."
    embed-xml="true|false"
>

    <key .... />
    <map-key .... />
    <element .... />
</map>
```

(1) name : le nom de la propriété contenant la collection

(2) table (optionnel - par défaut = nom de la propriété) : le nom de la table de la collection (non utilisé pour les associations one-to-many)

(3) schema (optionnel) : le nom du schéma pour surcharger le schéma déclaré dans l'élément racine

(4) lazy (optionnel - par défaut = true) : peut être utilisé pour désactiver l'initialisation tardive et spécifier que l'association est toujours rapportée, ou pour activer la récupération extra-paresseuse (NdT : extra-lazy) où la plupart des opérations n'initialisent pas la collection (approprié pour de très grosses collections)

(5) inverse (optionnel - par défaut = false) : définit cette collection comme l'extrêmité "inverse" de l'association bidirectionnelle

(6) cascade (optionnel - par défaut = none) : active les opérations de cascade vers les entités filles

(7) sort (optionnel) : spécifie une collection triée via un ordre de tri naturel, ou via une classe comparateur donnée (implémentant Comparator)

(8) order-by (optionnel, seulement à partir du JDK1.4) : spécifie une colonne de table (ou des colonnes) qui définit l'ordre d'itération de Map, Set ou Bag, avec en option asc ou desc

(9) where (optionnel) : spécifie une condition SQL arbitraire WHERE à utiliser au chargement ou à la suppression d'une collection (utile si la collection ne doit contenir qu'un sous ensemble des données disponibles)

(10) fetch (optionnel, par défaut = select) : à choisir entre récupération par jointures externes, récupération par selects séquentiels, et récupération par sous-selects séquentiels

(11) batch-size (optionnel, par défaut = 1) : une taille de batch (batch size) utilisée pour charger plusieurs instances de cette collection en initialisation tardive

(12) access (optionnel - par défaut = property) : La stratégie qu'Hibernate doit utiliser pour accéder à la

valeur de la propriété

(13) `optimistic-lock` (optionnel - par défaut = `true`) : spécifie que changer l'état de la collection entraîne l'incrémentation de la version appartenant à l'entité (Pour une association un vers plusieurs, il est souvent raisonnable de désactiver ce paramètre)

(14) `mutable` (optionnel - par défaut = `true`) : une valeur à `false` spécifie que les éléments de la collection ne changent jamais (une optimisation mineure dans certains cas)

6.2.1. Les clefs étrangères d'une collection

Les instances d'une collection sont distinguées dans la base par la clef étrangère de l'entité qui possède la collection. Cette clef étrangère est référencée comme la(es) *colonne(s) de la clef de la collection* de la table de la collection. La colonne de la clef de la collection est mappée par l'élément `<key>`.

Il peut y avoir une contrainte de nullité sur la colonne de la clef étrangère. Pour les associations unidirectionnelles un vers plusieurs, la colonne de la clef étrangère peut être nulle par défaut, donc vous pourriez avoir besoin de spécifier `not-null="true"`.

```
<key column="productSerialNumber" not-null="true"/>
```

La contraite de la clef étrangère peut utiliser `ON DELETE CASCADE`.

```
<key column="productSerialNumber" on-delete="cascade"/>
```

Voir le chapitre précédent pour une définition complète de l'élément `<key>`.

6.2.2. Les éléments d'une collection

Les collections peuvent contenir la plupart des autres types Hibernate, dont tous les types basiques, les types utilisateur, les composants, et bien sûr, les références vers d'autres entités. C'est une distinction importante : un objet dans une collection pourrait être géré avec une sémantique de "valeur" (sa durée de vie dépend complètement du propriétaire de la collection) ou il pourrait avoir une référence vers une autre entité, avec sa propre durée de vie. Dans le dernier cas, seul le "lien" entre les 2 objets est considéré être l'état retenu par la collection.

Le type contenu est référencé comme le *type de l'élément de la collection*. Les éléments de la collections sont mappés par `<element>` ou `<composite-element>`, ou dans le cas des références d'entité, avec `<one-to-many>` ou `<many-to-many>`. Les deux premiers mappent des éléments avec un sémantique de valeur, les deux suivants sont utilisés pour mapper des associations d'entité.

6.2.3. Collections indexées

Tous les mappings de collection, exceptés ceux avec les sémantiques d'ensemble (NdT : set) et de sac (NdT : bag), ont besoin d'une *colonne d'index* dans la table de la collection - une colonne qui mappe un index de tableau, ou un index de `List`, ou une clef de `Map`. L'index d'une `Map` peut être n'importe quel type basique, mappé avec `<map-key>`, ça peut être une référence d'entité mappée avec `<map-key-many-to-many>`, ou ça peut être un type composé, mappé avec `<composite-map-key>`. L'index d'un tableau ou d'une liste est toujours de type `integer` et est mappé en utilisant l'élément `<list-index>`. Les colonnes mappées contiennent des entiers séquentiels (numérotés à partir de zéro par défaut).

```
<list-index
        column="nom_de_colonne"            (1)
        base="0|1|..."/>
```

(1) `nom_de_colonne` (requis) : le nom de la colonne contenant les valeurs de l'index de la collection
(1) `base` (optionnel, par défaut = `0`) : la valeur de la colonne de l'index qui correspond au premier élément de la liste ou du tableau

```
<map-key
        column="nom_de_colonne"             (1)
        formula="n'importe quelle expression(2) SQL"
        type="nom_du_type"                  (3)
        node="@nom-d-attribut"
        length="N"/>
```

(1) `column` (optionnel) : le nom de la colonne contenant les valeurs de l'index de la collection
(2) `formula` (optionnel) : une formule SQL utilisée pour évaluer la clef de la map
(3) `type` (reguis): le type des clefs de la map

```
<map-key-many-to-many
        column="nom_de_colonne"             (1)
        formula="n'importe quelle expression(2)(3) SQL"
        class="NomDeClasse"
/>
```

(1) `column` (optionnel) : le nom de la colonne de la clef étrangère pour les valeurs de l'index de la collection
(2) `formula` (optionnel) : une formulre SQL utilisée pour évaluer la clef étrangère de la clef de la map
(3) `class` (requis): la classe de l'entité utilisée comme clef de la map

Si votre table n'a pas de colonne d'index, et que vous souhaitez tout de même utiliser `List` comme type de propriété, vous devriez mapper la propriété comme un *<bag>* Hibernate. Un sac (NdT : bag) ne garde pas son ordre quand il est récupéré de la base de données, mais il peut être optionnellement trié ou ordonné.

Il y a pas mal de variétés de mappings qui peuvent être générés pour les collections, couvrant beaucoup des modèles relationnels communs. Nous vous suggérons d'expérimenter avec l'outil de génération de schéma pour avoir une idée de comment traduire les différentes déclarations de mapping vers des table de la base de données.

6.2.4. Collections de valeurs et associations plusieurs-vers-plusieurs

N'importe quelle collection de valeurs ou association plusieurs-vers-plusieurs requiert une *table de collection* avec une(des) colonne(s) de clef étrangère, une(des) *colonne(s) d'élément de la collection* ou des colonnes et possiblement une(des) colonne(s) d'index.

Pour une collection de valeurs, nous utilisons la balise `<element>`.

```
<element
        column="nom_de_colonne"                 (1)
        formula="n'importe quelle expression SQL"(2)
        type="nomDeType"                        (3)
        length="L"
        precision="P"
        scale="S"
        not-null="true|false"
        unique="true|false"
        node="nom-d-element"
/>
```

(1) `column` (optionnel) : le nom de la colonne contenant les valeurs de l'élément de la collection
(2) `formula` (optionnel) : une formule SQL utilisée pour évaluer l'élément
(3) `type` (requis) : le type de l'élément de la collection

Une *association plusieurs-vers-plusieurs* est spécifiée en utilisant l'élément `<many-to-many>`.

```
<many-to-many
        column="nom_de_colonne"                              (1)
        formula="n'importe quelle expression SQL"            (2)
        class="NomDeClasse"                                  (3)
        fetch="select|join"                                  (4)
        unique="true|false"                                  (5)
        not-found="ignore|exception"                         (6)
        entity-name="NomDEntite"                             (7)
        property-ref="nomDeProprieteDeLaClasseAssociee"      (8)
        node="nom-d-element"
        embed-xml="true|false"
    />
```

(1) `column` (optionnel) : le nom de la colonne de la clef étrangère de l'élément

(2) `formula` (optionnel) : une formule SQL utilisée pour évaluer la valeur de la clef étrangère de l'élément

(3) `class` (requis) : le nom de la classe associée

(4) `fetch` (optionnel - par défaut `join`) : active les récupérations par jointures externes ou par selects séquentiels pour cette association. C'est un cas spécial ; pour une récupération complète sans attente (dans un seul `SELECT`) d'une entité et de ses relations plusieurs-vers-plusieurs vers d'autres entités, vous devriez activer la récupération `join` non seulement sur la collection elle-même, mais aussi avec cet attribut sur l'élément imbriqué `<many-to-many>`.

(5) `unique` (optionnel) : activer la génération DDL d'une contrainte d'unicité pour la colonne de la clef étrangère. Ça rend la pluralité de l'association effectivement un-vers-plusieurs.

(6) `not-found` (optionnel - par défaut `exception`) : spécifie comment les clefs étrangères qui référencent la lignes manquantes seront gérées : `ignore` traitera une ligne manquante comme une association nulle.

(7) `entity-name` (optionnel) : le nom de l'entité de la classe associée, comme une alternative à `class`

(8) `property-ref` (optionnel) : le nom d'une propriété de la classe associée qui est jointe à cette clef étrangère. Si non spécifiée, la clef primaire de la classe associée est utilisée.

Quelques exemples, d'abord, un ensemble de chaînes de caractères :

```
<set name="names" table="person_names">
    <key column="person_id"/>
    <element column="person_name" type="string"/>
</set>
```

Un bag contenant des entiers (avec un ordre d'itération déterminé par l'attribut `order-by`) :

```
<bag name="sizes"
        table="item_sizes"
        order-by="size asc">
    <key column="item_id"/>
    <element column="size" type="integer"/>
</bag>
```

Un tableau d'entités - dans ce cas, une association plusieurs-vers-plusieurs :

```
<array name="addresses"
        table="PersonAddress"
        cascade="persist">
    <key column="personId"/>
    <list-index column="sortOrder"/>
    <many-to-many column="addressId" class="Address"/>
</array>
```

Une map de chaînes de caractères vers des dates :

```
<map name="holidays"
        table="holidays"
        schema="dbo"
        order-by="hol_name asc">
```

```
    <key column="id"/>
    <map-key column="hol_name" type="string"/>
    <element column="hol_date" type="date"/>
</map>
```

Une liste de composants (discute dans le prochain chapitre) :

```
<list name="carComponents"
        table="CarComponents">
    <key column="carId"/>
    <list-index column="sortOrder"/>
    <composite-element class="CarComponent">
        <property name="price"/>
        <property name="type"/>
        <property name="serialNumber" column="serialNum"/>
    </composite-element>
</list>
```

6.2.5. Association un-vers-plusieurs

Une *association un vers plusieurs* lie les tables de deux classes par une clef étrangère, sans l'intervention d'une table de collection. Ce mapping perd certaines sémantiques des collections Java normales :

- Une instance de la classe de l'entité contenue ne peut pas appartenir à plus d'une instance de la collection
- Une instance de la classe de l'entité contenue ne peut pas apparaître plus plus d'une valeur d'index de la collection

Une association de `Product` vers `Part` requiert l'existence d'une clef étrangère et possiblement une colonne d'index pour la table `Part`. Une balise `<one-to-many>` indique que c'est une association un vers plusieurs.

```
<one-to-many
        class="NomDeClasse"                                (1)
        not-found="ignore|exception"                       (2)
        entity-name="NomDEntite"                           (3)
        node="nom-d-element"
        embed-xml="true|false"
    />
```

(1) `class` (requis) : le nom de la classe associée

(2) `not-found` (optionnel - par défaut `exception`) : spécifie comment les identifiants cachés qui référencent des lignes manquantes seront gérés : `ignore` traitera une ligne manquante comme une association nulle

(3) `entity-name` (optionnel) : le nom de l'entité de la classe associée, comme une alternative à `class`.

Notez que l'élément `<one-to-many>` n'a pas besoin de déclarer de colonnes. Il n'est pas non plus nécessaire de spécifier le nom de la table nulle part.

Note très importante : si la colonne de la clef d'une association `<one-to-many>` est déclarée `NOT NULL`, vous devez déclarer le mapping de `<key>` avec `not-null="true"` ou *utiliser une association bidirectionnelle* avec le mapping de la collection marqué `inverse="true"`. Voir la discussion sur les associations bidirectionnelles plus tard dans ce chapitre.

Cet exemple montre une map d'entités `Part` par nom (où `partName` est une propriété persistante de `Part`). Notez l'utilisation d'un index basé sur une formule.

```
<map name="parts"
        cascade="all">
    <key column="productId" not-null="true"/>
    <map-key formula="partName"/>
    <one-to-many class="Part"/>
```

```
</map>
```

6.3. Mappings de collection avancés

6.3.1. Collections triées

Hibernate supporte des collections implémentant `java.util.SortedMap` et `java.util.SortedSet`. Vous devez spécifier un comparateur dans le fichier de mapping :

```
<set name="aliases"
            table="person_aliases"
            sort="natural">
    <key column="person"/>
    <element column="name" type="string"/>
</set>

<map name="holidays" sort="my.custom.HolidayComparator">
    <key column="year_id"/>
    <map-key column="hol_name" type="string"/>
    <element column="hol_date" type="date"/>
</map>
```

Les valeurs permises pour l'attribut `sort` sont `unsorted`, `natural` et le nom d'une classe implémentant `java.util.Comparator`.

Les collections triées se comportent réellement comme `java.util.TreeSet` ou `java.util.TreeMap`.

Si vous voulez que la base de données elle-même ordonne les éléments de la collection, utilisez l'attribut `order-by` des mappings `set`, `bag` ou `map`. Cette solution est seulement disponible à partir du JDK 1.4 (c'est implémenté en utilisant `LinkedHashSet` ou `LinkedHashMap`). Ceci exécute le tri dans la requête SQL, pas en mémoire.

```
<set name="aliases" table="person_aliases" order-by="lower(name) asc">
    <key column="person"/>
    <element column="name" type="string"/>
</set>

<map name="holidays" order-by="hol_date, hol_name">
    <key column="year_id"/>
    <map-key column="hol_name" type="string"/>
    <element column="hol_date type="date"/>
</map>
```

Notez que la valeur de l'attribut `order-by` est un ordre SQL, pas un ordre HQL !

Les associations peuvent même être triées sur des critères arbitraires à l'exécution en utilisant un `filter()` de collection.

```
sortedUsers = s.createFilter( group.getUsers(), "order by this.name" ).list();
```

6.3.2. Associations bidirectionnelles

Une *association bidirectionnelle* permet une navigation à partir de la "fin" de l'association. Deux sortes d'associations bidirectionnelles sont supportées :

un-vers-plusieurs (NdT : one-to-many)
    ensemble ou sac à une extrémité, une seule valeur à l'autre

plusieurs-vers-plusieurs (NdT : many-to-many)
    ensemble ou sac aux deux extrémités

Vous pouvez spécifier une association plusieurs-vers-plusieurs bidirectionnelle simplement en mappant deux associations plusieurs-vers-plusieurs vers la même table de base de données et en déclarant une extrémité comme *inverse* (celle de votre choix, mais ça ne peut pas être une collection indexée).

Voici un exemple d'association bidirectionnelle plusieurs-vers-plusieurs ; chaque catégorie peut avoir plusieurs objets et chaque objet peut être dans plusieurs catégories :

```
<class name="Category">
    <id name="id" column="CATEGORY_ID"/>
    ...
    <bag name="items" table="CATEGORY_ITEM">
        <key column="CATEGORY_ID"/>
        <many-to-many class="Item" column="ITEM_ID"/>
    </bag>
</class>

<class name="Item">
    <id name="id" column="CATEGORY_ID"/>
    ...

    <!-- inverse end -->
    <bag name="categories" table="CATEGORY_ITEM" inverse="true">
        <key column="ITEM_ID"/>
        <many-to-many class="Category" column="CATEGORY_ID"/>
    </bag>
</class>
```

Les changements faits uniquement sur l'extréminté inverse de l'association *ne sont pas* persistés. Ceci signifie qu'Hibernate a deux représentations en mémoire pour chaque association bidirectionnelles, un lien de A vers B et un autre de B vers A. C'est plus facile à comprendre si vous pensez au modèle objet de Java et comment nous créons une relation plusieurs-vers-plusieurs en Java :

```
category.getItems().add(item);          // La catégorie est maintenant "au courant" de la relation
item.getCategories().add(category);     // L'objet est maintenant "au courant" de la relation

session.persist(item);                   // La relation ne sera pas sauvegardée !
session.persist(category);               // La relation sera sauvegardée
```

La partie non-inverse est utilisée pour sauvegarder la représentation en mémoire dans la base de données.

Vous pouvez définir une association un-vers-plusieurs bidirectionnelle en mappant une association un-vers-plusieurs vers la(es) même(s) colonne(s) de table qu'une association plusieurs-vers-un et en déclarant l'extrémité pluri-valuée `inverse="true"`.

```
<class name="Parent">
    <id name="id" column="parent_id"/>
    ....
    <set name="children" inverse="true">
        <key column="parent_id"/>
        <one-to-many class="Child"/>
    </set>
</class>

<class name="Child">
    <id name="id" column="child_id"/>
```

```
    ....
    <many-to-one name="parent"
        class="Parent"
        column="parent_id"
        not-null="true"/>
</class>
```

Mapper une extrémité d'une association avec `inverse="true"` n'affecte pas l'opération de cascades, ce sont des concepts orthogonaux !

6.3.3. Associations bidirectionnelles avec des collections indexées

Une association bidirectionnelle où une extrémité est représentée comme une `<list>` ou une `<map>` requiert une considération spéciale. Si il y a une propriété de la classe enfant qui mappe la colonne de l'index, pas de problème, nous pouvons continuer à utiliser `inverse="true"` sur le mapping de la collection :

```
<class name="Parent">
    <id name="id" column="parent_id"/>
    ....
    <map name="children" inverse="true">
        <key column="parent_id"/>
        <map-key column="name"
            type="string"/>
        <one-to-many class="Child"/>
    </map>
</class>

<class name="Child">
    <id name="id" column="child_id"/>
    ....
    <property name="name"
        not-null="true"/>
    <many-to-one name="parent"
        class="Parent"
        column="parent_id"
        not-null="true"/>
</class>
```

Mais, si il n'y a pas de telle prorpriété sur la classe enfant, nous ne pouvons pas penser à l'association comme vraiment bidirectionnelle (il y a des informations disponibles à une extrémité de l'association qui ne sont pas disponibles à l'autre extrémité). Dans ce cas, nous ne pouvons pas mapper la collection `inverse="true"`. À la place, nous pourrions utiliser le mapping suivant :

```
<class name="Parent">
    <id name="id" column="parent_id"/>
    ....
    <map name="children">
        <key column="parent_id"
            not-null="true"/>
        <map-key column="name"
            type="string"/>
        <one-to-many class="Child"/>
    </map>
</class>

<class name="Child">
    <id name="id" column="child_id"/>
    ....
    <many-to-one name="parent"
        class="Parent"
        column="parent_id"
        insert="false"
        update="false"
        not-null="true"/>
```

```
</class>
```

Notez que dans ce mapping, l'extrémité de l'association contenant la collection est responsable des mises à jour de la clef étrangère. À faire : cela entraîne-t-il réellement des expressions updates inutiles ?

6.3.4. Associations ternaires

Il y a trois approches possibles pour mapper une association ternaire. L'une est d'utiliser une `Map` avec une association en tant qu'index :

```
<map name="contracts">
    <key column="employer_id" not-null="true"/>
    <map-key-many-to-many column="employee_id" class="Employee"/>
    <one-to-many class="Contract"/>
</map>
```

```
<map name="connections">
    <key column="incoming_node_id"/>
    <map-key-many-to-many column="outgoing_node_id" class="Node"/>
    <many-to-many column="connection_id" class="Connection"/>
</map>
```

Une seconde approche est simplement de remodeler l'association comme une classe d'entité. C'est l'approche la plus commune.

Une alternative finale est d'utiliser des éléments composites, dont nous discuterons plus tard.

6.3.5. Utiliser un `<idbag>`

Si vous embrassez pleinement notre vue que les clefs composées sont une mauvaise chose et que des entités devraient avoir des identifiants artificiels (des clefs subrogées), alors vous pourriez trouver un peu curieux que les associations plusieurs-vers-plusieurs et les collections de valeurs que nous avons montré jusqu'ici mappent toutes des tables avec des clefs composées ! Maintenant, ce point est assez discutable ; une table d'association pure ne semble pas beaucoup bénéficier d'une clef subrogée (bien qu'une collection de valeur composées le *pourrait*). Néanmoins, Hibernate fournit une foncionnalité qui vous permet de mapper des associations plusieurs-vers-plusieurs et des collections de valeurs vers une table avec une clef subrogée.

L'élément `<idbag>` vous laisse mapper une `List` (ou une `Collection`) avec une sémantique de sac.

```
<idbag name="lovers" table="LOVERS">
    <collection-id column="ID" type="long">
        <generator class="sequence"/>
    </collection-id>
    <key column="PERSON1"/>
    <many-to-many column="PERSON2" class="Person" fetch="join"/>
</idbag>
```

Comme vous pouvez voir, un `<idbag>` a un généréteur d'id artificiel, comme une classe d'entité ! Une clef subrogée différente est assignée à chaque ligne de la collection. Cependant, Hibernate ne fournit pas de mécanisme pour découvrir la valeur d'une clef subrogée d'une ligne particulière.

Notez que les performances de la mise à jour d'un `<idbag>` sont *bien* meilleures qu'un `<bag>` ordinaire ! Hibernate peut localiser des lignes individuelles efficacement et les mettre à jour ou les effacer individuellement, comme une liste, une map ou un ensemble.

Dans l'implémentation actuelle, la stratégie de la génération de l'identifiant `native` n'est pas supportée pour les

identifiants de collection `<idbag>`.

6.4. Exemples de collections

Les sections précédentes sont assez confuses. Donc prenons un exemple. Cette classe :

```
package eg;
import java.util.Set;

public class Parent {
    private long id;
    private Set children;

    public long getId() { return id; }
    private void setId(long id) { this.id=id; }

    private Set getChildren() { return children; }
    private void setChildren(Set children) { this.children=children; }

    ....
    ....
}
```

a une collection d'instances de `Child`. Si chaque enfant a au plus un parent, le mapping le plus naturel est une association un-vers-plusieurs :

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children">
            <key column="parent_id"/>
            <one-to-many class="Child"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

Ceci mappe les définitions de tables suivantes :

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, name varchar(255), parent_id bigint )
alter table child add constraint childfk0 (parent_id) references parent
```

Si le parent est *requis*, utilisez une association un-vers-plusieurs unidirectionnelle :

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" inverse="true">
            <key column="parent_id"/>
            <one-to-many class="Child"/>
        </set>
```

```
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
        <many-to-one name="parent" class="Parent" column="parent_id" not-null="true"/>
    </class>

</hibernate-mapping>
```

Notez la contrainte `NOT NULL` :

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null
                     primary key,
                     name varchar(255),
                     parent_id bigint not null )
alter table child add constraint childfk0 (parent_id) references parent
```

Alternativement, si vous insistez absolument pour que cette association soit unidirectionnelle, vous pouvez déclarer la contrainte `NOT NULL` sur le mapping `<key>` :

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children">
            <key column="parent_id" not-null="true"/>
            <one-to-many class="Child"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

D'un autre côté, si un enfant pouvait avoir plusieurs parent, une association plusieurs-vers-plusieurs est plus appropriée :

```
<hibernate-mapping>

    <class name="Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" table="childset">
            <key column="parent_id"/>
            <many-to-many class="Child" column="child_id"/>
        </set>
    </class>

    <class name="Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>
```

```
</hibernate-mapping>
```

Définitions des tables :

```
create table parent ( id bigint not null primary key )
create table child ( id bigint not null primary key, name varchar(255) )
create table childset ( parent_id bigint not null,
                        child_id bigint not null,
                        primary key ( parent_id, child_id ) )
alter table childset add constraint childsetfk0 (parent_id) references parent
alter table childset add constraint childsetfk1 (child_id) references child
```

Pour plus d'exemples et une revue complète du mapping de la relation parent/enfant, voir see Chapitre 21, *Exemple : Père/Fils*.

Des mappings d'association plus exotiques sont possibles, nous cataloguerons toutes les possibilités dans le prochain chapitre.

Chapitre 7. Mapper les associations

7.1. Introduction

Correctement mapper les associations est souvent la tâche la plus difficile. Dans cette section nous traiterons les cas classiques les uns après les autres. Nous commencerons d'abbord par les mappings unidirectionnels, puis nous aborderons la question des mappings bidirectionnels. Nous illustrerons tous nos exemples avec les classes `Person` et `Address`.

Nous utiliserons deux critères pour classer les associations : le premier sera de savoir si l'association est bâti sur une table supplémentaire d'association et le deuxieme sera basé sur la multiplicité de cette association.

Autoriser une clé étrangère nulle est considéré comme un mauvais choix dans la construction d'un modèle de données. Nous supposerons donc que dans tous les exemples qui vont suivre on aura interdit la valeur nulle pour les clés étrangères. Attention, ceci ne veut pas dire que Hibernate ne supporte pas les clés étrangères pouvant prendre des valeurs nulles, les exemples qui suivent continueront de fonctionner si vous décidiez ne plus imposer la contrainte de non-nullité sur les clés étrangères.

7.2. Association unidirectionnelle

7.2.1. plusieurs à un

Une *association plusieurs-à-un (many-to-one) unidirectionnelle* est le type que l'on rencontre le plus souvent dans les associations unidirectionnelles.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null )
create table Address ( addressId bigint not null primary key )
```

7.2.2. un à un

une *association un-à-un (one-to-one) sur une clé étrangère* est presque identique. La seule différence est sur la contrainte d'unicité que l'on impose à cette colonne.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
```

```
    </id>
    <many-to-one name="address"
        column="addressId"
        unique="true"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```

Une *association un-à-un (one-to-one) unidirectionnelle sur une clé primaire* utilise un générateur d'identifiant particulier. (Remarquez que nous avons inversé le sens de cette association dans cet exemple.)

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
</class>

<class name="Address">
    <id name="id" column="personId">
        <generator class="foreign">
            <param name="property">person</param>
        </generator>
    </id>
    <one-to-one name="person" constrained="true"/>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( personId bigint not null primary key )
```

7.2.3. un à plusieurs

Une *association un-à-plusieurs (one-to-many) unidirectionnelle sur une clé étrangère* est vraiment inhabituelle, et n'est pas vraiment recommandée.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses">
        <key column="personId"
            not-null="true"/>
        <one-to-many class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
```

```
create table Person ( personId bigint not null primary key )
create table Address ( addressId bigint not null primary key, personId bigint not null )
```

Nous pensons qu'il est préférable d'utiliser une table de jointure pour ce type d'association.

7.3. Associations unidirectionnelles avec tables de jointure

7.3.1. un à plusieurs

Une *association unidirectionnelle un-à-plusieurs (one-to-many) avec une table de jointure* est un bien meilleur choix. Remarquez qu'en spécifiant `unique="true"`, on a changé la multiplicité plusieurs-à-plusieurs (many-to-many) pour un-à-plusieurs (one-to-many).

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            unique="true"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId not null, addressId bigint not null primary key )
create table Address ( addressId bigint not null primary key )
```

7.3.2. plusieurs à un

Une *assiociation plusieurs-à-un (many-to-one) unidirectionnelle sur une table de jointure* est très fréquente quand l'association est optionnelle.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true">
        <key column="personId" unique="true"/>
        <many-to-one name="address"
            column="addressId"
            not-null="true"/>
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
```

```
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key, addressId bigint not null )
create table Address ( addressId bigint not null primary key )
```

7.3.3. un à un

Une *association unidirectionnelle un-à-un (one-to-one) sur une table de jointure* est extrèmement rare mais envisageable.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true">
        <key column="personId"
            unique="true"/>
        <many-to-one name="address"
            column="addressId"
            not-null="true"
            unique="true"/>
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```

7.3.4. plusieurs à plusieurs

Finallement, nous avons *l'association unidirectionnelle plusieurs-à-plusieurs (many-to-many)*.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
</class>
```

```
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null, primary key (personI
create table Address ( addressId bigint not null primary key )
```

7.4. Associations bidirectionnelles

7.4.1. un à plusieurs / plusieurs à un

Une *association bidirectionnelle plusieurs à un (many-to-one)* est le type d'association que l'on rencontre le plus souvent. (c'est la façon standard de créer des relations parents/enfants.)

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <set name="people" inverse="true">
        <key column="addressId"/>
        <one-to-many class="Person"/>
    </set>
</class>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null )
create table Address ( addressId bigint not null primary key )
```

Si vous utilisez une `List` (ou toute autre collection indexée) vous devez paramétrer la colonne `key` de la clé étrangère à `not null`, et laisser Hibernate gérer l'association depuis l'extrémité collection pour maintenir l'index de chaque élément (rendant l'autre extrémité virtuellement inverse en paramétrant `update="false"` et `insert="false"`):

```
<class name="Person">
   <id name="id"/>
   ...
   <many-to-one name="address"
      column="addressId"
      not-null="true"
      insert="false"
      update="false"/>
</class>

<class name="Address">
   <id name="id"/>
   ...
   <list name="people">
      <key column="addressId" not-null="true"/>
      <list-index column="peopleIdx"/>
      <one-to-many class="Person"/>
   </list>
</class>
```

7.4.2. Un à un

Une *association bidirectionnelle un à un (one-to-one) sur une clé étrangère* est aussi très fréquente.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <many-to-one name="address"
        column="addressId"
        unique="true"
        not-null="true"/>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
   <one-to-one name="person"
        property-ref="address"/>
</class>
```

```
create table Person ( personId bigint not null primary key, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```

Une *association bidirectionnelle un-à-un (one-to-one) sur une clé primaire* utilise un générateur particulier d'id.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <one-to-one name="address"/>
</class>

<class name="Address">
    <id name="id" column="personId">
        <generator class="foreign">
            <param name="property">person</param>
        </generator>
    </id>
    <one-to-one name="person"
        constrained="true"/>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table Address ( personId bigint not null primary key )
```

7.5. Associations bidirectionnelles avec table de jointure

7.5.1. un à plusieurs / plusieurs à un

Une *association bidirectionnelle un-à-plusieurs (one-to-many) sur une table de jointure* . Remarquez que `inverse="true"` peut s'appliquer sur les deux extrémités de l' association, sur la collection, ou sur la jointure.

```
<class name="Person">
    <id name="id" column="personId">
```

```
        <generator class="native"/>
    </id>
    <set name="addresses"
        table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            unique="true"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        inverse="true"
        optional="true">
        <key column="addressId"/>
        <many-to-one name="person"
            column="personId"
            not-null="true"/>
    </join>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null primary key )
create table Address ( addressId bigint not null primary key )
```

7.5.2. Un à un

Une *association bidirectionnelle un-à-un (one-to-one) sur une table de jointure* est extrèmement rare mais envisageable.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true">
        <key column="personId"
            unique="true"/>
        <many-to-one name="address"
            column="addressId"
            not-null="true"
            unique="true"/>
    </join>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <join table="PersonAddress"
        optional="true"
        inverse="true">
        <key column="addressId"
            unique="true"/>
        <many-to-one name="person"
            column="personId"
            not-null="true"
            unique="true"/>
    </join>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null primary key, addressId bigint not null unique )
create table Address ( addressId bigint not null primary key )
```

7.5.3. plusieurs à plusieurs

Finallement nous avons *l'association bidirectionnelle plusieurs à plusieurs*.

```
<class name="Person">
    <id name="id" column="personId">
        <generator class="native"/>
    </id>
    <set name="addresses" table="PersonAddress">
        <key column="personId"/>
        <many-to-many column="addressId"
            class="Address"/>
    </set>
</class>

<class name="Address">
    <id name="id" column="addressId">
        <generator class="native"/>
    </id>
    <set name="people" inverse="true"  table="PersonAddress">
        <key column="addressId"/>
        <many-to-many column="personId"
            class="Person"/>
    </set>
</class>
```

```
create table Person ( personId bigint not null primary key )
create table PersonAddress ( personId bigint not null, addressId bigint not null, primary key (personI
create table Address ( addressId bigint not null primary key )
```

7.6. Des mappings plus complexes

Des associations encore plus complexes sont *extrêmement* rares. Hibernate permet de gérer des situations plus complexes en utilisant des parties SQL dans les fichiers de mapping. Par exemple, si une table avec l'historiques des informations d'un compte définit les colonnes `accountNumber`, `effectiveEndDate` et `effectiveStartDate`, mappées de telle sorte:

```
<properties name="currentAccountKey">
    <property name="accountNumber" type="string" not-null="true"/>
    <property name="currentAccount" type="boolean">
        <formula>case when effectiveEndDate is null then 1 else 0 end</formula>
    </property>
</properties>
<property name="effectiveEndDate" type="date"/>
<property name="effectiveStateDate" type="date" not-null="true"/>
```

alors nous pouvons mapper une association à l'instance *courante* (celle avec une `effectiveEndDate`) nulle en utilisant:

```
<many-to-one name="currentAccountInfo"
        property-ref="currentAccountKey"
        class="AccountInfo">
    <column name="accountNumber"/>
```

```
    <formula>'1'</formula>
</many-to-one>
```

Dans un exemple plus complexe, imaginez qu'une association entre `Employee` et `Organization` est gérée dans une table `Employment` pleines de données historiques. Dans ce cas, une association vers l'employeur *le plus récent* (celui avec la `startDate` la plus récente) pourrait être mappée comme cela:

```
<join>
    <key column="employeeId"/>
    <subselect>
        select employeeId, orgId
        from Employments
        group by orgId
        having startDate = max(startDate)
    </subselect>
    <many-to-one name="mostRecentEmployer"
            class="Organization"
            column="orgId"/>
</join>
```

Vous pouvez être créatif grace à ces possibilités, mais il est généralement plus pratique d'utiliser des requêtes HQL ou criteria dans ce genre de situation.

Chapitre 8. Mapping de composants

La notion de *composants* est réutilisé dans différents contextes, avec différents objectifs, à travers Hibernate.

8.1. Objects dépendants

Le composant est un objet inclu dans un autre qui est sauvegardé comme une valeur, et non pas comme une entité. Le composant fait référence à la notion (au sens objet) de composition (et non pas de composant au sens d'architecture de composants). Par exemple on pourrait modélisé l'objet personne de cette façon:

```
public class Person {
    private java.util.Date birthday;
    private Name name;
    private String key;
    public String getKey() {
        return key;
    }
    private void setKey(String key) {
        this.key=key;
    }
    public java.util.Date getBirthday() {
        return birthday;
    }
    public void setBirthday(java.util.Date birthday) {
        this.birthday = birthday;
    }
    public Name getName() {
        return name;
    }
    public void setName(Name name) {
        this.name = name;
    }
    ......
    ......
}
```

```
public class Name {
    char initial;
    String first;
    String last;
    public String getFirst() {
        return first;
    }
    void setFirst(String first) {
        this.first = first;
    }
    public String getLast() {
        return last;
    }
    void setLast(String last) {
        this.last = last;
    }
    public char getInitial() {
        return initial;
    }
    void setInitial(char initial) {
        this.initial = initial;
    }
}
```

Maintenant `Name` peut-être sauvegardé comme un composant de `Person`. Remarquer que `Name` définit des methodes d'accès et de modification pour ses propriétés persistantes, mais il n'a pas besoin des interfaces ou des

propriétés d'identification (par exemple getId()) qui sont propres aux entités.

Nous serions alors amené à mapper ce composant de cette façon:

```
<class name="eg.Person" table="person">
    <id name="Key" column="pid" type="string">
        <generator class="uuid"/>
    </id>
    <property name="birthday" type="date"/>
    <component name="Name" class="eg.Name"> <!-- class attribute optional -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </component>
</class>
```

La table person aurai les colonnes `pid`, `birthday`, `initial`, `first` and `last`.

Comme tous les types valeurs, les composants ne supportent pas les références partagés. En d'autres mots, deux instances de person peuvent avoir un même nom, mais ces noms sont indépendants, ils peuvent être identiques si on les compare par valeur mais ils représentent deux objets distincts en mémoire. La notion de nullité pour un composant est *ad hoc*. Quand il recharge l'objet qui contient le composant, Hibernate supposera que si tous les champs du composants sont nuls alors le composant sera positionné à la valeur null. Ce choix programmatif devrait être satisfaisant dans la plupart des cas.

Les propriétés d'un composant peuvent être de tous les types qu'Hibernate supporte habituellement (collections, many-to-one associations, autres composants, etc). Les composants inclus ne doivent *pas* être vus comme quelque chose d'exotique. Hibernate a été conçu pour supporter un modèle objet très granulaire.

Le `<component>` peut inclure dans la liste de ses propriétés une référence au `<parent>` conteneur.

```
<class name="eg.Person" table="person">
    <id name="Key" column="pid" type="string">
        <generator class="uuid"/>
    </id>
    <property name="birthday" type="date"/>
    <component name="Name" class="eg.Name" unique="true">
        <parent name="namedPerson"/> <!-- référence arrière à Person -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </component>
</class>
```

8.2. Collection d'objets dépendants

Les collections d'objets dépendants sont supportés (exemple: un tableau de type `Name`). Déclarer la collection de composants en remplaçant le tag `<element>` par le tag `<composite-element>`.

```
<set name="someNames" table="some_names" lazy="true">
    <key column="id"/>
    <composite-element class="eg.Name"> <!-- class attribute required -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </composite-element>
</set>
```

Remarque: Si vous définissez un `Set` d'élément composite, il est très important d'implémenter la méthode `equals()` et `hashCode()` correctement.

Les élements composite peuvent aussi contenir des composants mais pas des collections. Si votre élément composite contient aussi des composants, utilisez l'élément `<nested-composite-element>` . Une collections de composants qui ccontiennent eux-mêmes des composants est un cas très exotique. A ce stade demandez-vous si une association un-à-plusieurs ne serait pas plus approprié. Essayez de re remodeler votre élément composite comme une entité (Dans ce cas même si le modèle Java est le même la logique de persitence et de relation sont tout de même différentes)

Remarque, le mapping d'éléments composites ne supporte pas la nullité des propriétés lorsqu'on utilise un `<set>`. Hibernate lorsqu'il supprime un objet utilise chaque colonne pour identifier un objet (on ne peut pas utiliser des clés primaires distinctes dans une table d'éléments composites), ce qui n'est pas possible avec des valeurs nulles. Vous devez donc choisir d'interdire la nullité des propriétés d'un élément composite ou choisir un autre type de collection comme : `<list>`, `<map>`, `<bag>` ou `<idbag>`.

Un cas particulier d'élément composite est un élément composite qui inclut un élément `<many-to-one>`. Un mapping comme celui-ci vous permet d'associer les colonnes d'une table d'association plusieurs à plusieurs (many-to-many) à la classse de l'élément composite. L'exemple suivant est une association plusieurs à plusieurs de `Order` à `Item` à `purchaseDate`, `price` et `quantity` sont des propriétés de l'association.

```
<class name="eg.Order" .... >
    ....
    <set name="purchasedItems" table="purchase_items" lazy="true">
        <key column="order_id">
        <composite-element class="eg.Purchase">
            <property name="purchaseDate"/>
            <property name="price"/>
            <property name="quantity"/>
            <many-to-one name="item" class="eg.Item"/> <!-- class attribute is optional -->
        </composite-element>
    </set>
</class>
```

Bien sûr, il ne peut pas y avoir de référence à l'achat (purchase) depuis l'article (item), pour pouvoir naviguer de façon bidirectionnelle dans l'association. N'oubliez pas que les composants sont de type valeurs et n'autorise pas les références partagées.

Même les associations ternaires ou quaternaires sont possibles:

```
<class name="eg.Order" .... >
    ....
    <set name="purchasedItems" table="purchase_items" lazy="true">
        <key column="order_id">
        <composite-element class="eg.OrderLine">
            <many-to-one name="purchaseDetails class="eg.Purchase"/>
            <many-to-one name="item" class="eg.Item"/>
        </composite-element>
    </set>
</class>
```

Les éléments composites peuvent apparaître dans les requêtes en utilisant la même syntaxe que associations

8.3. Utiliser les composants comme index de map

l'élément `<composite-map-key>` vous permet d'utiliser une classe de composant comme indice de `Map`. Assurez-vous d'avoir surdéfini `hashCode()` et `equals()` dans la classe du composant.

8.4. Utiliser un composant comme identifiant

Vous pouvez utiliser un composant comme identifiant d'une entité. Mais pour cela la classe du composant doit respecter certaines règles.

- Elle doit implémenter `java.io.Serializable`.
- Elle doit redéfinir `equals()` et `hashCode()`, de façon cohérente avec le fait qu'elle définit une clé composite dans la base de données.

Remarque: avec hibernate3, la seconde règle n'est plus absolument necessaire mais faîtes le quand même.

Vous ne pouvez pas utiliser de `IdentifierGenerator` pour générer une clé composite, l'application devra définir elle même ses propres identifiants.

Utiliser l'élément `<composite-id>` (en incluant l'élément `<key-property>`) à la place de l'habituel déclaration `<id>`. Par exemple la classe `OrderLine` qui dépend de la clé primaire (composite) de `Order`.

```
<class name="OrderLine">

    <composite-id name="id" class="OrderLineId">
        <key-property name="lineId"/>
        <key-property name="orderId"/>
        <key-property name="customerId"/>
    </composite-id>

    <property name="name"/>

    <many-to-one name="order" class="Order"
            insert="false" update="false">
        <column name="orderId"/>
        <column name="customerId"/>
    </many-to-one>
    ....

</class>
```

Maintenant toutes clés étrangères référençant la table `OrderLine` devra aussi être composite. Vous devez en tenir compte lorsque vous écrivez vos mapping d'association pour les autres classes. Une association à `OrderLine` devrait être mappé de la façon suivante :

```
<many-to-one name="orderLine" class="OrderLine">
<!-- the "class" attribute is optional, as usual -->
    <column name="lineId"/>
    <column name="orderId"/>
    <column name="customerId"/>
</many-to-one>
```

(Remarque: l'élément `<column>` est une alternative à l'attribut `column` que l'on utilise partout.)

Une association `plusieurs-à-plusieurs` (many-to-many) à `OrderLine` utilisera aussi une clé étrangère composite:

```
<set name="undeliveredOrderLines">
    <key column name="warehouseId"/>
    <many-to-many class="OrderLine">
        <column name="lineId"/>
        <column name="orderId"/>
        <column name="customerId"/>
    </many-to-many>
</set>
```

La collection des `OrderLines` dans `Order` utilisera:

```
<set name="orderLines" inverse="true">
```

```
    <key>
        <column name="orderId"/>
        <column name="customerId"/>
    </key>
    <one-to-many class="OrderLine"/>
</set>
```

(L'élément `<one-to-many>`, comme d'habitude, ne déclare pas de colonne.)

Si `OrderLine` lui-même possède une collection, celle-ci aura aussi une clé composite étrangère.

```
<class name="OrderLine">
    ....
    ....
    <list name="deliveryAttempts">
        <key>   <!-- a collection inherits the composite key type -->
            <column name="lineId"/>
            <column name="orderId"/>
            <column name="customerId"/>
        </key>
        <list-index column="attemptId" base="1"/>
        <composite-element class="DeliveryAttempt">
            ...
        </composite-element>
    </set>
</class>
```

8.5. Composant Dynamique

Vous pouvez même mapper une propriété de type `Map`:

```
<dynamic-component name="userAttributes">
    <property name="foo" column="FOO"/>
    <property name="bar" column="BAR"/>
    <many-to-one name="baz" class="Baz" column="BAZ_ID"/>
</dynamic-component>
```

La sémantique de l'association à un `<dynamic-component>` est identique à celle que l'on utilise pour les composants. L'avantage de ce type de mapping est qu'il pemet de déterminer les véritables propriétés du bean au moment su déploiement en éditant simplement le document de mapping. La manipulation du document de mapping pendant l'execution de l'application est aussi possible en utilisant un parser DOM. Il ya même mieux, vous pouvez accéder (et changer) le metamodel de configuration d'hibernate en utilisant l'objet `Configuration`

Chapitre 9. Mapping d'héritage de classe

9.1. Les trois stratégies

Hibernate supporte les trois stratégies d'héritage de base :

- une table par hiérarchie de classe (table per class hierarchy)
- une table par classe fille (table per subclass)
- une table par classe concrète (table per concrete class)

Hibernate supporte en plus une quatrièmestratégie, légèrement différente, qui supporte le polymorphisme :

- le polymorphisme implicite

Il est possible d'utiliser différentes stratégies de mapping pour différentes branches d'une même hiérarchie d'héritage, et alors d'employer le polymorphisme implicite pour réaliser le polymorphisme à travers toute la hiérarchie. Pourtant, Hibernate ne supporte pas de mélanger des mappings `<subclass>` et `<joined-subclass>` et `<union-subclass>` pour le même élément `<class>` racine. Il est possible de mélanger ensemble les stratégies d'une table par hiérarchie et d'une table par sous-classe, pour le même élément `<class>`, en combinant les éléments `<subclass>` et `<join>` (voir dessous).

Il est possible de définir des mappings de `subclass`, `union-subclass`, et `joined-subclass` dans des documents de mapping séparés, directement sous `hibernate-mapping`. Ceci vous permet d'étendre une hiérarchie de classe juste en ajoutant un nouveau fichier de mapping. Vous devez spécifier un attribut `extends` dans le mapping de la sous-classe, en nommant une super-classe précédemment mappée. Note : précédemment cette foncionnalité rendait l'ordre des documents de mapping important. Depuis Hibernate3, l'ordre des fichier de mapping n'importe plus lors de l'utilisation du mot-clef "extends". L'ordre à l'intérieur d'un simple fichier de mapping impose encore de définir les classes mères avant les classes filles.

```
<hibernate-mapping>
    <subclass name="DomesticCat" extends="Cat" discriminator-value="D">
         <property name="name" type="string"/>
    </subclass>
</hibernate-mapping>
```

9.1.1. Une table par hiérarchie de classe

Supposons que nous ayons une interface `Payment`, implémentée par `CreditCardPayment`, `CashPayment`, `ChequePayment`. La stratégie une table par hiérarchie serait :

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </subclass>
```

```
    <subclass name="CashPayment" discriminator-value="CASH">
        ...
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        ...
    </subclass>
</class>
```

Une seule table est requise. Une grande limitation de cette stratégie est que les colonnes déclarées par les classes filles, telles que `CCTYPE`, ne peuvent avoir de contrainte `NOT NULL`.

9.1.2. Une table par classe fille

La stratégie une table par classe fille serait :

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="AMOUNT"/>
    ...
    <joined-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </joined-subclass>
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </joined-subclass>
</class>
```

Quatre tables sont requises. Les trois tables des classes filles ont une clé primaire associée à la table classe mère (le modèle relationnel est une association un-vers-un).

9.1.3. Une table par classe fille, en utilisant un discriminant

Notez que l'implémentation Hibernate de la stratégie un table par classe fille ne nécessite pas de colonne discriminante dans la table classe mère. D'autres implémentations de mappers Objet/Relationnel utilisent une autre implémentation de la stratégie une table par classe fille qui nécessite une colonne de type discriminant dans la table de la classe mère. L'approche prise par Hibernate est plus difficile à implémenter mais plus correcte d'une point de vue relationnel. Si vous aimeriez utiliser une colonne discriminante avec la stratégie d'une table par classe fille, vous pourriez combiner l'utilisation de `<subclass>` et `<join>`, comme suit :

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
        <join table="CREDIT_PAYMENT">
            <key column="PAYMENT_ID"/>
            <property name="creditCardType" column="CCTYPE"/>
            ...
        </join>
    </subclass>
```

```
    <subclass name="CashPayment" discriminator-value="CASH">
        <join table="CASH_PAYMENT">
            <key column="PAYMENT_ID"/>
            ...
        </join>
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        <join table="CHEQUE_PAYMENT" fetch="select">
            <key column="PAYMENT_ID"/>
            ...
        </join>
    </subclass>
</class>
```

La déclaration optionnelle `fetch="select"` indique à Hibernate de ne pas récupérer les données de la classe fille `ChequePayment` par une jointure externe lors des requêtes sur la classe mère.

9.1.4. Mélange d'une table par hiérarchie de classe avec une table par classe fille

Vous pouvez même mélanger les stratégies d'une table par hiérarchie de classe et d'une table par classe fille en utilisant cette approche :

```
<class name="Payment" table="PAYMENT">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="PAYMENT_TYPE" type="string"/>
    <property name="amount" column="AMOUNT"/>
    ...
    <subclass name="CreditCardPayment" discriminator-value="CREDIT">
        <join table="CREDIT_PAYMENT">
            <property name="creditCardType" column="CCTYPE"/>
            ...
        </join>
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        ...
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        ...
    </subclass>
</class>
```

Pour importe laquelle de ces stratégies, une association polymorphique vers la classe racine `Payment` est mappée en utilisant `<many-to-one>`.

```
<many-to-one name="payment" column="PAYMENT_ID" class="Payment"/>
```

9.1.5. Une table par classe concrète

Il y a deux manières d'utiliser la stratégie d'une table par classe concrète. La première est d'employer `<union-subclass>`.

```
<class name="Payment">
    <id name="id" type="long" column="PAYMENT_ID">
        <generator class="sequence"/>
    </id>
    <property name="amount" column="AMOUNT"/>
    ...
    <union-subclass name="CreditCardPayment" table="CREDIT_PAYMENT">
```

```
        <property name="creditCardType" column="CCTYPE"/>
        ...
    </union-subclass>
    <union-subclass name="CashPayment" table="CASH_PAYMENT">
        ...
    </union-subclass>
    <union-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        ...
    </union-subclass>
</class>
```

Trois tables sont nécessaires pour les classes filles. Chaque table définit des colonnes pour toutes les propriétés de la classe, incluant les propriétés héritéés.

La limitation de cette approche est que si une propriété est mappée sur la classe mère, le nom de la colonne doit être le même pour toutes les classes filles. (Nous pourrions être plus souple dans une future version d'Hibernate). La stratégie du générateur d'identifiant n'est pas permise dans l'héritage de classes filles par union, en effet la valeur (NdT : seed) de la clef primaire doit être partagée par toutes les classes filles "union" d'une hiérarchie.

Si votre classe mère est abstraite, mappez la avec `abstract="true"`. Bien sûr, si elle n'est pas abstraite, une table supplémentaire (par défaut, `PAYMENT` dans l'exemple ci-dessus) est requise pour contenir des instances de la classe mère.

9.1.6. Une table par classe concrète, en utilisant le polymorphisme implicite

Une approche alternative est l'emploi du polymorphisme implicite :

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
    <id name="id" type="long" column="CREDIT_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CREDIT_AMOUNT"/>
    ...
</class>

<class name="CashPayment" table="CASH_PAYMENT">
    <id name="id" type="long" column="CASH_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CASH_AMOUNT"/>
    ...
</class>

<class name="ChequePayment" table="CHEQUE_PAYMENT">
    <id name="id" type="long" column="CHEQUE_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <property name="amount" column="CHEQUE_AMOUNT"/>
    ...
</class>
```

Notez que nulle part nous ne mentionnons l'interface `Payment` explicitement. Notez aussi que des propriétés de `Payment` sont mappées dans chaque classe fille. Si vous voulez éviter des duplications, considérez l'utilisation des entités XML (cf. `[ <!ENTITY allproperties SYSTEM "allproperties.xml"> ]` dans la déclaration du `DOCTYPE` et `&allproperties;` dans le mapping).

L'inconvénient de cette approche est qu'Hibernate ne génère pas d'`UNION`s SQL lors de l'exécution des requêtes polymorphiques.

Pour cette stratégie de mapping, une association polymorphique pour `Payment` est habituellement mappée en utilisant `<any>`.

```
<any name="payment" meta-type="string" id-type="long">
    <meta-value value="CREDIT" class="CreditCardPayment"/>
    <meta-value value="CASH" class="CashPayment"/>
    <meta-value value="CHEQUE" class="ChequePayment"/>
    <column name="PAYMENT_CLASS"/>
    <column name="PAYMENT_ID"/>
</any>
```

9.1.7. Mélange du polymorphisme implicite avec d'autres mappings d'héritage

Il y a une chose supplémentaire à noter à propos de ce mapping. Puisque les classes filles sont chacune mappées avec leur propre élément `<class>` (et puisque `Payment` est juste une interface), chaque classe fille pourrait facilement faire partie d'une autre hiérarchie d'héritage ! (Et vous pouvez encore faire des requêtes polymorphiques pour l'interface `Payment`).

```
<class name="CreditCardPayment" table="CREDIT_PAYMENT">
    <id name="id" type="long" column="CREDIT_PAYMENT_ID">
        <generator class="native"/>
    </id>
    <discriminator column="CREDIT_CARD" type="string"/>
    <property name="amount" column="CREDIT_AMOUNT"/>
    ...
    <subclass name="MasterCardPayment" discriminator-value="MDC"/>
    <subclass name="VisaPayment" discriminator-value="VISA"/>
</class>

<class name="NonelectronicTransaction" table="NONELECTRONIC_TXN">
    <id name="id" type="long" column="TXN_ID">
        <generator class="native"/>
    </id>
    ...
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="amount" column="CASH_AMOUNT"/>
        ...
    </joined-subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        <key column="PAYMENT_ID"/>
        <property name="amount" column="CHEQUE_AMOUNT"/>
        ...
    </joined-subclass>
</class>
```

Encore une fois, nous ne mentionnons pas explicitement `Payment`. Si nous exécutons une requête sur l'interface `Payment` - par exemple, `from Payment` - Hibernate retournera automatiquement les instances de `CreditCardPayment` (et ses classes filles puisqu'elles implémentent aussi `Payment`), `CashPayment` et `ChequePayment` mais pas les instances de `NonelectronicTransaction`.

9.2. Limitations

Il y a certaines limitations à l'approche du "polymorphisme implicite" pour la stratégie de mapping d'une table par classe concrète. Il y a plutôt moins de limitations restrictives aux mappings `<union-subclass>`.

La table suivante montre les limitations des mappings d'une table par classe concrète, et du polymorphisme implicite, dans Hibernate.

Tableau 9.1. Caractéristiques du mapping d'héritage

Stratégie d'héritage	many-to-one polymorphique	one-to-one polymorphique	one-to-many polymorphique	many-to-many polymorphique	load()/get() polymorphique	Requêtes polymorphiques	Jointures polymorphiques
une table par hiérarchie de classe	`<many-to-one>`	`<one-to-one>`	`<one-to-many>`	`<many-to-many>`	`s.get(Payment.class, id)`	`from Payment p`	`from Order o join o.payment p`
une table par classe fille	`<many-to-one>`	`<one-to-one>`	`<one-to-many>`	`<many-to-many>`	`s.get(Payment.class, id)`	`from Payment p`	`from Order o join o.payment p`
une table par classe concrète (union-subclass)	`<many-to-one>`	`<one-to-one>`	`<one-to-many>` (pour `inverse="true"` seulement)	`<many-to-many>`	`s.get(Payment.class, id)`	`from Payment p`	`from Order o join o.payment p`
une table par classe concrète (polymorphisme implicite)	`<any>`	*non supporté*	*non supporté*	`<many-to-any>`	`s.createCriteria(Payment.class).add( Restrictions.idEq(id) ).uniqueResult()`	`from Payment p`	*non supportées*

Chapitre 10. Travailler avec des objets

Hibernate est une solution de mapping objet/relationnel complète qui ne masque pas seulement au développpeur les détails du système de gestion de base de données sous-jacent, mais offre aussi *la gestion d'état* des objets. C'est, contrairement à la gestion de `statements` SQL dans les couches de persistance habituelles JDBC/SQL, une vue orientée objet très naturelle de la persistance dans les applications Java.

En d'autres mots, les développeurs d'applications Hibernate devrait toujours réfléchir à *l'état* de leurs objets, et pas nécessairement à l'exécution des expressions SQL. Cette part est prise en charge pas Hibernate et seulement importante pour les développeurs d'applications lors du réglage de la performance de leur système.

10.1. États des objets Hibernate

Hibernate définit et comprend les états suivants :

- *Éphémère* (NdT : transient) - un objet est éphémère s'il a juste été instancié en utilisant l'opérateur `new`. Il n'a aucune représentation persistante dans la base de données et aucune valeur d'identifiant n'a été assignée. Les instances éphémères seront détruites par le ramasse-miettes si l'application n'en conserve aucune référence. Utilisez la `Session` d'Hibernate pour rendre un objet persistant (et laisser Hibernate s'occuper des expressions SQL qui ont besoin d'être exécutées pour cette transistion).
- *Persistant* - une instance persistante a une représentation dans la base de données et une valeur d'identifiant. Elle pourrait avoir juste été sauvegardée ou chargée, pourtant, elle est par définition dans la portée d'une `Session`. Hibernate détectera n'importe quels changements effectués sur un objet dans l'état persistant et synchronisera l'état avec la base de données lors de la fin l'unité de travail. Les développeurs n'exécutent pas d'expressions `UPDATE` ou `DELETE` manuelles lorsqu'un objet devrait être rendu éphémère.
- *Détaché* - une instance détachée est un objet qui a été persistant, mais dont sa `Session` a été fermée. La référence à l'objet est encore valide, bien sûr, et l'instance détachée pourrait même être modifiée dans cet état. Une instance détachée peut être réattachée à une nouvelle `Session` plus tard dans le temps, la rendant (et toutes les modifications avec) de nouveau persistante. Cette fonctionnalité rend possible un modèle de programmation pour de longues unités de travail qui requièrent un temps de réflexion de l'utilisateur. Nous les appelons des *conversations*, c'est-à-dire une unité de travail du point de vue de l'utilisateur.

Nous alons maintenant dicuster des états et des transitions d'état (et des méthodes d'Hibernate qui déclenchent une transition) plus en détails.

10.2. Rendre des objets persistants

Les instances nouvellement instanciées d'une classe persistante sont considérées *éphémères* par Hibernate. Nous pouvons rendre une instance éphémère *persistante* en l'associant avec une session :

```
DomesticCat fritz = new DomesticCat();
fritz.setColor(Color.GINGER);
fritz.setSex('M');
fritz.setName("Fritz");
Long generatedId = (Long) sess.save(fritz);
```

Si `Cat` a un identifiant généré, l'identifiant est généré et assigné au `cat` lorsque `save()` est appelée. Si `Cat` a un identifiant `assigned`, ou une clef composée, l'identifiant devrait être assigné à l'instance de `cat` avant d'appeler `save()`. Vous pouvez aussi utiliser `persist()` à la place de`save()`, avec la sémantique définie plus tôt dans le

brouillon d'EJB3.

Alternativement, vous pouvez assigner l'identifiant en utilisant une version surchargée de `save()`.

```
DomesticCat pk = new DomesticCat();
pk.setColor(Color.TABBY);
pk.setSex('F');
pk.setName("PK");
pk.setKittens( new HashSet() );
pk.addKitten(fritz);
sess.save( pk, new Long(1234) );
```

Si l'objet que vous rendez persistant a des objets associés (par exemple, la collection `kittens` dans l'exemple précédent), ces objets peuvent être rendus persistants dans n'importe quel ordre que vous souhaitez à moins que vous ayez une contrainte `NOT NULL` sur la colonne de la clef étrangère. Il n'y a jamais de risque de violer une contrainte de clef étrangère. Cependant, vous pourriez violer une contrainte `NOT NULL` si vous appeliez `save()` sur les objets dans le mauvais ordre.

Habituellement, vous ne vous préoccupez pas de ce détail, puisque vous utiliserez très probablement la fonctionnalité de *persistance transitive* d'Hibernate pour sauvegarder les objets associés automatiquement. Alors, même les violations de contrainte `NOT NULL` n'ont plus lieu - Hibernate prendra soin de tout. La persistance transitive est traitée plus loin dans ce chapitre.

10.3. Chargement d'un objet

Les méthodes `load()` de `Session` vous donnent un moyen de récupérer une instance persistante si vous connaissez déjà son identifiant. `load()` prend un objet de classe et chargera l'état dans une instance nouvellement instanciée de cette classe, dans un état persistant.

```
Cat fritz = (Cat) sess.load(Cat.class, generatedId);
```

```
// vous avez besoin d'envelopper les identiants primitifs
long pkId = 1234;
DomesticCat pk = (DomesticCat) sess.load( Cat.class, new Long(pkId) );
```

Alternativement, vous pouvez charger un état dans une instance donnée :

```
Cat cat = new DomesticCat();
// load pk's state into cat
sess.load( cat, new Long(pkId) );
Set kittens = cat.getKittens();
```

Notez que `load()` lèvera une exception irrécupérable s'il n'y a pas de ligne correspondante dans la base de données. Si la classe est mappée avec un proxy, `load()` retourne juste un proxy non initialisé et n'accède en fait pas à la base de données jusqu'à ce que vous invoquiez une méthode du proxy. Ce comportement est très utile si vous souhaitez créer une association vers un objet sans réellement le charger à partir de la base de données. Cela permet aussi à de multiples instances d'être chargées comme un lot si `batch-size` est défini pour le mapping de la classe.

Si vous n'êtes pas certain qu'une ligne correspondante existe, vous devriez utiliser la méthode `get()`, laquelle accède à la base de données immédiatement et retourne null s'il n'y a pas de ligne correspondante.

```
Cat cat = (Cat) sess.get(Cat.class, id);
if (cat==null) {
    cat = new Cat();
    sess.save(cat, id);
}
```

```
return cat;
```

Vous pouvez même charger un objet en employant un `SELECT ... FOR UPDATE` SQL, en utilisant un `LockMode`. Voir la documentation de l'API pour plus d'informations.

```
Cat cat = (Cat) sess.get(Cat.class, id, LockMode.UPGRADE);
```

Notez que n'importe quelles instances associées ou collections contenues *ne sont pas* sélectionnées par `FOR UPDATE`, à moins que vous ne décidiez de spécifier `lock` ou `all` en tant que style de cascade pour l'association.

Il est possible de re-charger un objet et toutes ses collections à n'importe quel moment, en utilisant la méthode `refresh()`. C'est utile lorsque des "triggers" de base de données sont utilisés pour initiliser certains propriétés de l'objet.

```
sess.save(cat);
sess.flush(); //force the SQL INSERT
sess.refresh(cat); //re-read the state (after the trigger executes)
```

Une question importante apparaît généralement à ce point : combien (NdT : de données) Hibernate charge-t-il de la base de données et combient de `SELECT`s utilisera-t-il ? Cela dépent de la *stratégie de récupération* et cela est expliqué dans Section 19.1, « Stratégies de chargement ».

10.4. Requêtage

Si vous ne connaissez par les identifiants des objets que vous recherchez, vous avez besoin d'une requête. Hibernate supporte un langage de requêtes orientées objet facile à utiliser mais puissant. Pour la création de requêtes par programmation, Hibernate supporte une fonction de requêtage sophistiqué Criteria et Example (QBC et QBE). Vous pouvez aussi exprimez votre requête dans le SQL natif de votre base de données, avec un support optionnel d'Hibernate pour la conversion des ensembles de résultats en objets.

10.4.1. Exécution de requêtes

Les requêtes HQL et SQL natives sont représentées avec une instance de `org.hibernate.Query`. L'interface offre des méthodes pour la liaison des paramètres, la gestion des ensembles de resultats, et pour l'exécution de la requête réelle. Vous obtenez toujours une `Query` en utilisant la `Session` courante :

```
List cats = session.createQuery(
    "from Cat as cat where cat.birthdate < ?")
    .setDate(0, date)
    .list();

List mothers = session.createQuery(
    "select mother from Cat as cat join cat.mother as mother where cat.name = ?")
    .setString(0, name)
    .list();

List kittens = session.createQuery(
    "from Cat as cat where cat.mother = ?")
    .setEntity(0, pk)
    .list();

Cat mother = (Cat) session.createQuery(
    "select cat.mother from Cat as cat where cat = ?")
    .setEntity(0, izi)
    .uniqueResult();

Query mothersWithKittens = (Cat) session.createQuery(
```

```
    "select mother from Cat as mother left join fetch mother.kittens");
Set uniqueMothers = new HashSet(mothersWithKittens.list());
```

Une requête est généralement exécutée en invoquant `list()`, le résultat de la requête sera chargée complètement dans une collection en mémoire. Les intances d'entités recupérées par une requête sont dans un état persistant. La méthode `uniqueResult()` offre un raccourci si vous savez que votre requête retournera seulement un seul objet.

10.4.1.1. Itération de résultats

Occasionnellement, vous pourriez être capable d'obtenir de meilleures performances en exécutant la requête avec la méthode `iterate()`. Ce sera généralement seulement le cas si vous espérez que les intances réelles d'entité retournées par la requête soient déjà chargées dans la session ou le cache de second niveau. Si elles ne sont pas cachées, `iterate()` sera plus lent que `list()` et pourrait nécessiter plusieurs accès à la base de données pour une simple requête, généralement *1* pour le select initial qui retourne seulement les identifiants, et *n* selects supplémentaires pour initialiser les instances réelles.

```
// fetch ids
Iterator iter = sess.createQuery("from eg.Qux q order by q.likeliness").iterate();
while ( iter.hasNext() ) {
    Qux qux = (Qux) iter.next();  // fetch the object
    // something we couldnt express in the query
    if ( qux.calculateComplicatedAlgorithm() ) {
        // delete the current instance
        iter.remove();
        // dont need to process the rest
        break;
    }
}
```

10.4.1.2. Requêtes qui retournent des tuples

Les requêtes d'Hibernate retournent parfois des tuples d'objets, auquel cas chaque tuple est retourné comme un tableau :

```
Iterator kittensAndMothers = sess.createQuery(
            "select kitten, mother from Cat kitten join kitten.mother mother")
            .list()
            .iterator();

while ( kittensAndMothers.hasNext() ) {
    Object[] tuple = (Object[]) kittensAndMothers.next();
    Cat kitten  = tuple[0];
    Cat mother  = tuple[1];
    ....
}
```

10.4.1.3. Résultats scalaires

Des requêtes peuvent spécifier une propriété d'une classe dans la clause `select`. Elles peuvent même appeler des fonctions d'aggrégat SQL. Les propriétés ou les aggrégats sont considérés comme des résultats "scalaires" (et pas des entités dans un état persistant).

```
Iterator results = sess.createQuery(
        "select cat.color, min(cat.birthdate), count(cat) from Cat cat " +
        "group by cat.color")
        .list()
        .iterator();

while ( results.hasNext() ) {
```

```
    Object[] row = (Object[]) results.next();
    Color type = (Color) row[0];
    Date oldest = (Date) row[1];
    Integer count = (Integer) row[2];
    .....
}
```

10.4.1.4. Lier des paramètres

Des méthodes de `Query` sont fournies pour lier des valeurs à des paramètres nommés ou à des paramètres de style JDBC `?`. *Contrairement à JDBC, les numéros des paramètres d'Hibernate commencent à zéro.* Les paramètres nommés sont des identifiants de la forme `:nom` dans la chaîne de caractères de la requête. Les avantages des paramètres nommés sont :

- les paramètres nommés sont insensibles à l'ordre de leur place dans la chaîne de la requête
- ils peuvent apparaître plusieurs fois dans la même requête
- ils sont auto-documentés

```
//paramètre nomme (préféré)
Query q = sess.createQuery("from DomesticCat cat where cat.name = :name");
q.setString("name", "Fritz");
Iterator cats = q.iterate();
```

```
//paramètre positionnel
Query q = sess.createQuery("from DomesticCat cat where cat.name = ?");
q.setString(0, "Izi");
Iterator cats = q.iterate();
```

```
//liste de paramètres nommés
List names = new ArrayList();
names.add("Izi");
names.add("Fritz");
Query q = sess.createQuery("from DomesticCat cat where cat.name in (:namesList)");
q.setParameterList("namesList", names);
List cats = q.list();
```

10.4.1.5. Pagination

Si vous avez besoin de spécifier des liens sur votre ensemble de résultats (le nombre maximum de lignes que vous voulez récupérez et/ou la première ligne que vous voulez récupérer) vous devriez utiliser des méthodes de l'interface `Query` :

```
Query q = sess.createQuery("from DomesticCat cat");
q.setFirstResult(20);
q.setMaxResults(10);
List cats = q.list();
```

Hibernate sait comment traduite cette requête de limite en SQL natif pour votre SGBD.

10.4.1.6. Itération "scrollable"

Si votre connecteur JDBC supporte les `ResultSets` "scrollables", l'interface `Query` peut être utilisée pour obtenir un objet `ScrollableResults`, lequel permet une navigation flexible dans les résultats de la requête.

```
Query q = sess.createQuery("select cat.name, cat from DomesticCat cat " +
                           "order by cat.name");
ScrollableResults cats = q.scroll();
if ( cats.first() ) {

```

```
    // trouve le premier nom sur chaque page d'une liste alphabétique de noms de chats
    firstNamesOfPages = new ArrayList();
    do {
        String name = cats.getString(0);
        firstNamesOfPages.add(name);
    }
    while ( cats.scroll(PAGE_SIZE) );

    // Maintenant, obtiens la première page de chats
    pageOfCats = new ArrayList();
    cats.beforeFirst();
    int i=0;
    while( ( PAGE_SIZE > i++ ) && cats.next() ) pageOfCats.add( cats.get(1) );

}
cats.close()
```

Notez qu'une connexion ouverte (et un curseur) est requise pour cette fonctionnalité, utilisez `setMaxResult()`/`setFirstResult()` si vous avez besoin d'une fonctionnalité de pagination hors ligne.

10.4.1.7. Externaliser des requêtes nommées

Vous pouvez aussi définir des requêtes nommées dans le document de mapping. (Souvenez-vous d'utiliser une section `CDATA` si votre requête contient des caractères qui pourraient être interprétés comme des éléments XML.)

```
<query name="eg.DomesticCat.by.name.and.minimum.weight"><![CDATA[
    from eg.DomesticCat as cat
        where cat.name = ?
        and cat.weight > ?
] ]></query>
```

La liaison de paramètres et l'exécution sont fait par programmation :

```
Query q = sess.getNamedQuery("eg.DomesticCat.by.name.and.minimum.weight");
q.setString(0, name);
q.setInt(1, minWeight);
List cats = q.list();
```

Notez que le code réel du programme est indépendant du langage de requête qui est utilisé, vous pouvez aussi définir des requêtes SQL nativez dans les méta-données, ou migrer des requêtes existantes vers Hibernate en les plaçant dans les fichiers de mapping.

Notez aussi que la déclaration d'une requête dans un élément `<hibernate-mapping>` nécessite un nom globalement unique pour la requête, alors que la déclaration d'une requête dans une élément `<class>` est rendue unique de manière automatique par la mise en préfixe du nom entièrement de la classe, par exemple `eg.Cat.ByNameAndMaximumWeight`.

10.4.2. Filtrer des collections

Un *filtre* de collection est un type spécial de requête qui peut être appliqué à une collection persistante ou à un tableau. La chaîne de requête peut se référer à `this`, correspondant à l'élément de la collection courant.

```
Collection blackKittens = session.createFilter(
    pk.getKittens(),
    "where this.color = ?")
    .setParameter( Color.BLACK, Hibernate.custom(ColorUserType.class) )
    .list()
);
```

La collection retournée est considérée comme un bag, et c'est une copie de la collection donnée. La collection originale n'est pas modifiée (c'est contraire à l'implication du nom "filtre"; mais cohérent avec le comportement attendu).

Observez que les filtres ne nécessitent pas une clause `from` (bien qu'ils puissent en avoir une si besoin est). Les filtres ne sont pas limités à retourner des éléments de la collection eux-mêmes.

```
Collection blackKittenMates = session.createFilter(
    pk.getKittens(),
    "select this.mate where this.color = eg.Color.BLACK.intValue")
    .list();
```

Même une requête de filtre vide est utile, par exemple pour charger un sous-ensemble d'éléments dans une énorme collection :

```
Collection tenKittens = session.createFilter(
    mother.getKittens(), "")
    .setFirstResult(0).setMaxResults(10)
    .list();
```

10.4.3. Requêtes Criteria

HQL est extrêmement puissant mais certains développeurs préfèrent construire des requêtes dynamiquement, en utilisant l'API orientée objet, plutôt que construire des chaînes de requêtes. Hibernate fournit une API intuitive de requête `Criteria` pour ces cas :

```
Criteria crit = session.createCriteria(Cat.class);
crit.add( Expression.eq( "color", eg.Color.BLACK ) );
crit.setMaxResults(10);
List cats = crit.list();
```

Les APIs `Criteria` et `Example` associé sont traitées plus en détail dans Chapitre 15, *Requêtes par critères*.

10.4.4. Requêtes en SQL natif

Vous pouvez exprimer une requête en SQL, en utilisant `createSQLQuery()` et laisser Hibernate s'occuper du mapping des résultats vers des objets. Notez que vous pouvez n'importe quand appeler `session.connection()` et utiliser directement la `Connection` JDBC. Si vous choisissez d'utiliser l'API Hibernate, vous devez mettre les alias SQL entre accolades :

```
List cats = session.createSQLQuery(
    "SELECT {cat.*} FROM CAT {cat} WHERE ROWNUM<10",
    "cat",
    Cat.class
).list();
```

```
List cats = session.createSQLQuery(
    "SELECT {cat}.ID AS {cat.id}, {cat}.SEX AS {cat.sex}, " +
           "{cat}.MATE AS {cat.mate}, {cat}.SUBCLASS AS {cat.class}, ... " +
    "FROM CAT {cat} WHERE ROWNUM<10",
    "cat",
    Cat.class
).list()
```

Les requêtes SQL peuvent contenir des paramètres nommés et positionnels, comme des requêtes Hibernate. Plus d'informations à propos des requêtes SQL natives dans Hibernate peuvent être trouvées dans Chapitre 16,

SQL natif.

10.5. Modifier des objets persistants

Les *instances persistantes transactionnelles* (c'est-à-dire des objets chargés, sauvegardés, créés ou requêtés par la `Session`) peuvent être manipulées par l'application et n'importe quel changement vers l'état persistant sera persisté lorsque la `Session` est *"flushée"* (traité plus tard dans ce chapitre). Il n'y a pas besoin d'appeler une méthode particulière (comme `update()`, qui a un but différent) pour rendre vos modifications persistantes. Donc la manière la plus directe de mettre à jour l'état d'un objet est de le charger avec `load()`, et puis le manipuler directement, tant que la `Session` est ouverte :

```
DomesticCat cat = (DomesticCat) sess.load( Cat.class, new Long(69) );
cat.setName("PK");
sess.flush();  // changes to cat are automatically detected and persisted
```

Parfois ce modèle de programmation est inefficace puisqu'il nécessiterait un `SELECT` SQL (pour charger l'objet) et un `UPDATE` SQL (pour persister son état mis à jour) dans la même session. Aussi Hibernate offre une autre approche, en utilisant des instances détachées.

Notez que Hibernate n'offre par sa propre API pour l'exécution directe d'expressions `UPDATE` ou `DELETE`. Hibernate est un service de gestion d'état, *vous n'avez pas à penser aux* expressions *pour l'utiliser. JDBC est une API parfaite pour exécuter des expressions SQL, vous pouvez obtenir une `Connection` JDBC n'importe quand en appelant `session.connection()`. En outre, la notion d'opérations de masse entre en conflit avec le mapping objet/relationnel pour les applications orientées processus de transactions en ligne. Les futures versions d'Hibernate peuvent cependant fournir des fonctions d'opération de masse. Voir Chapitre 13, Traitement par paquet pour les astuces possibles d'opérations groupées.*

10.6. Modifier des objets détachés

Beaucoup d'applications ont besoin de récupérer un objet dans une transaction, l'envoyer à la couche interfacée avec l'utilisateur pour les manipulations, puis sauvegarder les changements dans une nouvelle transaction. Les applications qui utilisent cette approche dans un environnement à haute concurrence utilisent généralement des données versionnées pour assurer l'isolation pour les "longues" unités de travail.

Hibernate supporte ce modèle en permettant pour le réattachement d'instances détachées l'utilisation des méthodes `Session.update()` ou `Session.merge()` :

```
// dans la première session
Cat cat = (Cat) firstSession.load(Cat.class, catId);
Cat potentialMate = new Cat();
firstSession.save(potentialMate);

// dans une couche plus haute de l'application
cat.setMate(potentialMate);

// plus tard, dans une nouvelle session
secondSession.update(cat);  // update cat
secondSession.update(mate); // update mate
```

Si le `Cat` avec l'identifiant `catId` avait déjà été chargé par `secondSession` lorsque l'application a essayé de le réattacher, une exception aurait été levée.

Utilisez `update()` si vous êtes sure que la session ne contient pas déjà une instance persistante avec le même identifiant, et `merge()` si vous voulez fusionner vos modifications n'importe quand sans considérer l'état de la

session. En d'autres mots, `update()` est généralement la première méthode que vous devriez appeler dans une session fraîche, pour s'assurer que le réattachement de vos instances détachées est la première opération qui est exécutée.

L'application devrait individuellement `update()` (NdT : mettre à jour) les instances détachées accessibles depuis l'instance détachée donnée si et *seulement* si elle veut que leur état soit aussi mis à jour. Ceci peut être automatisé bien sûr, en utilisant la *persistance transitive*, voir Section 10.11, « Persistance transitive ».

La méthode `lock()` permet aussi à une application de réassocier un objet avec une nouvelle session. Pourtant, l'instance détachée doit être non modifiée !

```
//réassocie :
sess.lock(fritz, LockMode.NONE);
//fait une vérification de version, puis réassocie :
sess.lock(izi, LockMode.READ);
//fait une vérification de version, en utilisant SELECT ... FOR UPDATE, puis réassocie :
sess.lock(pk, LockMode.UPGRADE);
```

Notez que `lock()` peut être utilisé avec différents `LockModes`, voir la documentation de l'API documentation et le chapitre sur la gestion des transactions pour plus d'informations. Le réattachement n'est pas le seul cas d'utilisation pour `lock()`.

D'autres modèles pour de longues unités de travail sont traités dans Section 11.3, « Contrôle de consurrence optimiste ».

10.7. Détection automatique d'un état

Les utilisateurs d'Hibernate ont demandé une méthode dont l'intention générale serait soit de sauvegarder une instance éphémère en générant un nouvel identifiant, soit mettre à jour/réattacher les instances détachées associées à l'identifiant courant. La méthode `saveOrUpdate()` implémente cette fonctionnalité.

```
// dans la première session
Cat cat = (Cat) firstSession.load(Cat.class, catID);

// dans une partie plus haute de l'application
Cat mate = new Cat();
cat.setMate(mate);

// plus tard, dans une nouvelle session
secondSession.saveOrUpdate(cat);   // met à jour un état existant (cat a un identifiant non-null)
secondSession.saveOrUpdate(mate);  // sauvegarde les nouvelles instances (mate a un identiant null)
```

L'usage et la sémantique de `saveOrUpdate()` semble être confuse pour les nouveaux utilisateurs. Premièrement, aussi longtemps que vous n'essayez pas d'utiliser des instances d'une session dans une autre, vous ne devriez pas avoir besoin d'utiliser `update()`, `saveOrUpdate()`, ou `merge()`. Certaines applications n'utiliseront jamais ces méthodes.

Généralement `update()` ou `saveOrUpdate()` sont utilisées dans le scénario suivant :

- l'application charge un objet dans la première session
- l'objet est passé à la couche utilisateur
- certaines modifications sont effectuées sur l'objet
- l'objet est retourné à la couche logique métier
- l'application persiste ces modifications en appelant `update()` dans une seconde sessin

`saveOrUpdate()` s'utilise dans le cas suivant :

- si l'objet est déjà persistant dans cette session, ne rien faire
- si un autre objet associé à la session a le même identifiant, lever une exception
- si l'objet n'a pas de propriété d'identifiant, appeler `save()`
- si l'identifiant de l'objet a une valeur assignée à un objet nouvellement instancié, appeler `save()`
- si l'objet est versionné (par `<version>` ou `<timestamp>`), et la valeur de la propriété de version est la même valeur que celle assignée à un objet nouvellement instancié, appeler `save()`
- sinon mettre à jour l'objet avec `update()`

et `merge()` est très différent :

- s'il y a une instance persistante avec le même identifiant couramment associée à la session, copier l'état de l'objet donné dans l'instance persistante
- s'il n'y a pas d'instance persistante associée à cette session, essayer de le charger à partir de la base de données, ou créer une nouvelle instance persistante
- l'instance persistante est retournée
- l'instance donnée ne devient pas associée à la session, elle reste détachée

10.8. Suppression d'objets persistants

`Session.delete()` supprimera l'état d'un objet de la base de données. Bien sûr, votre application pourrait encore conserver une référence vers un objet effacé. Il est mieux de penser à `delete()` comme rendant une instance persistante éphémère.

```
sess.delete(cat);
```

Vous pouvez effacer des objets dans l'ordre que vous voulez, sans risque de violations de contrainte de clef étrangère. Il est encore possible de violer une contrainte `NOT NULL` sur une colonne de clef étrangère en effaçant des objets dans le mauvais ordre, par exemple si vous effacer le parent, mais oubliez d'effacer les enfants.

10.9. Réplication d'objets entre deux entrepôts de données

Il est occasionnellement utile de pouvoir prendre un graphe d'instances persistantes et de les rendre persistantes dans un entrepôt différent, sans regénérer les valeurs des identifiants.

```
//récupère un cat de la base de données
Session session1 = factory1.openSession();
Transaction tx1 = session1.beginTransaction();
Cat cat = session1.get(Cat.class, catId);
tx1.commit();
session1.close();

// réconcilie la seconde base de données
Session session2 = factory2.openSession();
Transaction tx2 = session2.beginTransaction();
session2.replicate(cat, ReplicationMode.LATEST_VERSION);
tx2.commit();
session2.close();
```

Le `ReplicationMode` détermine comment `replicate()` traitera les conflits avec les lignes existantes dans la base de données.

- `ReplicationMode.IGNORE` - ignore l'objet s'il y a une ligne existante dans la base de données avec le même identifiant
- `ReplicationMode.OVERWRITE` - écrase n'importe quelle ligne existante dans la base de données avec le même identifiant

- `ReplicationMode.EXCEPTION` - lève une exception s'il y une ligne dans la base de données avec le même identifiant
- `ReplicationMode.LATEST_VERSION` - écrase la ligne si son numéro de version est plus petit que le numéro de version de l'objet, ou ignore l'objet sinon

Les cas d'utilisation de cette fonctionnalité incluent la réconciliation de données entrées dans différentes base de données, l'extension des informations de configuration du système durant une mise à jour du produit, retour en arrière sur les changements effectués durant des transactions non-ACID, et plus.

10.10. Flush de la session

De temps en temps la `Session` exécutera les expressions SQL requises pour syncrhoniser l'état de la connexion JDBC avec l'état des objets retenus en mémoire. Ce processus, *flush*, arrive par défaut aux points suivants :

- lors de certaines exécutions de requête
- lors d'un appel à `org.hibernate.Transaction.commit()`
- lors d'un appel à `Session.flush()`

Les expressions SQL sont effectuées dans l'ordre suivant :

1. insertion des entités, dans le même ordre que celui des objets correspondants sauvegardés par l'appel à `Session.save()`
2. mise à jours des entités
3. suppression des collections
4. suppression, mise à jour et insertion des éléments des collections
5. insertion des collections
6. suppression des entités, dans le même ordre que celui des objets correspondants qui ont été supprimés par l'appel à `Session.delete()`

(Une exception est que des objets utilisant la génération `native` d'identifiants sont insérés lorsqu'ils sont sauvegardés.)

Excepté lorsque vous appelez `flush()` explicitement, il n'y absolument aucune garantie à propos de *quand* la `Session` exécute les appels JDBC, seulement sur l'*ordre* dans lequel ils sont exécutés. Cependant, Hibernate garantit que `Query.list(..)` ne retournera jamais de données périmées, ni des données fausses.

Il est possible de changer le comportement par défaut, donc que le flush se produise moins fréquemment. La classe `FlushMode` définit trois modes différents : flush seulement lors du commit (et seulement quand l'API `Transaction` d'Hibernate est utilisée), flush automatiquement en utilisant la procédure expliquée, ou jamais de flush à moins que `flush()` soit appelée explicitement. Le dernier mode est utile pour l'exécution de longues unités de travail, où une `Session` est gardée ouverte et déconnectée pour un long moment (voir Section 11.3.2, « Les sessions longues et le versionnage automatique. »).

```
sess = sf.openSession();
Transaction tx = sess.beginTransaction();
sess.setFlushMode(FlushMode.COMMIT); // permet aux requêtes de retourner un état périmé

Cat izi = (Cat) sess.load(Cat.class, id);
izi.setName(iznizi);

// pourrait retourner des données périmées
sess.find("from Cat as cat left outer join cat.kittens kitten");

// le changement pour izi n'est pas flushé !
...
tx.commit(); // le flush se produit
```

Durant le flush, une exception peut se produire (par exemple, si une opération de la DML viole une contrainte). Puisque les exceptions de gestion impliquent une certaine compréhension du comportement transactionnel d'Hibernate, nous le traitons dans Chapitre 11, *Transactions et accès concurrents*.

10.11. Persistance transitive

Il est assez pénible de sauvegarder, supprimer, ou réattacher des objets un par un, surtout si vous traitez un graphe d'objets associés. Un cas habituel est une relation parent/enfant. Considérez l'exemple suivant :

Si les enfants de la relation parent/enfant étaient des types de valeur (par exemple, une collection d'adresses ou de chaînes de caractères), leur cycle de vie dépendraient du parent et aucune action ne serait requise pour "cascader" facilement les changements d'état. Si le parent est sauvegardé, les objets enfants de type de valeur sont sauvegardés également, si le parent est supprimé, les enfants sont supprimés, etc. Ceci fonctionne même pour des opérations telles que la suppression d'un enfant de la collection ; Hibernate détectera cela et, puisque les objets de type de valeur ne peuvent pas avoir des références partagées, supprimera l'enfant de la base de données.

Maintenant considérez le même scénario avec un parent et dont les objets enfants sont des entités, et non des types de valeur (par exemple, des catégories et des objets, ou un parent et des chatons). Les entités ont leur propre cycle de vie, supportent les références partagées (donc supprimer une entité de la collection ne signifie pas qu'elle peut être supprimée), et il n'y a par défaut pas de cascade d'état d'une entité vers n'importe quelle entité associée. Hibernate n'implémente pas la *persistance par accessibilité* par défaut.

Pour chaque opération basique de la session d'Hibernate - incluant `persist()`, `merge()`, `saveOrUpdate()`, `delete()`, `lock()`, `refresh()`, `evict()`, `replicate()` - il y a un style de cascade correspondant. Respectivement, les styles de cascade s'appellent `persist`, `merge`, `save-update`, `delete`, `lock`, `refresh`, `evict`, `replicate`. Si vous voulez qu'une opération soit cascadée le long d'une association, vous devez l'indiquer dans le document de mapping. Par exemple :

```
<one-to-one name="person" cascade="persist"/>
```

Les styles de cascade peuvent être combinés :

```
<one-to-one name="person" cascade="persist,delete,lock"/>
```

Vous pouvez même utiliser `cascade="all"` pour spécifier que *toutes* les opérations devraient être cascadées le long de l'association. La valeur par défaut `cascade="none"` spécifie qu'aucune opération ne sera cascadée.

Une style de cascade spécial, `delete-orphan`, s'applique seulement aux associations un-vers-plusieurs, et indique que l'opération `delete()` devrait être appliquée à n'importe quel enfant qui est supprimé de l'association.

Recommandations :

- Cela n'a généralement aucun sens d'activer la cascade sur une association `<many-to-one>` ou `<many-to-many>`. Les cascades sont souvent utiles pour des associations `<one-to-one>` et `<one-to-many>`.
- Si la durée de vie de l'objet enfant est liée à la durée de vie de l'objet parent, faites en un *objet du cycle de vie* en spécifiant `cascade="all,delete-orphan"`.
- Sinon, vous pourriez ne pas avoir besoin de cascade du tout. Mais si vous pensez que vous travaillerez souvent avec le parent et les enfants ensemble dans la même transaction, et que vous voulez vous éviter quelques frappes, considérez l'utilisation de `cascade="persist,merge,save-update"`.

Mapper une association (soit une simple association valuée, soit une collection) avec `cascade="all"` marque

l'association comme une relation de style *parent/enfant* où la sauvegarde/mise à jour/suppression du parent entraîne la sauvegarde/mise à jour/suppression de l'enfant ou des enfants.

En outre, une simple référence à un enfant d'un parent persistant aura pour conséquence la sauvegarde/mise à jour de l'enfant. Cette métaphore est cependant incomplète. Un enfant qui devient non référencé par son parent *n'est pas* automatiquement supprimée, excepté dans le cas d'une association `<one-to-many>` mappée avec `cascade="delete-orphan"`. La sémantique précise des opérations de cascade pour une relation parent/enfant est la suivante :

- Si un parent est passé à `persist()`, tous les enfant sont passés à `persist()`
- Si un parent est passé à `merge()`, tous les enfants sont passés à `merge()`
- Si un parent est passé à `save()`, `update()` ou `saveOrUpdate()`, tous les enfants sont passés à `saveOrUpdate()`
- Si un enfant détaché ou éphémère devient référencé par un parent persistant, il est passé à `saveOrUpdate()`
- Si un parent est supprimé, tous les enfants sont passés à `delete()`
- Si un enfant est déréférencé par un parent persistant, *rien de spécial n'arrive* - l'application devrait explicitement supprimer l'enfant si nécessaire - à moins que `cascade="delete-orphan"` soit paramétré, au quel cas l'enfant "orphelin" est supprimé.

Enfin, la cascade des opérations peut être effectuée sur un graphe donné lors de l'*appel de l'opération* or lors du *flush* suivant. Toutes les opérations, lorsque cascadées, le sont sur toutes les entités associées atteignables lorsque l'opétation est exécutée. Cependant `save-upate` et `delete-orphan` sont cascadées à toutes les entités associées atteignables lors du flush de la `Session`.

10.12. Utilisation des méta-données

Hibernate requiert un modèle de méta-niveau très riche de toutes les entités et types valués. De temps en temps, ce modèle est très utile à l'application elle même. Par exemple, l'application pourrait utiliser les méta-données d'Hibernate pour implémenter un algorithme de copie en profondeur "intelligent" qui comprendrait quels objets devraient copiés (par exemple les types de valeur mutables) et lesquels ne devraient pas l'être (par exemple les types de valeurs immutables et, possiblement, les entités associées).

Hibernate expose les méta-données via les interfaces `ClassMetadata` et `CollectionMetadata` et la hiérarchie `Type`. Les instances des interfaces de méta-données peuvent être obtenues à partir de la `SessionFactory`.

```
Cat fritz = ......;
ClassMetadata catMeta = sessionfactory.getClassMetadata(Cat.class);

Object[] propertyValues = catMeta.getPropertyValues(fritz);
String[] propertyNames = catMeta.getPropertyNames();
Type[] propertyTypes = catMeta.getPropertyTypes();

// récupère une Map de toutes les propriétés qui ne sont pas des collections ou des associations
Map namedValues = new HashMap();
for ( int i=0; i<propertyNames.length; i++ ) {
    if ( !propertyTypes[i].isEntityType() && !propertyTypes[i].isCollectionType() ) {
        namedValues.put( propertyNames[i], propertyValues[i] );
    }
}
```

Chapitre 11. Transactions et accès concurrents

L'un des principaux avantages du mécanisme de contrôle des accès concurrents d'Hibernate est qu'il est très facile à comprendre. Hibernate utilise directement les connexions JDBC ainsi que les ressources JTA sans y ajouter davantage de mécanisme de blocage. Nous vous recommandons de vous familiariser avec les spécifications JDBC, ANSI et d'isolement de transaction de la base de données que vous utilisez.

Hibernate ne vérouille pas vos objets en mémoire. Votre application peut suivre le comportement défini par le niveau d'isolation de vos transactions de base de données. Notez que grâce à la `Session`, qui est aussi un cache de scope transaction, Hibernate fournit des lectures répétées pour les récupération par identifiants et les requêtes d'entités (pas celle de valeurs scalaires).

En addition au versionning pour le controle automatique de concurrence, Hibernate fournit une API (mineure) pour le verrouillage perssimiste des enregistrements, en générant une syntaxe `SELECT FOR UPDATE`. Le controle de concurrence optimiste et cette API seront détaillés plus tard dans ce chapitre.

Nous aborderons la gestion des accès concurrents en discutant de la granularité des objets `Configuration`, `SessionFactory`, et `Session`, ainsi que de certains concepts relatifs à la base de données et aux longues transactions applicatives.

11.1. Gestion de session et délimitation de transactions

Il est important de savoir qu'un objet `SessionFactory` est un objet complexe et optimisé pour fonctionner avec les threads(thread- safe). Il est coûteux à créer et est ainsi prévu pour n'être instancié qu?une seule fois via un objet `Configuration` au démarrage de l'application, et être partagé par tous les threads d'une application.

Un objet `Session` est relativement simple et n'est threadsafe. Il est également peu coûteux à créer. Il devrait n'être utilisé qu'une seule fois, pour un processus d'affaire ou une unité de travail ou une conversation et ensuite être relâché. Un objet `Session` ne tentera pas d'obtenir de connexion (`Connection`) JDBC (ou de `Datasource`) si ce n'est pas nécessaire.

Afin de compléter ce tableau, vous devez également penser aux transactions de base de données. Une transaction de base de données se doit d'être la plus courte possible afin de réduire les risques de collision sur des enregistrements verrouillés. De longues transactions à la base de données nuiront à l'extensibilité de vos applications lorsque confrontées à de hauts niveaux de charge. Par conséquent, il n'est jamais bon de maintenir une transaction ouverte pendant la durée de reflexion de l'utilisateur, jusqu'a ce que l'unité de travail soit achevée.

Maintenant, comment délimiter une unité de travail? Est-ce qu'une instance de `Session` peut avoir une durée de vie dépassant plusieurs transactions à la base de données, ou bien est-ce que celles-ci doivent être liées une à une? Quand faut-il ouvrir et fermer une `Session` ? Comment définir la démarcation de vos transactions à la base de données?

11.1.1. Unité de travail

Il est important de mentionner que d'utiliser un paradigme *session-par-operation* est un anti-pattern. Autrement dit: n'ouvrez et ne fermez pas la `Session` à chacun de vos accès simples à la base de données dans un même thread! Bien sûr, le même raisonnement s'applique sur la gestion des transactions à la base de données. Les appels à la base de données devraient être faits en ordre et selon une séquence définie. Ils devraient également être regroupés en des unités de travail atomiques. (Notez que l?utilisation d?une connexion auto-commit constitue le même anti-pattern. Ce mode de fonctionnement existe pour les applications émettant des

commandes SQL à partir d?une console. Hibernate désengage le mode auto-commit et s'attend à ce qu'un serveur d'applications le fasse également.) Les transactions avec la base de données ne sont jamais optionnelles, toute communication avec une base de données doit se dérouler dans une transaction, peu importe si vous lisez ou écrivez des données. Comme évoqué, le comportement auto-commit pour lire les données devrait être évité, puisque plusieurs petites transactions ne seront jamais aussi efficaces qu'une seule plus grosse clairement définie comme unité de travail. Ce dernier choix et en plus beaucoup plus facile a maintenir et à faire évoluer.

Le pattern d'utilisation le plus fréquemment rencontré dans des applications clients serveur multi-usagers est le *session-per-request* (littéralement : Session par requête). Dans ce modèle, la requête d'un client est envoyée à un serveur (Où la couche de persistance est implémentée via Hibernate), une nouvelle `Session` est ouverte et toutes les opérations d'accès à la base de données sont exécutées à l'intérieur de celle-ci. Lorsque le travail est terminé (et que les réponses à envoyer au client ont été préparées), la session est flushée et fermée. Une seule transaction à la base de données peut être utilisée pour répondre à la requête du client. La transaction est démarrée et validée au même moment où la Session est ouverte et fermée. La relation entre la `Session` et la `Transaction` est donc one-to-one. Ce modèle permet de répondre parfaitement aux attentes de la grande majorité des applications.

Le défi réside dans l'implémentation. Hibernate fournit une fonction de gestion de la "session courante" pour simplifier ce pattern. Tout ce que vous devez faire est démarrer une transaction lorsqu'une requête est traitée par le serveur, et la terminer avant que la réponse ne soit envoyée au client. Vous pouvez le faire de la manière que vous voulez, les solutions communes sont un `ServletFilter`, l'interception via AOP avec une pointcut sur les méthodes de type "service", ou un conteneur avec interception/proxy. Un conteneur EJB est un moyen standard d'implémenter ce genre d'acpect tranverse comme la démarcation des transactions sur les EJBs session, de manière déclarative avec CMT. Si vous décidez d'utiliser la démarcation programmatique des transactions, préferrez l'API Hibernate `Transaction` détaillée plus tard dans ce chapitre, afin de facilité l'utilisation et la portabilité du code.

Votre application peut accéder la "session courante" pour exécuter une requête en invoquant simplement `sessionFactory.getCurrentSession()` n'importe où et autant de fois que souhaité. Vous obtiendrez toujours une `Session` dont le scope est la transaction courante avec la base de données. Ceci doit être configuré soit dans les ressources local ou dans l'environnement JTA, voir Section 2.5, « Sessions Contextuelles ».

Il est parfois utile d'étendre le scope d'une `Session` et d'une transaction à la base de données jusqu'à ce que "la vue soit rendue". Ceci est particulièrement utile dans des applications à base de servlet qui utilisent une phase de rendue séparée une fois que la réponse a été préparée. Etendre la transaction avec la base de données jusqu'à la fin du rendering de la vue est aisé si vous implémentez votre propre intercepteur. Cependant, ce n'est pas facile si vous vous appuyez sur les EJBs avec CMT, puisqu'une transaction sera achevée au retour de la méthode EJB, avant le rendu de la vue. Rendez vous sur le site Hibernate et sur le forum pour des astuces et des exemples sur le pattern *Open Session in View* pattern..

11.1.2. Longue conversation

Le paradigme *session-per-request* n'est pas le seul élément à utiliser dans le design de vos unités de travail. Plusieurs processus d'affaire requièrent toute une série d'interactions avec l'utilisateur, entrelacées d'accès à la base de donnée. Dans une application Web ou une application d'entreprise, il serait inacceptable que la durée de vie d'une transaction s'étale sur plusieurs interactions avec l'usager. Considérez l'exemple suivant:

- Un écran s'affiche. Les données vues par l'usager ont été chargées dans l'instance d'un objet `Session` , dans le cadre d'une transaction de base de données. L'usager est libre de modifier ces objets.
- L'usager clique "Sauvegarder" après 5 minutes et souhaite persister les modifications qu'il a apportées. Il s'attend à être la seule personne a avoir modifié ces données et qu'aucune modification conflictuelle ne se soit produite durant ce laps de temps.

Ceci s'appelle une unité de travail. Du point de vue de l'utilisateur: une *conversation* (ou *transaction d'application*). Il y a plusieurs façon de mettre ceci en place dans votre application.

Une première implémentation naïve pourrait consister à garder la `Session` et la transaction à la base de données ouvertes durant le temps de travail de l'usager, à maintenir les enregistrements verrouillés dans la base de données afin d'éviter des modifications concurrentes et de maintenir l'isolation et l'atomicité de la transaction de l'usager. Ceci est un anti-pattern à éviter, puisque le verrouillage des enregistrements dans la base de données ne permettrait pas à l'application de gérer un grand nombre d'usagers concurrents.

Il apparaît donc évident qu'il faille utiliser plusieurs transactions BDD afin d'implémenter la conversation. Dans ce cas, maintenir l'isolation des processus d'affaire devient partiellement la responsabilité de la couche applicative. Ainsi, la durée de vie d'une conversation devrait englober celle d'une ou de plusieurs transactions de base de données. Celle-ci sera atomique seulement si l'écriture des données mises à jour est faite exclusivement par la dernière transaction BDD la composant. Toutes les autres sous transactions BD ne doivent faire que la lecture de données. Ceci est relativement facile à mettre en place, surtout avec l'utilisation de certaines fonctionnalités d'Hibernate:

- *Versionnage Automatique* - Hibernate peut gérer automatiquement les accès concurrents de manière optimiste et détecter si une modification concurrente s'est produite durant le temps de réflexion d'un usager.
- *Objets Détachés* - Si vous décidez d'utiliser le paradigme *session-par-requête* discuté plus haut, toutes les entités chargées en mémoire deviendront des objets détachés durant le temps de réflexion de l'usager. Hibernate vous permet de rattacher ces objets et de persister les modifications y ayant été apportées. Ce pattern est appelé: *session-per- request-with-detached-objects* (littéralement: session-par-requête-avec-objets-détachés). Le versionnage automatique est utilisé afin d'isoler les modifications concurrentes.
- *Session Longues (conversation)* - Une `Session` Hibernate peut être déconnectée de la couche JDBC sous-jacente après que commit() ait été appelé sur une transaction à la base de données et reconnectée lors d'une nouvelle requête-client. Ce pattern s'appelle: *session-per-conversation* (Littéralement: session-par-conversation) et rend superflu le rattachement des objets. Le versionnage automatique est utilisé afin d'isoler les modifications concurrentes.

Les deux patterns *session-per-request-with- detached- objects* (session-par-requête-avec-objets- détachés) et *session-per-conversation* (session-par-conversation) ont chacun leurs avantages et désavantages qui seront exposés dans ce même chapitre, dans la section au sujet du contrôle optimiste de concurrence.

11.1.3. L'identité des objets

Une application peut accéder à la même entité persistante de manière concurrente dans deux `Session` s différentes. Toutefois, une instance d'une classe persistante n'est jamais partagée par deux instances distinctes de la classe `Session` . Il existe donc deux notions de l'identité d'un objet:

Identité BD

```
foo.getId().equals( bar.getId() )
```

Identité JVM

```
foo==bar
```

Ainsi, pour des objets attachés à une `Session` *précise* (dans la cadre d'exécution (scope) d'une instance de `Session`), ces deux notions d'identité sont équivalentes et garanties par Hibernate. Par contre, si une application peut accéder de manière concurrente à la même entité persistante dans deux sessions différentes, les

deux instances seront en fait différentes (en ce qui a trait à l'identité JVM). Les conflits sont résolus automatiquement par approche optimiste grâce au système de versionnage automatique lorsque `Session.flush()` ou `Transaction.commit()` est appelé.

Cette approche permet de reléguer à Hibernate et à la base de données sous-jacente le soin de gérer les problèmes d'accès concurrents. Cette manière de faire assure également une meilleure extensibilité de l'application puisque assurer l'identité JVM dans un thread ne nécessite pas de mécanismes de verrouillage coûteux ou d'autres dispositifs de synchronisation. Une application n'aura jamais le besoin de synchroniser des objets d'affaire tant qu'elle peut garantir qu'un seul thread aura accès à une instance de `Session` . Dans le cadre d'exécution d'un objet `Session` , l'application peut utiliser en toute sécurité `==` pour comparer des objets.

Une application qui utiliserait == à l'extérieur du cadre d'exécution d'une `Session` pourrait obtenir des résultats inattendus et causer certains effets de bords. Par exemple, si vous mettez 2 objets dans le même `Set` , ceux-ci pourraient avoir la même identité BD (i.e. ils représentent le même enregistrement), mais leur identité JVM pourrait être différente (elle ne peut, par définition, pas être garantie sur deux objets détachés). Le développeur doit donc redéfinir l'implémentation des méthodes `equals()` et `hashcode()` dans les classes persistantes et y adjoindre sa propre notion d'identité. Il existe toutefois une restriction: Il ne faut jamais utiliser uniquement l'identifiant de la base de données dans l'implémentation de l'égalité; Il faut utiliser une clé d'affaire, généralement une combinaison de plusieurs attributs uniques, si possible immuables. Les identifiants de base de données vont changer si un objet transitoire (transient) devient persistant. Si une instance transitoire est contenue dans un `Set` , changer le hashcode brisera le contrat du `Set` . Les attributs pour les clés d'affaire n'ont pas à être aussi stables que des clés primaires de bases de données. Il suffit simplement qu'elles soient stables tant et aussi longtemps que les objets sont dans le même `Set` . Veuillez consulter le site web Hibernate pour des discussions plus pointues à ce sujet. Notez que ce concept n'est pas propre à Hibernate mais bien général à l'implémentation de l'identité et de l'égalité en Java.

11.1.4. Problèmes communs

Bien qu'il puisse y avoir quelques rares exceptions à cette règle, il est recommandé de ne jamais utiliser les anti-patterns *session-per- user-session* et *session-per-application* . Vous trouverez ici- bas quelques problèmes que vous risquez de rencontrer si vous en faite l?utilisation. (Ces problèmes pourraient quand même survenir avec des patterns recommandés) Assurez-vous de bien comprendre les implications de chacun des patterns avant de prendre votre décision.

- L'objet `Session` n?est pas conçu pour être utilisé par de multiples threads. En conséquence, les objets potentiellement multi-thread comme les requêtes HTTP, les EJB Session et Swing Worker, risquent de provoquer des conditions de course dans la `Session` si celle-ci est partagée. Dans un environnement web classique, il serait préférable de synchroniser les accès à la session http afin d?éviter qu?un usager ne recharge une page assez rapidement pour que deux requêtes s?exécutant dans des threads concurrents n?utilisent la même `Session` .

- Lorsque Hibernate lance une exception, le roll back de la transaction en cours doit être effectué et la `Session` doit être immédiatement fermée. (Ceci sera exploré plus tard dans le chapitre.) Si la `Session` est directement associée à une application, il faut arrêter l?application. Le roll back de la transaction ne remettra pas les objets dans leur état du début de la transaction. Ainsi, ceux-ci pourraient être désynchronisés d?avec les enregistrements. (Généralement, cela ne cause pas de réels problèmes puisque la plupart des exceptions sont non traitables et requièrent la reprise du processus d?affaire ayant échoué.)

- La `Session` met en mémoire cache tous les objets persistants (les objets surveillés et dont l'état est géré par Hibernate.) Si la `Session` est ouverte indéfiniment ou si une trop grande quantité d'objets y est chargée, l?utilisation de la mémoire peut potentiellement croître jusqu?à atteindre le maximum allouable à l?application (java.lang.OutOfMemoryError.) Une solution à ce problème est d?appeler les méthodes

`Session.clear()` et `Session.evict()` pour gérer la mémoire cache de la `Session` . Vous pouvez également utiliser des stored procedures si vous devez lancer des traitements sur de grandes quantités d?informations. Certaines solutions sont décrites ici : Chapitre 13, *Traitement par paquet* . Garder une `Session` ouverte pour toute la durée d?une session usager augmente également considérablement le risque de travailler avec de l?information périmée.

11.2. Démarcation des transactions

La démarcation des transactions est importante dans le design d?une application. Aucune communication avec la base de données ne peut être effectuée à l?extérieur du cadre d?une transaction. (Il semble que ce concept soit mal compris par plusieurs développeurs trop habitués à utiliser le mode auto-commit.) Même si certains niveaux d'isolation et certaines possibilités offertes par les bases de données permettent de l?éviter, il n'est jamais désavantageux de toujours explicitement indiquer les bornes de transaction pour les opérations complexes comme pour les opérations simples de lecture.

Une application utilisant Hibernate peut s'exécuter dans un environnement léger n?offrant pas la gestion automatique des transactions (application autonome, application web simple ou applications Swing) ou dans un environnement J2EE offrant des services de gestion automatique des transactions JTA. Dans un environnement simple, Hibernate a généralement la responsabilité de la gestion de son propre pool de connexions à la base de données. Le développeur de l'application doit manuellement délimiter les transactions. En d'autres mots, il appartient au développeur de gérer les appels à `Transaction.begin()` , `Transaction.commit()` et `Transaction.rollback()` . Un environnement transactionnel J2EE (serveur d'application J2EE) doit offrir la gestion des transactions au niveau du container J2EE. Les bornes de transaction peuvent normalement être définies de manière déclarative dans les descripteurs de déploiement d'EJB Session, par exemple. La gestion programmatique des transactions n'y est donc pas nécessaire. Même les appels à `Session.flush()` sont faits automatiquement.

Il peut être requis d'avoir une couche de persistance portable. Hibernate offre donc une API appelée `Transaction` qui sert d'enveloppe pour le système de transaction natif de l'environnement de déploiement. Il n'est pas obligatoire d'utiliser cette API mais il est fortement conseillé de le faire, sauf lors de l'utilisation de CMT Session Bean (EJB avec transactions gérées automatiquement par le container EJB).

Il existe quatre étapes disctinctes lors de la fermeture d'une `Session`

- flush de la session
- commit de la transaction
- Fermeture de la session (Close)
- Gestion des exceptions

La synchronisation de bdd depuis la session (flush) a déjà été expliqué, nous nous attarderons maintenant à la démarcation des transactions et à la gestion des exceptions dans les environnements légers et les environnements J2EE.

11.2.1. Environnement non managé

Si la couche de persistance Hibernate s'exécute dans un environnement non managé, les connexions à la base de données seront généralement prises en charge par le mécanisme de pool d'Hibernate. La gestion de la session et de la transaction se fera donc de la manière suivante:

```
// Non-managed environment idiom
Session sess = factory.openSession();
Transaction tx = null;
try {
```

```
    tx = sess.beginTransaction();

    // do some work
    ...

    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

Vous n'avez pas à invoquer `flush()` explicitement sur la `Session` - l'appel de `commit()` déclenchera automatiquement la synchronisation (selon le Section 10.10, « Flush de la session » de la session. Un appel à `close()` marque la fin de la session. La conséquence directe est que la connexion à la base de données sera relachée par la session. Ce code est portable est fonctionne dans les environnements non managé ET les environnements JTA.

Une solution plus flexible est la gestion par contexte fourni par Hibernate que nous avons déjà rencontré:

```
// Non-managed environment idiom with getCurrentSession()
try {
    factory.getCurrentSession().beginTransaction();

    // do some work
    ...

    factory.getCurrentSession().getTransaction().commit();
}
catch (RuntimeException e) {
    factory.getCurrentSession().getTransaction().rollback();
    throw e; // or display error message
}
```

Vous ne verrez probablement jamais ces exemples de code dans les applications; les exceptions fatales (exceptions du système) ne devraient être traitées que dans la couche la plus "haute". En d'autres termes, le code qui exécute les appels à Hibernate (à la couche de persistance) et le code qui gère les `RuntimeException` (qui ne peut généralement effectuer qu'un nettoyage et une sortie) sont dans des couches différentes. La gestion du contexte courant par Hibernate peut simplifier notablement ce design, puisque vous devez accéder à la gestion des exceptions de la `SessionFactory`, ce qui est décrit plus tard dans ce chapitre.

Notez que vous devriez sélectionner `org.hibernate.transaction.JDBCTransactionFactory` (le défaut), pour le second exemple `"thread"` comme `hibernate.current_session_context_class`.

11.2.2. Utilisation de JTA

Si votre couche de persistance s'exécute dans un serveur d'application (par exemple, derrière un EJB Session Bean), toutes les datasource utilisées par Hibernate feront automatiquement partie de transactions JTA globales. Hibernate propose deux stratégies pour réussir cette intégration.

Si vous utilisez des transactions gérées par un EJB (bean managed transactions - BMT), Hibernate informera le serveur d'application du début et de la fin des transactions si vous utilisez l'API `Transaction` . Ainsi, le code de gestion des transactions sera identique dans les deux types d'environnements.

```
// BMT idiom
Session sess = factory.openSession();
Transaction tx = null;
```

```
try {
    tx = sess.beginTransaction();

    // do some work
    ...

    tx.commit();
}
catch (RuntimeException e) {
    if (tx != null) tx.rollback();
    throw e; // or display error message
}
finally {
    sess.close();
}
```

Si vous souhaitez utiliser une `Session` couplée à la transaction, c'est à dire, utiliser la fonctionnalité `getCurrentSession()` pour la propagation facile du contexte, vous devrez utiliser l'API JTA `UserTransaction` directement:

```
// BMT idiom with getCurrentSession()
try {
    UserTransaction tx = (UserTransaction)new InitialContext()
                            .lookup("java:comp/UserTransaction");

    tx.begin();

    // Do some work on Session bound to transaction
    factory.getCurrentSession().load(...);
    factory.getCurrentSession().persist(...);

    tx.commit();
}
catch (RuntimeException e) {
    tx.rollback();
    throw e; // or display error message
}
```

Avec CMT, la démarcation des transactions est faite dans les descripteurs de déploiement des Beans Sessions et non de manière programmmatique, ceci réduit le code:

```
// CMT idiom
 Session sess = factory.getCurrentSession();

 // do some work
 ...
```

Dans un EJB CMT même le rollback intervient automatiquement, puisqu'une `RuntimeException` non traitée et soulevée par une méthode d'un bean session indique au conteneur d'annuler la transaction globale. *Ceci veut donc dire que vous n'avez pas à utiliser l'API `Transaction` d'Hibernate dans CMT.*

Notez que le fichier de configuration Hibernate devrait contenir les valeurs `org.hibernate.transaction.JTATransactionFactory` dans un environnement BMT ou `org.hibernate.transaction.CMTTransactionFactory` dans un environnement CMT là où vous configurez votre transaction factory Hibernate. N'oubliez pas non plus de spécifier le paramètre `org.hibernate.transaction.manager_lookup_class` . De plus, assurez vous de fixez votre `hibernate.current_session_context_class` soit à `"jta"` ou de ne pas le configurer (compatibilité avec les versions précédentes).

La méthode `getCurrentSession()` a un inconvénient dans les environnement JTA. Il y a une astuce qui est d'utiliser un mode de libération de connexion `after_statement` , qui est alors utilisé par défaut. Du à une étrange limitation de la spec JTA, il n'est pas possible pour Hibernate de nettoyer et ferme automatiquement un

`ScrollableResults` ouvert ou une instance d'`Iterator` retournés `scroll()` ou `iterate()`. Vous *devez* libérer le curseur base de données sous jacent ou invoquer `Hibernate.close(Iterator)` explicitement depuis un bloc `finally`. (Bien sur, la plupart des applications peuvent éviter d'uiliser `scroll()` ou `iterate()` dans un code CMT.)

11.2.3. Gestion des exceptions

Si une `Session` lance une exception (incluant les exceptions du type `SQLException` ou d'un sous-type), vous devez immédiatement faire le rollback de la transaction, appeler `Session.close()` et relâcher les références sur l'objet `Session` . La `Session` contient des méthodes pouvant la mettre dans un état inutilisable. Vous devez considérer qu'*aucune* exception lancée par Hibernate n'est traitable. Assurez-vous de fermer la session en faisant l'appel à `close()` dans un bloc `finally` .

L'exception `HibernateException` , qui englobe la plupart des exceptions pouvant survenir dans la couche de persistance Hibernate, est une exception non vérifiée (Ceci n'était pas le cas dans certaines versions antérieures de Hibernate.) Il est de notre avis que nous ne devrions pas forcer un développeur à gérer une exception qu'il ne peut de toute façon pas traiter dans une couche technique. Dans la plupart des applications, les exceptions non vérifiées et les exceptions fatales sont gérées en amont du processus (dans les couches hautes) et un message d'erreur est alors affiché à l'usager (ou un traitement alternatif est invoqué.) Veuillez noter qu'Hibernate peut également lancer des exceptions non vérifiées d'un autre type que `HibernateException` . Celles-ci sont également non traitables et vous devez les traiter comme telles.

Hibernate englobe les `SQLException` s lancées lors des interactions directes avec la base de données dans des exceptions de type: `JDBCException` . En fait, Hibernate essaiera de convertir l'exception dans un sous-type plus significatif de `JDBCException` . L'exception `SQLException` sous-jacente est toujours disponible via la méthode `JDBCException.getCause()` . Cette conversion est faite par un objet de type `SQLExceptionConverter` , qui est rattaché à l'objet `SessionFactory` . Par défaut, le `SQLExceptionConverter` est associé au dialecte de BD configuré dans Hibernate. Toutefois, il est possible de fournir sa propre implémentation de l'interface. (Veuillez vous référer à la javadoc sur la classe `SQLExceptionConverterFactory` pour plus de détails. Les sous-types standard de `JDBCException` sont:

- `JDBCConnectionException` - Indique une erreur de communication avec la couche JDBC sous-jacente.
- `SQLGrammarException` - Indique un problème de grammaire ou de syntaxe avec la requête SQL envoyée.
- `ConstraintViolationException` - Indique une violation de contrainte d'intégrité.
- `LockAcquisitionException` - Indique une erreur de verrouillage lors de l'éxécution de la requête.
- `GenericJDBCException` - Indique une erreur générique JDBC d'une autre catégorie.

11.2.4. Timeout de transaction

L'un des avantages fournis par les environnements transactionnels JTA (tels les containers EJB) est la gestion du timeout de transaction. La gestion des dépassements de temps de transaction vise à s'assurer qu'une transaction agissant incorrectement ne viendra pas bloquer indéfiniment les ressources de l'application. Hibernate ne peut fournir cette fonctionnalité dans un environnement transactionnel non-JTA. Par contre, Hibernate gère les opérations d'accès aux données en allouant un temps maximal aux requêtes pour s'exécuter. Ainsi, une requête créant de l'inter blocage ou retournant de très grandes quantités d'information pourrait être interrompue. Dans un environnement transactionnel JTA, Hibernate peut déléguer au gestionnaire de transaction le soin de gérer les dépassements de temps. Cette fonctionnalité est abstraite par l'objet `Transaction` .

```
Session sess = factory.openSession();
try {
    //mettre le timeout à 3 secondes.
    sess.getTransaction().setTimeout(3);
```

```
            sess.getTransaction().begin();

            // Effectuer le travail ...

            sess.getTransaction().commit()
        }
        catch (RuntimeException e) {
            if ( sess.getTransaction().isActive() ) {
                sess.getTransaction().rollback();
            }
            throw e;
            // ou afficher le message d'erreur.
        }
        finally {
            sess.close();
        }
```

Notez que `setTimeout()` ne peut pas être appelé d'un EJB CMT, puisque le timeout des transaction doit être spécifié de manière déclarative.

11.3. Contrôle de consurrence optimiste

La gestion optimiste des accès concurrents avec versionnage est la seule approche pouvant garantir l'extensibilité des applications à haut niveau de charge. Le système de versionnage utilise des numéros de version ou l'horodatage pour détecter les mises à jour causant des conflits avec d'autres actualisations antérieures. Hibernate propose trois approches pour l'écriture de code applicatif utilisant la gestion optimiste d'accès concurrents. Le cas d'utilisation décrit plus bas fait mention de conversation, mais le versionnage peut également améliorer la qualité d'une application en prévenant la perte de mises à jour.

11.3.1. Gestion du versionnage au niveau applicatif

Dans cet exemple d'implémentation utilisant peu les fonctionnalités d'Hibernate, chaque interaction avec la base de données se fait en utilisant une nouvelle `Session` et le développeur doit recharger les données persistantes à partir de la BD avant de les manipuler. Cette implémentation force l'application à vérifier la version des objets afin de maintenir l'isolation transactionnelle. Cette approche, semblable à celle retrouvée pour les EJB, est la moins efficace de celles présentées dans ce chapitre.

```
            // foo est une instance chargée antérieurement par une autre
            Session session = factory.openSession();
            Transaction t = session.beginTransaction();

            int oldVersion = foo.getVersion();
            session.load( foo, foo.getKey() ); // Charger l'état courant

            if ( oldVersion!=foo.getVersion )
                throw new StaleObjectStateException();

            foo.setProperty("bar");
            t.commit();
            session.close();
```

Le mapping de la propriété `version` est fait via `<version>` et Hibernate l'incrémentera automatiquement à chaque flush() si l'entité doit être mise à jour.

Bien sûr, si votre application ne fait pas face à beaucoup d'accès concurrents et ne nécessite pas l'utilisation du versionnage, cette approche peut également être utilisée, il n'y a qu'à ignorer le code relié au versionnage. Dans ce cas, la stratégie du *last commit wins* (littéralement: le dernier commit l'emporte) sera utilisée pour les

conversations (longues transactions applicatives). Gardez à l'esprit que cette approche pourrait rendre perplexe les utilisateurs de l'application car ils pourraient perdre des données mises à jour sans qu'aucun message d'erreur ne leur soit présenté et sans avoir la possibilité de fusionner les données.

Il est clair que la gestion manuelle de la vérification du versionnage des objets ne peut être effectuée que dans certains cas triviaux et que cette approche n'est pas valable pour la plupart des applications. De manière générale, les applications ne cherchent pas à actualiser de simples objets sans relations, elles le font généralement pour de larges graphes d'objets. Pour toute application utilisant le paradigme des conversations ou des objets détachés, Hibernate peut gérer automatiquement la vérification des versions d'objets.

11.3.2. Les sessions longues et le versionnage automatique.

Dans ce scénario, une seule instance de `Session` et des objets persistants est utilisée pour toute l'application. Hibernate vérifie la version des objets persistants avant d'effectuer le flush() et lance une exception si une modification concurrente est détectée. Il appartient alors au développeur de gérer l'exception. Les traitements alternatifs généralement proposés sont alors de permettre à l'usager de faire la fusion des données ou de lui offrir de recommencer son travail à partie des données les plus récentes dans la BD.

Il est à noter que lorsqu'une application est en attente d'une action de la part de l?usager, La `Session` n'est pas connectée à la couche JDBC sous-jacente. C'est la manière la plus efficace de gérer les accès à la base de données. L'application ne devrait pas se préoccuper du versionnage des objets, de la réassociation des objets détachés, ni du rechargement de tous les objets à chaque transaction.

```
// foo est une instance chargée antérieurement par une autre session

session.reconnect();// Obtention d'une nouvelle connexion JDBC
Transaction t = session.beginTransaction();
foo.setProperty("bar");
t.commit(); //Terminer la transaction, propager les changements et vérifier les versions.
session.disconnect(); // Retourner la connexion JDBC
```

L'objet `foo` sait quel objet `Session` l'a chargé. `Session.reconnect()` obtient une nouvelle connexion (celle-ci peut être également fournie) et permet à la session de continuer son travail. La méthode `Session.disconnect()` déconnecte la session de la connexion JDBC et retourne celle-ci au pool de connexion (à moins que vous ne lui ayez fourni vous même la connexion.) Après la reconnexion, afin de forcer la vérification du versionnage de certaines entités que vous ne cherchez pas à actualiser, vous pouvez faire un appel à `Session.lock()` en mode `LockMode.READ` pour tout objet ayant pu être modifié par une autre transaction. Il n'est pas nécessaire de verrouiller les données que vous désirez mettre à jour.

Si des appels implicites aux méthodes `disconnect()` et `reconnect()` sont trop coûteux, vous pouvez les éviter en utilisant `hibernate.connection.release_mode` .

Ce pattern peut présenter des problèmes si la `Session` est trop volumineuse pour être stockée entre les actions de l'usager. Plus spécifiquement, une session `HttpSession` se doit d'être la plus petite possible. Puisque la `Session` joue obligatoirement le rôle de mémoire cache de premier niveau et contient à ce titre tous les objets chargés, il est préférable de n'utiliser cette stratégie que pour quelques cycles de requêtes car les objets risquent d'y être rapidement périmés.

Notez que la `Session` déconnectée devrait être conservée près de la couche de persistance. Autrement dit, utilisez un EJB stateful pour conserver la `Session` et évitez de la sérialiser et de la transférer à la couche de présentation (i.e. Il est préférable de ne pas la conserver dans la session `HttpSession` .)

11.3.3. Les objets détachés et le versionnage automatique

Chaque interaction avec le système de persistance se fait via une nouvelle `Session` . Toutefois, les mêmes instances d'objets persistants sont réutilisées pour chacune de ces interactions. L'application doit pouvoir manipuler l'état des instances détachées ayant été chargées antérieurement via une autre session. Pour ce faire, ces objets persistants doivent être rattachés à la `Session` courante en utilisant `Session.update()` , `Session.saveOrUpdate()` , ou `Session.merge()` .

```
// foo est une instance chargée antérieurement par une autre session

foo.setProperty("bar");
session = factory.openSession();
Transaction t = session.beginTransaction();
session.saveOrUpdate(foo);  //Utiliser merge() si "foo" pourrait avoir été chargé précéder
t.commit();
session.close();
```

Encore une fois, Hibernate vérifiera la version des instances devant être actualisées durant le flush(). Une exception sera lancée si des conflits sont détectés.

Vous pouvez également utiliser `lock()` au lieu de `update()` et utiliser le mode `LockMode.READ` (qui lancera une vérification de version, en ignorant tous les niveaux de mémoire cache) si vous êtes certain que l'objet n'a pas été modifié.

11.3.4. Personnaliser le versionnage automatique

Vous pouvez désactiver l'incrémentation automatique du numéro de version de certains attributs et collections en mettant la valeur du paramètre de mapping `optimistic-lock` à false. Hibernate cessera ainsi d'incrémenter leur numéro de version s'ils sont mis à jour.

Certaines entreprises possèdent de vieux systèmes dont les schémas de bases de données sont statiques et ne peuvent être modifiés. Il existe aussi des cas où plusieurs applications doivent accéder à la même base de données, mais certaines d'entre elles ne peuvent gérer les numéros de version ou les champs horodatés. Dans les deux cas, le versionnage ne peut être implanté par le rajout d'une colonne dans la base de données. Afin de forcer la vérification de version dans un système sans en faire le mapping, mais en forçant une comparaison des états de tous les attributs d'une entité, vous pouvez utiliser l'attribut `optimistic- lock="all"` sous l'élément `<class>` . Veuillez noter que cette manière de gérer le versionnage ne peut être utilisée que si l'application utilises de longues sessions, lui permettant de comparer l'ancien état et le nouvel état d'une entité. L'utilisation d'un pattern `session-per-request-with-detached- objects` devient alors impossible.

Il peut être souhaitable de permettre les modifications concurrentes lorsque des champs distincts sont modifiés. En mettant la propriété `optimistic-lock="dirty"` dans l'élément `<class>` , Hibernate ne fera la comparaison que des champs devant être actualisés lors du flush().

Dans les deux cas: en utilisant une colonne de version/horodatée ou via la comparaison de l'état complet de l'objet ou de ses champs modifiés, Hibernate ne créera qu'une seule commande d'UPDATE par entité avec la clause WHERE appropriée pour mettre à jour l'entité *ET* en vérifier la version. Si vous utilisez la persistance transitive pour propager l'évènement de rattachement à des entités associées, il est possible qu'Hibernate génère des commandes d'UPDATE inutiles. Ceci n'est généralement pas un problème, mais certains déclencheurs *on update* dans la base de données pourraient être activés même si aucun changement n'était réellement persisté sur des objets associés. Vous pouvez personnaliser ce comportement en indiquant `select-before-update="true"` dans l'élément de mapping `<class>` . Ceci forcera Hibernate à faire le SELECT de l'instance afin de s'assurer que l'entité doit réellement être actualisée avant de lancer la commande d'UPDATE.

11.4. Verouillage pessimiste

Il n'est nécessaire de s'attarder à la stratégie de verrouillage des entités dans une application utilisant Hibernate. Il est généralement suffisant de définir le niveau d'isolation pour les connexions JDBC et de laisser ensuite la base de donnée effectuer son travail. Toutefois, certains utilisateurs avancés peuvent vouloir obtenir un verrouillage pessimiste exclusif sur un enregistrement et le réobtenir au lancement d'une nouvelle transaction.

Hibernate utilisera toujours le mécanisme de verrouillage de la base de données et ne verrouillera jamais les objets en mémoire!

La classe `LockMode` définit les différents niveaux de verrouillage pouvant être obtenus par Hibernate. Le verrouillage est obtenu par les mécanismes suivants:

- `LockMode.WRITE` est obtenu automatiquement quand Hibernate actualise ou insert un enregistrement.
- `LockMode.UPGRADE` peut être obtenu de manière explicite via la requête en utilisant `SELECT ... FOR UPDATE` sur une base de données supportant cette syntaxe.
- `LockMode.UPGRADE_NOWAIT` peut être obtenu de manière explicite en utilisant `SELECT ... FOR UPDATE NOWAIT` sur Oracle.
- `LockMode.READ` est obtenu automatiquement quand Hibernate lit des données dans un contexte d'isolation `Repeatable Read` ou `Serializable` . Peut être réobtenu explicitement via une requête.
- `LockMode.NONE` représente l'absence de verouillage. Tous les objets migrent vers ce mode a la fin d'une `Transaction` . Les objets associés à une session via un appel à `saveOrUpdate()` commencent également leur cycle de vie dans cet état.

Les niveaux de verrouillage peuvent être explicitement obtenus de l'une des manières suivantes:

- Un appel à `Session.load()` , en spécifiant un niveau verrouillage `LockMode` .
- Un appel à `Session.lock()` .
- Une appel à `Query.setLockMode()` .

Si `Session.load()` est appelé avec le paramètre de niveau de verouillage `UPGRADE` ou `UPGRADE_NOWAIT` et que l'objet demandé n'est pas présent dans la session, celui-ci sera chargé à l'aide d'une requête `SELECT ... FOR UPDATE` . Si la méthode `load()` est appelée pour un objet déjà en session avec un verrouillage moindre que celui demandé, Hibernate appellera la méthode `lock()` pour cet objet.

`Session.lock()` effectue une vérification de version si le niveau de verrouillage est `READ` , `UPGRADE` ou `UPGRADE_NOWAIT` . (Dans le cas des niveaux `UPGRADE` ou `UPGRADE_NOWAIT` , une requête `SELECT ... FOR UPDATE` sera utilisée.)

Si une base de données ne supporte pas le niveau de verrouillage demandé, Hibernate utilisera un niveau alternatif convenable au lieux de lancer une exception. Ceci assurera la portabilité de votre application.

11.5. Mode de libération de Connection

Le comportement original (2.x) d'Hibernate pour la gestion des connexions JDBC était que la `Session` obtenait une connexion dès qu'elle en avait besoin et la libérait une fois la session fermée. Hibernate 3 a introduit les modes de libération de connexion pour indiquer à la session comment gérer les transactions JDBC. Notez que la discussion suivante n'est pertinente que pour des connexions fournies par un `ConnectionProvider`, celles gérées par l'utilisateur sont en dehors du scope de cette discussion. Les différents modes sont définies par `org.hibernate.ConnectionReleaseMode`:

- `ON_CLOSE` - est essentiellement le comportement passé. La session Hibernate obtient une connexion

lorsqu'elle en a besoin et la garde jusqu'à ce que la session se ferme.

- `AFTER_TRANSACTION` - indique de relacher la connexion après qu'une `org.hibernate.Transaction` se soit achevée.
- `AFTER_STATEMENT` (aussi appelé libération brutale) - indique de relacher les connexions après chaque exécution d'un statement. Ce relachement aggressif est annulé si ce statement laisse des ressources associées à une session donnée ouvertes, actuellement ceci n'arrive que lors de l'utilisation de `org.hibernate.ScrollableResults`.

Le paramètre de configuration `hibernate.connection.release_mode` est utilisé pour spécifier quel mode de libération doit être utiliser. Les valeurs possibles sont:

- `auto` (valeur par défaut) - ce choix délègue le choix de libération à la méthode `org.hibernate.transaction.TransactionFactory.getDefaultReleaseMode()` Pour la JTATransactionFactory, elle retourne ConnectionReleaseMode.AFTER_STATEMENT; pour JDBCTransactionFactory, elle retourne ConnectionReleaseMode.AFTER_TRANSACTION. C'est rarement une bonne idée de changer ce comportement par défaut puisque les erreurs soulevées par ce paramétrage tend à prouver une erreur dans le code de l'utilisateur.
- `on_close` - indique d'utiliser ConnectionReleaseMode.ON_CLOSE. Ce paramétrage existe pour garantir la compatibilité avec les versions précédentes, mais ne devrait plus être utilisé.
- `after_transaction` - indique d'utiliser ConnectionReleaseMode.AFTER_TRANSACTION. Ne devrait pas être utilisé dans les environnements JTA. Notez aussi qu'avec ConnectionReleaseMode.AFTER_TRANSACTION, si une session est considérée comme étant en mode auto-commit les connexions seront relachées comme si le mode était AFTER_STATEMENT.
- `after_statement` - indique d'utiliser ConnectionReleaseMode.AFTER_STATEMENT. Additonnellement, le `ConnectionProvider` utilisé est consulté pour savoir s'il supporte ce paramétrage (`supportsAggressiveRelease()`). Si ce n'est pas le cas, le mode de libération est ré initialisé à ConnectionReleaseMode.AFTER_TRANSACTION. Ce paramétrage n'est sûr que dans les environnements où il est possible d'obtenir à nouveau la même connexion JDBC à chaque fois que l'on fait un appel de `ConnectionProvider.getConnection()` ou dans les envrionnements auto-commit où il n'est pas important d'obtenir plusieurs fois la même connexion.

Chapitre 12. Les intercepteurs et les événements

Il est souvent utile pour l'application de réagir à certains événements qui surviennent dans Hibernate. Cela autorise l'implémentation de certaines sortes de fonctionnalités génériques, et d'extensions de fonctionnalités d'Hibernate.

12.1. Intercepteurs

L'interface `Interceptor` fournit des "callbacks" de la session vers l'application et permettent à l'application de consulter et/ou de manipuler des propriétés d'un objet persistant avant qu'il soit sauvegardé, mis à jour, supprimé ou chargé. Une utilisation possible de cette fonctionnalité est de tracer l'accès à l'information. Par exemple, l'`Interceptor` suivant positionne `createTimestamp` quand un `Auditable` est créé et met à jour la propriété `lastUpdateTimestamp` quand un `Auditable` est mis à jour.

Vous pouvez soit implémenter `Interceptor` directement ou (mieux) étendre `EmptyInterceptor`.

```
package org.hibernate.test;

import java.io.Serializable;
import java.util.Date;
import java.util.Iterator;

import org.hibernate.EmptyInterceptor;
import org.hibernate.Transaction;
import org.hibernate.type.Type;

public class AuditInterceptor extends EmptyInterceptor {

    private int updates;
    private int creates;
    private int loads;

    public void onDelete(Object entity,
                         Serializable id,
                         Object[] state,
                         String[] propertyNames,
                         Type[] types) {
        // ne fait rien
    }

    public boolean onFlushDirty(Object entity,
                                Serializable id,
                                Object[] currentState,
                                Object[] previousState,
                                String[] propertyNames,
                                Type[] types) {

        if ( entity instanceof Auditable ) {
            updates++;
            for ( int i=0; i < propertyNames.length; i++ ) {
                if ( "lastUpdateTimestamp".equals( propertyNames[i] ) ) {
                    currentState[i] = new Date();
                    return true;
                }
            }
        }
        return false;
    }

    public boolean onLoad(Object entity,
                          Serializable id,
                          Object[] state,
                          String[] propertyNames,
```

```
                        Type[] types) {
        if ( entity instanceof Auditable ) {
            loads++;
        }
        return false;
    }

    public boolean onSave(Object entity,
                        Serializable id,
                        Object[] state,
                        String[] propertyNames,
                        Type[] types) {

        if ( entity instanceof Auditable ) {
            creates++;
            for ( int i=0; i<propertyNames.length; i++ ) {
                if ( "createTimestamp".equals( propertyNames[i] ) ) {
                    state[i] = new Date();
                    return true;
                }
            }
        }
        return false;
    }

    public void postFlush(Iterator entities) {
        System.out.println("Creations: " + creates + ", Updates: " + updates);
    }

    public void afterTransactionCompletion(Transaction tx) {
        if ( tx.wasCommitted() ) {
            System.out.println("Creations: " + creates + ", Updates: " + updates, "Loads: " + loads);
        }
        updates=0;
        creates=0;
        loads=0;
    }

}
```

Il y a deux types d'intercepteurs: lié à la `Session` et lié à la `SessionFactory`.

Un intercepteur lié à la `Session` est défini lorsqu'une session est ouverte via l'invocation des méthodes surchargées SessionFactory.openSession() acceptant un `Interceptor` (comme argument).

```
Session session = sf.openSession( new AuditInterceptor() );
```

Un intercepteur lié a `SessionFactory` est défini avec l'objet `Configuration` avant la construction de la `SessionFactory`. Dans ce cas, les intercepteurs fournis seront appliqués à toutes les sessions ouvertes pour cette `SessionFactory`; ceci est vrai à moins que la session ne soit ouverte en spécifiant l'intercepteur à utiliser. Les intercepteurs liés à la `SessionFactory` doivent être thread safe, faire attention à ne pas stocker des états spécifiques de la session puisque plusieurs sessions peuvent utiliser l'intercepteur de manière concurrente.

```
new Configuration().setInterceptor( new AuditInterceptor() );
```

12.2. Système d'événements

Si vous devez réagir à des événements particuliers dans votre couche de persistance, vous pouvez aussi utiliser l'architecture d'*événements* d'Hibernate3. Le système d'événements peut être utilisé en supplément ou en remplacement des interceptors.

Essentiellement toutes les méthodes de l'interface `Session` sont corrélées à un événement. Vous avez un `LoadEvent`, un `FlushEvent`, etc (consultez la DTD du fichier de configuration XML ou le paquet `org.hibernate.event` pour avoir la liste complète des types d'événement définis). Quand une requête est faite à partir d'une de ces méthodes, la `Session` Hibernate génère un événement approprié et le passe au listener configuré pour ce type. Par défaut, ces listeners implémentent le même traitement dans lequel ces méthodes aboutissent toujours. Cependant, vous êtes libre d'implémenter une version personnalisée d'une de ces interfaces de listener (c'est-à-dire, le `LoadEvent` est traité par l'implémentation de l'interface `LoadEventListener` déclarée), dans quel cas leur implémentation devrait être responsable du traitement des requêtes `load()` faites par la `Session`.

Les listeners devraient effectivement être considérés comme des singletons ; dans le sens où ils sont partagés entre des requêtes, et donc ne devraient pas sauvegarder des états de variables d'instance.

Un listener personnalisé devrait implémenter l'interface appropriée pour l'événement qu'il veut traiter et/ou étendre une des classes de base (ou même l'événement prêt à l'emploi utilisé par Hibernate comme ceux déclarés non-finaux à cette intention). Les listeners personnalisés peuvent être soit inscrits par programmation à travers l'objet `Configuration`, ou spécifiés la configuration XML d'Hibernate (la configuration déclarative à travers le fichier de propriétés n'est pas supportée). Voici un exemple de listener personnalisé pour l'événement de chargement :

```
public class MyLoadListener implements LoadEventListener {
    // C'est une simple méthode définie par l'interface LoadEventListener
    public void onLoad(LoadEvent event, LoadEventListener.LoadType loadType)
            throws HibernateException {
        if ( !MySecurity.isAuthorized( event.getEntityClassName(), event.getEntityId() ) ) {
            throw MySecurityException("Unauthorized access");
        }
    }
}
```

Vous avez aussi besoin d'une entrée de configuration disant à Hibernate d'utiliser ce listener en plus du listener par défaut :

```
<hibernate-configuration>
    <session-factory>
        ...
        <event type="load">
            <listener class="com.eg.MyLoadListener"/>
            <listener class="org.hibernate.event.def.DefaultLoadEventListener"/>
        </event>
    </session-factory>
</hibernate-configuration>
```

Vous pouvez aussi l'inscrire par programmation :

```
Configuration cfg = new Configuration();
LoadEventListener[] stack = { new MyLoadListener(), new DefaultLoadEventListener() };
cfg.EventListeners().setLoadEventListeners(stack);
```

Les listeners inscrits déclarativement ne peuvent pas partager d'instances. Si le même nom de classe est utilisée dans plusieurs éléments `<listener/>`, chaque référence sera une instance distincte de cette classe. Si vous avez besoin de la faculté de partager des instances de listener entre plusieurs types de listener, vous devez utiliser l'approche d'inscription par programmation.

Pourquoi implémenter une interface et définir le type spécifique durant la configuration ? Une implémentation de listener pourrait implémenter plusieurs interfaces de listener d'événements. Avoir en plus le type défini durant l'inscription rend plus facile l'activation ou la désactivation pendant la configuration.

12.3. Sécurité déclarative d'Hibernate

Généralement, la sécurité déclarative dans les applications Hibernate est gérée dans la couche de session. Maintenant, Hibernate3 permet à certaines actions d'être approuvées via JACC, et autorisées via JAAS. Cette fonctionnalité optionnelle est construite au dessus de l'architecture d'événements.

D'abord, vous devez configurer les listeners d'événements appropriés pour permettre l'utilisation d'autorisations JAAS.

```
<listener type="pre-delete" class="org.hibernate.secure.JACCPreDeleteEventListener"/>
<listener type="pre-update" class="org.hibernate.secure.JACCPreUpdateEventListener"/>
<listener type="pre-insert" class="org.hibernate.secure.JACCPreInsertEventListener"/>
<listener type="pre-load" class="org.hibernate.secure.JACCPreLoadEventListener"/>
```

Notez que `<listener type="..." class="..."/>` est juste un raccourci pour `<event type="..."><listener class="..."/></event>` quand il y a exactement un listener pour un type d'événement particulier.

Ensuite, toujours dans `hibernate.cfg.xml`, lier les permissions aux rôles :

```
<grant role="admin" entity-name="User" actions="insert,update,read"/>
<grant role="su" entity-name="User" actions="*"/>
```

Les noms de rôle sont les rôles compris par votre fournisseur JAAC.

Chapitre 13. Traitement par paquet

Une approche naïve pour insérer 100 000 lignes dans la base de données en utilisant Hibernate pourrait ressembler à ça :

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(.....);
    session.save(customer);
}
tx.commit();
session.close();
```

Ceci devrait s'écrouler avec une `OutOfMemoryException` quelque part aux alentours de la 50 000ème ligne. C'est parce qu'Hibernate cache toutes les instances de `Customer` nouvellement insérées dans le cache de second niveau.

Dans ce chapitre nous montrerons comment éviter ce problème. D'abord, cependant, si vous faites des traitements par batch, il est absolument critique que vous activiez l'utilisation ds paquet JDBC (NdT : JDBC batching), si vous avez l'intention d'obtenir des performances raisonnables. Configurez la taille du paquet JDBC avec un nombre raisonnable (disons, 10-50) :

```
hibernate.jdbc.batch_size 20
```

Notez qu'Hibernate désactive, de manière transparente, l'insertion par paquet au niveau JDBC si vous utilisez un générateur d'identifiant de type `identity`.

Vous pourriez aussi vouloir faire cette sorte de travail dans un traitement où l'interaction avec le cache de second niveau est complètement désactivé :

```
hibernate.cache.use_second_level_cache false
```

13.1. Insertions en paquet

Lorsque vous rendez des nouveaux objets persistants, vous devez régulièrement appeler `flush()` et puis `clear()` sur la session, pour contrôler la taille du cache de premier niveau.

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

for ( int i=0; i<100000; i++ ) {
    Customer customer = new Customer(.....);
    session.save(customer);
    if ( i % 20 == 0 ) { //20, même taille que la taille du paquet JDBC
        //flush un paquet d'insertions et libère la mémoire :
        session.flush();
        session.clear();
    }
}

tx.commit();
session.close();
```

13.2. Paquet de mises à jour

Pour récupérer et mettre à jour des données les mêmes idées s'appliquent. En plus, vous avez besoin d'utiliser `scroll()` pour tirer partie des curseurs côté serveur pour les requêtes qui retournent beaucoup de lignes de données.

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

ScrollableResults customers = session.getNamedQuery("GetCustomers")
    .setCacheMode(CacheMode.IGNORE)
    .scroll(ScrollMode.FORWARD_ONLY);
int count=0;
while ( customers.next() ) {
    Customer customer = (Customer) customers.get(0);
    customer.updateStuff(...);
    if ( ++count % 20 == 0 ) {
        //flush un paquet de mises à jour et libère la mémoire :
        session.flush();
        session.clear();
    }
}

tx.commit();
session.close();
```

13.3. L'interface StatelessSession

Alternativement, Hibernate fournit une API orientée commande qui peut être utilisée avec des flux de données pour et en provenance de la base de données sous la forme d'objets détachés. Une `StatelessSession` n'a pas de contexte de persistance associé et ne fournit pas beaucoup de sémantique de durée de vie de haut niveau. En particulier, une session sans état n'implémente pas de cache de premier niveau et n'interagit pas non plus avec un cache de seconde niveau ou un cache de requêtes. Elle n'implémente pas les transactions ou la vérification sale automatique (NdT : automatic dirty checking). Les opérations réalisées avec une session sans état ne sont jamais répercutées en cascade sur les instances associées. Les collections sont ignorées par une session sans état. Les opérations exécutées via une session sans état outrepasse le modèle d'événements d'Hibernate et les intercepteurs. Les sessions sans état sont vulnérables aux effets de modification des données, ceci est dû au manque de cache de premier niveau. Une session sans état est une abstraction bas niveau, plus proche de la couche JDBC sous-jacente.

```
StatelessSession session = sessionFactory.openStatelessSession();
Transaction tx = session.beginTransaction();

ScrollableResults customers = session.getNamedQuery("GetCustomers")
    .scroll(ScrollMode.FORWARD_ONLY);
while ( customers.next() ) {
    Customer customer = (Customer) customers.get(0);
    customer.updateStuff(...);
    session.update(customer);
}

tx.commit();
session.close();
```

Notez que dans le code de l'exemple, les intances de `Customer` retournées par la requête sont immédiatement détachées. Elles ne sont jamais associées à un contexte de persistance.

Les opérations `insert()`, `update()` et `delete()` définies par l'interface `StatelessSession` sont considérées comme des opérations d'accès direct aux lignes de la base de données, ce qui résulte en une exécution immédiate du SQL `INSERT`, `UPDATE` ou `DELETE` respectif. De là, elles ont des sémantiques tres différentes des opérations `save()`, `saveOrUpdate()` et `delete()` définies par l'interface `Session`.

13.4. Opérations de style DML

Comme déjà discuté avant, le mapping objet/relationnel automatique et transparent est intéressé par la gestion de l'état de l'objet. Ceci implique que l'état de l'objet est disponible en mémoire, d'où manipuler (en utilisant des expressions du langage de manipulation de données - `Data Manipulation Language` (DML) - SQL) les données directement dans la base n'affectera pas l'état en mémoire. Pourtant, Hibernate fournit des méthodes pour l'exécution d'expression DML de style SQL lesquelles sont réalisées à travers le langage de requête d'Hibernate (Chapitre 14, *HQL: Langage de requêtage d'Hibernate*).

La pseudo-syntaxe pour les expressions `UPDATE` et `DELETE` est : `( UPDATE | DELETE ) FROM? EntityName (WHERE where_conditions)?`. Certains points sont à noter :

- Dans la clause from, le mot-clef FROM est optionnel
- Il ne peut y avoir qu'une seule entité nommée dans la clause from ; elle peut optionnellement avoir un alias. Si le nom de l'entité a un alias, alors n'importe quelle référence de propriété doit être qualifiée en ayant un alias ; si le nom de l'entité n'a pas d'alias, alors il est illégal pour n'importe quelle référence de propriété d'être qualifiée.
- Aucune jointure (implicite ou explicite) ne peut être spécifiée dans une requête HQL. Les sous-requêtes peuvent être utilisées dans la clause where ; les sous-requêtes, elles-mêmes, peuvent contenir des jointures.
- La clause where est aussi optionnelle.

Par exemple, pour exécuter un `UPDATE` HQL, utilisez la méthode `Query.executeUpdate()` (la méthode est données pour ceux qui sont familiers avec `PreparedStatement.executeUpdate()` de JDBC) :

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();

String hqlUpdate = "update Customer c set c.name = :newName where c.name = :oldName";
// ou String hqlUpdate = "update Customer set name = :newName where name = :oldName";
int updatedEntities = s.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

Par défaut, les statements HQL `UPDATE`, n'affectent pas la valeur des propriétés Section 5.1.7, « version (optionnel) » ou Section 5.1.8, « timestamp (optionnel) » pour les entités affectées; ceci est compatible avec la spec EJB3. Toutefois, vous pouvez forcer Hibernate à mettre à jour les valeurs des propriétés `version` ou `timestamp` en utilisant le `versioned update`. Pour se faire, ajoutez le mot clé `VERSIONED` après le mot clé `UPDATE`.

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
String hqlVersionedUpdate = "update versioned Customer set name = :newName where name = :oldName";
int updatedEntities = s.createQuery( hqlUpdate )
        .setString( "newName", newName )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

Notez que les types personnalisés (`org.hibernate.usertype.UserVersionType`) ne sont pas supportés en conjonction avec le statement `update versioned` statement.

Pour exécuter un HQL `DELETE`, utilisez la même méthode`Query.executeUpdate()`:

```
Session session = sessionFactory.openSession();
```

```
Transaction tx = session.beginTransaction();

String hqlDelete = "delete Customer c where c.name = :oldName";
// or String hqlDelete = "delete Customer where name = :oldName";
int deletedEntities = s.createQuery( hqlDelete )
        .setString( "oldName", oldName )
        .executeUpdate();
tx.commit();
session.close();
```

La valeur du `int` retourné par la méthode `Query.executeUpdate()` indique le nombre d'entités affectées par l'opération. Considérez que cela peut ou pas corréler le nombre de lignes affectés dans la base de données. Une opération HQL pourrait entraîner l'exécution de multiples expressions SQL réelles, pour des classes filles mappées par jointure (NdT: join-subclass), par exemple. Le nombre retourné indique le nombre d'entités réelles affectées par l'expression. Retour à l'exemple de la classe fille mappée par jointure, un effacement d'une des classes filles peut réellement entraîner des suppressions pas seulement dans la table qui mappe la classe fille, mais aussi dans la table "racine" et potentillement dans les tables des classes filles plus bas dans la hiérarchie d'héritage.

La pseudo-syntaxe pour l'expression `INSERT` est : `INSERT INTO EntityName properties_list select_statement`. Quelques points sont à noter :

- Seule la forme INSERT INTO ... SELECT ... est supportée ; pas la forme INSERT INTO ... VALUES ... .

  La properties_list est analogue à la `spécification de la colonne` The properties_list is analogous to the `column speficiation` dans l'expression SQL `INSERT`. Pour les entités impliquées dans un héritage mappé, seules les propriétés directement définies à ce niveau de classe donné peuvent être utilisées dans properties_list. Les propriétés de la classe mère ne sont pas permises ; et les propriétés des classes filles n'ont pas de sens. En d'autres mots, les expressions `INSERT` par nature non polymorphiques.
- select_statement peut être n'importe quelle requête de sélection HQl valide, avec l'avertissement que les types de retour doivent correspondre aux types attendus par l'insertion. Actuellement, c'est vérifié durant la compilation de la requête plutôt que la vérification soit reléguée à la base de données. Notez cependant que cela pourrait poser des problèmes entre les `Types` d'Hibernate qui sont *équivalents* opposé à *égaux*. Cela pourrait poser des problèmes avec des disparités entre une propriété définie comme un `org.hibernate.type.DateType` et une propriété définie comme un `org.hibernate.type.TimestampType`, même si la base de données ne ferait pas de distinction ou ne serait pas capable de gérer la conversion.
- Pour la propriéte id, l'expression d'insertion vous donne deux options. Vous pouvez soit spécifier explicitement la propriété id dans properties_list (auquel cas sa valeur est extraite de l'expression de sélection correspondante), soit l'omettre de properties_list (auquel cas une valeur générée est utilisée). Cette dernière option est seulement disponible en utilisant le générateur d'identifiant qui opère dans la base de données ; tenter d'utiliser cette option avec n'importe quel type de générateur "en mémoire" causera une exception durant l'analyse. Notez que pour les buts de cette discussion, les générateurs "en base" sont considérés être `org.hibernate.id.SequenceGenerator` (et ses classes filles) et n'importe quelles implémentations de `org.hibernate.id.PostInsertIdentifierGenerator`. L'exception la plus notable ici est `org.hibernate.id.TableHiLoGenerator`, qu ne peut pas être utilisée parce qu'il ne propose pas un moyen de d'exposer ses valeurs par un select.
- Pour des propriétés mappées comme `version` ou `timestamp`, l'expression d'insertion vous donne deux options. Vous pouvez soit spécifier la propriété dans properties_list (auquel cas sa valeur est extraite des expressions select correspondantes), soit l'omettre de properties_list (auquel cas la `valeur de graine` (NdT : seed value) définie par le `org.hibernate.type.VersionType` est utilisée).

Un exemple d'exécution d'une expression `INSERT` HQL :

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
```

```
String hqlInsert = "insert into DelinquentAccount (id, name) select c.id, c.name from Customer c where
int createdEntities = s.createQuery( hqlInsert )
        .executeUpdate();
tx.commit();
session.close();
```

Chapitre 14. HQL: Langage de requêtage d'Hibernate

Hibernate fourni un langage d'interrogation extrêmement puissant qui ressemble (et c'est voulu) au SQL. Mais ne soyez pas distraits par la syntaxe ; HQL est totalement orienté objet, comprenant des notions d'héritage, de polymorphisme et d'association.

14.1. Sensibilité à la casse

Les requêtes sont insensibles à la casse, à l'exception des noms des classes Java et des propriétés. Ainsi, `SeLeCT` est identique à `sELEct` et à `SELECT` mais `net.sf.hibernate.eg.FOO` n'est pas identique `net.sf.hibernate.eg.Foo` et `foo.barSet` n'est pas identique à `foo.BARSET`.

Ce guide utilise les mots clés HQL en minuscule. Certains utilisateurs trouvent les requêtes écrites avec les mots clés en majuscule plus lisibles, mais nous trouvons cette convention pénible lorsqu'elle est lue dans du code Java.

14.2. La clause from

La requête Hibernate la plus simple est de la forme :

```
from eg.Cat
```

qui retourne simplement toutes les instances de la classe `eg.Cat`. Nous n'avons pas besoin d'habitude de qualifier le nom de la classe, puisque `auto-import` est la valeur par défaut. Donc nous écrivons presque toujours :

```
from Cat
```

La plupart du temps, vous devrez assigner un *alias* puisque vous voudrez faire référence à `Cat` dans d'autres parties de la requête.

```
from Cat as cat
```

Cette requête assigne l'alias `cat` à l'instance `Cat`, nous pouvons donc utiliser cet alias ailleurs dans la requête. Le mot clé `as` est optionnel ; nous aurions pu écrire :

```
from Cat cat
```

Plusieurs classes peuvent apparaître, ce qui conduira à un produit cartésien (encore appelé jointures croisées).

```
from Formula, Parameter
```

```
from Formula as form, Parameter as param
```

C'est une bonne pratique que de nommer les alias dans les requêtes en utilisant l'initiale en miniscule, ce qui a le mérite d'être en phase avec les standards de nommage Java pour les variables locales (`domesticCat`).

14.3. Associations et jointures

On peut aussi assigner des alias à des entités associées, ou même aux éléments d'une collection de valeurs, en utilisant un `join` (jointure).

```
from Cat as cat
    inner join cat.mate as mate
    left outer join cat.kittens as kitten
```

```
from Cat as cat left join cat.mate.kittens as kittens
```

```
from Formula form full join form.parameter param
```

Les types de jointures supportées sont celles de ANSI SQL

- `inner join` (jointure fermée)
- `left outer join` (jointure ouverte par la gauche)
- `right outer join` (jointure ouverte par la droite)
- `full join` (jointure ouverte totalement - généralement inutile)

Les constructions des jointures `inner join`, `left outer join` et `right outer join` peuvent être abbrégées.

```
from Cat as cat
    join cat.mate as mate
    left join cat.kittens as kitten
```

Nous pouvons soumettre des conditions de jointure supplémentaires en utilisant le mot-clef HQL `with`.

```
from Cat as cat
    left join cat.kittens as kitten
        with kitten.bodyWeight > 10.0
```

Par ailleurs, une jointure "fetchée" (rapportée) permet d'initialiser les associations ou collections de valeurs en même temps que leur objet parent, le tout n'utilisant qu'un seul Select. Ceci est particulièrement utile dans le cas des collections. Ce système permet de surcharger les déclarations "lazy" et "outer-join" des fichiers de mapping pour les associations et collections. Voir Section 19.1, « Stratégies de chargement » pour plus d'informations.

```
from Cat as cat
    inner join fetch cat.mate
    left join fetch cat.kittens
```

Une jointure "fetchée" (rapportée) n'a généralement pas besoin de se voir assigner un alias puisque les objets associés n'ont pas à être utilisés dans les autres clauses. Notez aussi que les objets associés ne sont pas retournés directement dans le résultat de la requête mais l'on peut y accéder via l'objet parent. La seule raison pour laquelle nous pourrions avoir besoin d'un alias est si nous récupérions récursivement une collection supplémentaire :

```
from Cat as cat
    inner join fetch cat.mate
    left join fetch cat.kittens child
    left join fetch child.kittens
```

Notez que la construction de `fetch` ne peut pas être utilisée dans les requêtes appelées par `scroll()` ou `iterate()`. `fetch` ne devrait pas non plus être utilisé avec `setMaxResults()` ou `setFirstResult()`, ces

opérations étant basées sur le nombre de résultats qui contient généralement des doublons dès que des collections sont chargées. `fetch` ne peut pas non plus être utilisé avec une condition `with` ad hoc. Il est possible de créer un produit cartésien par jointure en récupérant plus d'une collection dans une requête, donc faites attention dans ce cas. Récupérer par jointure de multiples collections donne aussi parfois des résultats inattendus pour des mappings de bag, donc soyez prudent lorsque vous formulez vos requêtes dans de tels cas. Finalement, notez que `full join fetch` et `right join fetch` ne sont pas utiles en général.

Si vous utilisez un chargement retardé pour les propriétés (avec une instrumentation par bytecode), il est possible de forcer Hibernate à récupérer les propriétés non encore chargées immédiatement (dans la première requête) en utilisant `fetch all properties`.

```
from Document fetch all properties order by name
```

```
from Document doc fetch all properties where lower(doc.name) like '%cats%'
```

14.4. Formes de syntaxes pour les jointures

HQL supporte deux formes pour joindre les associations: `implicite` et `explicite`.

Les requêtes présentes dans la section précédente utilisent la forme `explicite` où le mode clé join est explicitement utilisé dans la clause from. C'est la forme recommandée.

La forme `implicite` n'utilise pas le mot clé join. A la place, les associations sont "déréférencées" en utilisant le notation '.'. Ces jointures peuvent apparaitre dans toutes les clauses. Les jointures `implicites` résultent en des inner join dans le SQL généré.

```
from Cat as cat where cat.mate.name like '%s%'
```

14.5. La clause select

La clause `select` sélectionne les objets et propriétés qui doivent être retournés dans le résultat de la requête. Soit :

```
select mate
from Cat as cat
    inner join cat.mate as mate
```

La requête recherchera les `mates` liés aux `Cats`. Vous pouvez explimer la requête d'une manière plus compacte :

```
select cat.mate from Cat cat
```

Les requêtes peuvent retourner des propriétés de n'importe quel type, même celles de type composant (component) :

```
select cat.name from DomesticCat cat
where cat.name like 'fri%'
```

```
select cust.name.firstName from Customer as cust
```

Les requêtes peuvent retourner plusieurs objets et/ou propriétés sous la forme d'un tableau du type `Object[]`,

```
select mother, offspr, mate.name
from DomesticCat as mother
```

```
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

ou sous la forme d'une `List`,

```
select new list(mother, offspr, mate.name)
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

ou sous la forme d'un objet Java typé,

```
select new Family(mother, mate, offspr)
from DomesticCat as mother
    join mother.mate as mate
    left join mother.kittens as offspr
```

à condition que la classe `Family` possède le constructeur approprié.

Vous pouvez assigner des alias aux expressions sélectionnées en utilisant `as` :

```
select max(bodyWeight) as max, min(bodyWeight) as min, count(*) as n
from Cat cat
```

C'est surtout utile lorsque c'est utilisé avec `select new map` :

```
select new map( max(bodyWeight) as max, min(bodyWeight) as min, count(*) as n )
from Cat cat
```

Cette requête retourne une `Map` à partir des alias vers les valeurs sélectionnées.

14.6. Fonctions d'aggrégation

Les requêtes HQL peuvent aussi retourner le résultat de fonctions d'aggrégation sur les propriétés :

```
select avg(cat.weight), sum(cat.weight), max(cat.weight), count(cat)
from Cat cat
```

Les fonctions supportées sont

- `avg(...), sum(...), min(...), max(...)`
- `count(*)`
- `count(...), count(distinct ...), count(all...)`

Vous pouvez utiliser des opérateurs arithmétiques, la concaténation, et des fonctions SQL reconnues dans la clause select :

```
select cat.weight + sum(kitten.weight)
from Cat cat
    join cat.kittens kitten
group by cat.id, cat.weight
```

```
select firstName||' '||initial||' '||upper(lastName) from Person
```

Les mots clé `distinct` et `all` peuvent être utilisés et ont la même signification qu'en SQL.

```
select distinct cat.name from Cat cat

select count(distinct cat.name), count(cat) from Cat cat
```

14.7. Requêtes polymorphiques

Une requête comme:

```
from Cat as cat
```

retourne non seuleument les instances de `Cat`, mais aussi celles des sous classes comme `DomesticCat`. Les requêtes Hibernate peuvent nommer n'importe quelle classe ou interface Java dans la clause `from`. La requête retournera les instances de toutes les classes persistantes qui étendent cette classe ou implémente cette interface. La requête suivante retournera tous les objets persistants :

```
from java.lang.Object o
```

L'interface `Named` peut être implémentée par plusieurs classes persistantes :

```
from Named n, Named m where n.name = m.name
```

Notez que ces deux dernières requêtes nécessitent plus d'un `SELECT` SQL. Ce qui signifie que la clause `order by` ne trie pas correctement la totalité des résultats (cela signifie aussi que vous ne pouvez exécuter ces requêtes en appelant `Query.scroll()`).

14.8. La clause where

La clause `where` vous permet de réduire la liste des instances retournées. Si aucun alias n'existe, vous pouvez vous référer aux propriétés par leur nom :

```
from Cat where name='Fritz'
```

S'il y a un alias, utilisez un nom de propriété qualifié :

```
from Cat as cat where cat.name='Fritz'
```

retourne les instances de `Cat` dont name est égale à 'Fritz'.

```
select foo
from Foo foo, Bar bar
where foo.startDate = bar.date
```

retournera les instances de `Foo` pour lesquelles il existe une instance de `bar` avec la propriété `date` est égale à la propriété `startDate` de `Foo`. Les expressions utilisant la navigation rendent la clause `where` extrêmement puissante. Soit :

```
from Cat cat where cat.mate.name is not null
```

Cette requête se traduit en SQL par une jointure interne à une table. Si vous souhaitez écrire quelque chose comme :

```
from Foo foo
```

```
where foo.bar.baz.customer.address.city is not null
```

vous finiriez avec une requête qui nécessiterait quatre jointures en SQL.

L'opérateur = peut être utilisé pour comparer aussi bien des propriétés que des instances :

```
from Cat cat, Cat rival where cat.mate = rival.mate
```

```
select cat, mate
from Cat cat, Cat mate
where cat.mate = mate
```

La propriété spéciale (en minuscule) `id` peut être utilisée pour faire référence à l'identifiant d'un objet (vous pouvez aussi utiliser le nom de cette propriété).

```
from Cat as cat where cat.id = 123

from Cat as cat where cat.mate.id = 69
```

La seconde requête est particulièrement efficace. Aucune jointure n'est nécessaire !

Les propriétés d'un identifiant composé peuvent aussi être utilisées. Supposez que `Person` ait un identifiant composé de `country` et `medicareNumber`.

```
from bank.Person person
where person.id.country = 'AU'
    and person.id.medicareNumber = 123456
```

```
from bank.Account account
where account.owner.id.country = 'AU'
    and account.owner.id.medicareNumber = 123456
```

Une fois de plus, la seconde requête ne nécessite pas de jointure.

De même, la propriété spéciale `class` interroge la valeur discriminante d'une instance dans le cas d'une persistance polymorphique. Le nom d'une classe Java incorporée dans la clause where sera traduite par sa valeur discriminante.

```
from Cat cat where cat.class = DomesticCat
```

Vous pouvez aussi spécifier les propriétés des composants ou types utilisateurs composés (components, composite user types etc). N'essayez jamais d'utiliser un expression de navigation qui se terminerait par une propriété de type composant (qui est différent d'une propriété d'un composant). Par exemple, si `store.owner` est une entité avec un composant `address`

```
store.owner.address.city    // okay
store.owner.address         // error!
```

Un type "any" possède les propriétés spéciales `id` et `class`, qui nous permettent d'exprimer une jointure de la manière suivante (où `AuditLog.item` est une propriété mappée avec `<any>`).

```
from AuditLog log, Payment payment
where log.item.class = 'Payment' and log.item.id = payment.id
```

Dans la requête précédente, notez que `log.item.class` et `payment.class` feraient référence à des valeurs de colonnes de la base de données complètement différentes.

14.9. Expressions

Les expressions permises dans la clause `where` incluent la plupart des choses que vous pouvez utiliser en SQL :

- opérateurs mathématiques `+, -, *, /`
- opérateur de comparaison binaire `=, >=, <=, <>, !=, like`
- opérateurs logiques `and, or, not`
- Parenthèses `( )`, indiquant un regroupement
- `in`, `not in`, `between`, `is null`, `is not null`, `is empty`, `is not empty`, `member of` and `not member of`
- "Simple" case, `case ... when ... then ... else ... end`, and "searched" case, `case when ... then ... else ... end`
- concatenation de chaîne de caractères `...||...` ou `concat(...,...)`
- `current_date()`, `current_time()`, `current_timestamp()`
- `second(...)`, `minute(...)`, `hour(...)`, `day(...)`, `month(...)`, `year(...)`,
- N'importe quel fonction ou opérateur défini par EJB-QL 3.0 : `substring(), trim(), lower(), upper(), length(), locate(), abs(), sqrt(), bit_length(), mod()`
- `coalesce()` et `nullif()`
- `str()` pour convertir des valeurs numériques ou temporelles vers une chaîne de caractères lisible
- `cast(... as ...)`, où le second argument est le nom d'un type Hibernate, et `extract(... from ...)` si le `cast()` ANSI et `extract()` sont supportés par la base de données sous-jacente
- La fonction HQL `index()`, qui s'applique aux alias d'une collection indexée jointe
- Les fonctions HQL qui s'appliquent expressions représentant des collections : `size(), minelement(), maxelement(), minindex(), maxindex()`, ainsi que les fonctions spéciales `elements()` et `indices` qui peuvent être quantifiées en utilisant `some, all, exists, any, in`.
- N'importe quelle fonction scalaire supportée par la base de données comme `sign()`, `trunc()`, `rtrim()`, `sin()`
- Les paramètres positionnels de JDBC `?`
- paramètres nommés `:name`, `:start_date`, `:x1`
- littéral SQL `'foo'`, `69`, `'1970-01-01 10:00:01.0'`
- Constantes Java `public static final eg.Color.TABBY`

`in` et `between` peuvent être utilisés comme suit :

```
from DomesticCat cat where cat.name between 'A' and 'B'
```

```
from DomesticCat cat where cat.name in ( 'Foo', 'Bar', 'Baz' )
```

et la forme négative peut être écrite

```
from DomesticCat cat where cat.name not between 'A' and 'B'
```

```
from DomesticCat cat where cat.name not in ( 'Foo', 'Bar', 'Baz' )
```

De même, `is null` et `is not null` peuvent être utilisés pour tester les valeurs nulle.

Les booléens peuvent être facilement utilisés en déclarant les substitutions de requêtes dans la configuration Hibernate :

```
<property name="hibernate.query.substitutions">true 1, false 0</property>
```

Ce qui remplacera les mots clés `true` et `false` par `1` et `0` dans la traduction SQL du HQL suivant :

```
from Cat cat where cat.alive = true
```

Vous pouvez tester la taille d'une collection par la propriété spéciale `size`, ou la fonction spéciale `size()`.

```
from Cat cat where cat.kittens.size > 0
```

```
from Cat cat where size(cat.kittens) > 0
```

Pour les collections indexées, vous pouvez faire référence aux indices minimum et maximum en utilisant les fonctions `minindex` and `maxindex`. De manière similaire, vous pouvez faire référence aux éléments minimum et maximum d'une collection de type basiques en utilisant les fonctions `minelement` et `maxelement`.

```
from Calendar cal where maxelement(cal.holidays) > current date
```

```
from Order order where maxindex(order.items) > 100
```

```
from Order order where minelement(order.items) > 10000
```

Les fonctions SQL `any, some, all, exists, in` supportent que leur soient passées l'élément, l'index d'une collection (fonctions `elements` et `indices`) ou le résultat d'une sous requête (voir ci dessous).

```
select mother from Cat as mother, Cat as kit
where kit in elements(foo.kittens)
```

```
select p from NameList list, Person p
where p.name = some elements(list.names)
```

```
from Cat cat where exists elements(cat.kittens)
```

```
from Player p where 3 > all elements(p.scores)
```

```
from Show show where 'fizard' in indices(show.acts)
```

Notez que l'écriture de - `size`, `elements`, `indices`, `minindex`, `maxindex`, `minelement`, `maxelement` - peuvent seulement être utilisée dans la clause where dans Hibernate3.

Les éléments de collections indexées (arrays, lists, maps) peuvent être référencés via index (dans une clause where seulement) :

```
from Order order where order.items[0].id = 1234
```

```
select person from Person person, Calendar calendar
where calendar.holidays['national day'] = person.birthDay
    and person.nationality.calendar = calendar
```

```
select item from Item item, Order order
where order.items[ order.deliveredItemIndices[0] ] = item and order.id = 11
```

```
select item from Item item, Order order
where order.items[ maxindex(order.items) ] = item and order.id = 11
```

L'expression entre `[]` peut même être une expression arithmétique.

```
select item from Item item, Order order
where order.items[ size(order.items) - 1 ] = item
```

HQL propose aussi une fonction `index()` interne, pour les éléments d'une association one-to-many ou d'une collections de valeurs.

```
select item, index(item) from Order order
    join order.items item
where index(item) < 5
```

Les fonctions SQL scalaires supportées par la base de données utilisée peuvent être utilisées

```
from DomesticCat cat where upper(cat.name) like 'FRI%'
```

Si vous n'êtes pas encore convaincu par tout cela, imaginez la taille et l'illisibilité qui caractériseraient la transformation SQL de la requête HQL suivante :

```
select cust
from Product prod,
    Store store
    inner join store.customers cust
where prod.name = 'widget'
    and store.location.name in ( 'Melbourne', 'Sydney' )
    and prod = all elements(cust.currentOrder.lineItems)
```

Un indice : cela donnerait quelque chose comme

```
SELECT cust.name, cust.address, cust.phone, cust.id, cust.current_order
FROM customers cust,
    stores store,
    locations loc,
    store_customers sc,
    product prod
WHERE prod.name = 'widget'
    AND store.loc_id = loc.id
    AND loc.name IN ( 'Melbourne', 'Sydney' )
    AND sc.store_id = store.id
    AND sc.cust_id = cust.id
    AND prod.id = ALL(
        SELECT item.prod_id
        FROM line_items item, orders o
        WHERE item.order_id = o.id
            AND cust.current_order = o.id
    )
```

14.10. La clause order by

La liste retounée par la requête peut être triée par n'importe quelle propriété de la classe ou du composant retourné :

```
from DomesticCat cat
order by cat.name asc, cat.weight desc, cat.birthdate
```

Le mot optionnel `asc` ou `desc` indique respectivement si le tri doit être croissant ou décroissant.

14.11. La clause group by

Si la requête retourne des valeurs aggrégées, celles-ci peuvent être groupées par propriété d'une classe retournée ou par composant :

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
```

```
select foo.id, avg(name), max(name)
from Foo foo join foo.names name
group by foo.id
```

Une clause `having` est aussi permise.

```
select cat.color, sum(cat.weight), count(cat)
from Cat cat
group by cat.color
having cat.color in (eg.Color.TABBY, eg.Color.BLACK)
```

Les fonctions SQL et les fonctions d'agrégat sont permises dans les clauses `having` et `order by`, si elles sont prises en charge par la base de données (ce que ne fait pas MySQL par exemple).

```
select cat
from Cat cat
    join cat.kittens kitten
group by cat.id, cat.name, cat.other, cat.properties
having avg(kitten.weight) > 100
order by count(kitten) asc, sum(kitten.weight) desc
```

Notez que ni la clause `group by` ni la clause `order by` ne peuvent contenir d'expressions arithmétiques. Notez aussi qu'Hibernate ne développe pas une entité faisant partie du regroupement, donc vous ne pouvez pas écrire `group by cat` si toutes les propriétés de `cat` sont non-agrégées. Vous devez lister toutes les propriétés non-agrégées explicitement.

14.12. Sous-requêtes

Pour les bases de données le supportant, Hibernate supporte les sous requêtes dans les requêtes. Une sous requête doit être entre parenthèses (souvent pour un appel à une fonction d'agrégation SQL) Même les sous requêtes corrélées (celles qui font référence à un alias de la requête principale) sont supportées.

```
from Cat as fatcat
where fatcat.weight > (
    select avg(cat.weight) from DomesticCat cat
)
```

```
from DomesticCat as cat
where cat.name = some (
    select name.nickName from Name as name
)
```

```
from Cat as cat
where not exists (
    from Cat as mate where mate.mate = cat
)
```

```
from DomesticCat as cat
where cat.name not in (
    select name.nickName from Name as name
)
```

```
select cat.id, (select max(kit.weight) from cat.kitten kit)
from Cat as cat
```

Notez que les sous-requêtes HQL peuvent arriver seulememnt dans les clauses select ou where.

Pour des sous-requêtes avec plus d'une expression dans le select, vous pouvez utiliser un constructeur de tuples :

```
from Cat as cat
where not ( cat.name, cat.color ) in (
    select cat.name, cat.color from DomesticCat cat
)
```

Notez que sur certaines bases de données (mais par Oracle ou HSQL), vous pouvez utiliser des constructeurs de tuples dans d'autres contextes, par exemple lors du requêtage de composants ou de types utilisateur composites :

```
from Person where name = ('Gavin', 'A', 'King')
```

Ce qui est équivalent à la forme plus verbeuse suivante :

```
from Person where name.first = 'Gavin' and name.initial = 'A' and name.last = 'King')
```

Il y a deux bonnes raisons que vous ne puissiez ne pas vouloir faire cette sorte de choses : d'abord, ce n'est pas complètement portable entre les plateformes de base de données ; deuxièmement, la requête est maintenant dépendante de l'ordre des propriétés dans le document de mapping.

14.13. Exemples HQL

Les requêtes Hibernate peuvent être relativement puissantes et complexes. En fait, la puissance du langage de requêtage est l'un des avantages principaux d'Hibernate. Voici quelques exemples très similaires aux requêtes que nous avons utilisées lors d'un récent projet. Notez que la plupart des requêtes que vous écrirez seront plus simples que les exemples suivantes !

La requête suivante retourne l'id de commande (order), le nombre d'articles (items) et la valeur totale de la commande (order) pour toutes les commandes non payées d'un client (customer) particulier pour un total minimum donné, le tout trié par la valeur totale. La requête SQL générée sur les tables `ORDER`, `ORDER_LINE`, `PRODUCT`, `CATALOG` et `PRICE` est composée de quatre jointures interne ainsi que d'une sous-requête (non corrélée).

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog.effectiveDate < sysdate
    and catalog.effectiveDate >= all (
        select cat.effectiveDate
        from Catalog as cat
        where cat.effectiveDate < sysdate
    )
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount) desc
```

Quel monstre ! En principe, nous ne sommes pas très fan des sous-requêtes, la requête ressemblait donc plutôt à cela :

```
select order.id, sum(price.amount), count(item)
from Order as order
    join order.lineItems as item
    join item.product as product,
    Catalog as catalog
    join catalog.prices as price
where order.paid = false
    and order.customer = :customer
    and price.product = product
    and catalog = :currentCatalog
group by order
having sum(price.amount) > :minAmount
order by sum(price.amount) desc
```

La requête suivante compte le nombre de paiements (payments) pour chaque status, en excluant les paiements dans le status `AWAITING_APPROVAL` où le changement de status le plus récent à été fait par l'utilisateur courant. En SQL, cette requête effectue deux jointures internes et des sous requêtes corrélées sur les tables `PAYMENT`, `PAYMENT_STATUS` et `PAYMENT_STATUS_CHANGE`.

```
select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
    join payment.statusChanges as statusChange
where payment.status.name <> PaymentStatus.AWAITING_APPROVAL
    or (
        statusChange.timeStamp = (
            select max(change.timeStamp)
            from PaymentStatusChange change
            where change.payment = payment
        )
        and statusChange.user <> :currentUser
    )
group by status.name, status.sortOrder
order by status.sortOrder
```

Si nous avions mappé la collection `statusChanges` comme une liste, au lieu d'un ensemble, la requête aurait été plus facile à écrire.

```
select count(payment), status.name
from Payment as payment
    join payment.currentStatus as status
where payment.status.name <> PaymentStatus.AWAITING_APPROVAL
    or payment.statusChanges[ maxIndex(payment.statusChanges) ].user <> :currentUser
group by status.name, status.sortOrder
order by status.sortOrder
```

La requête qui suit utilise la fonction de MS SQL `isNull()` pour retourner tous les comptes (accounts) et paiements (payments) impayés pour l'organisation à laquelle l'uilisateur (user) courant appartient. Elle est traduite en SQL par trois jointures internes, une jointure externe ainsi qu'une sous requête sur les tables `ACCOUNT`, `PAYMENT`, `PAYMENT_STATUS`, `ACCOUNT_TYPE`, `ORGANIZATION` et `ORG_USER`.

```
select account, payment
from Account as account
    left outer join account.payments as payment
where :currentUser in elements(account.holder.users)
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name, PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber, payment.dueDate
```

Pour d'autres base de données, nous aurions dû faire sans la sous-requête (corrélée).

```
select account, payment
from Account as account
    join account.holder.users as user
    left outer join account.payments as payment
```

```
where :currentUser = user
    and PaymentStatus.UNPAID = isNull(payment.currentStatus.name, PaymentStatus.UNPAID)
order by account.type.sortOrder, account.accountNumber, payment.dueDate
```

14.14. Mise à jour et suppression

HQL supporte maintenant les expressions `update`, `delete` et `insert ... select ...`. Voir Section 13.4, « Opérations de style DML » pour les détails.

14.15. Trucs & Astuces

Vous pouvez compter le nombre de résultats d'une requête sans les retourner :

```
( (Integer) session.iterate("select count(*) from ....").next() ).intValue()
```

Pour trier les résultats par la taille d'une collection, utilisez la requête suivante :

```
select usr.id, usr.name
from User as usr
    left join usr.messages as msg
group by usr.id, usr.name
order by count(msg)
```

Si votre base de données supporte les sous-requêtes, vous pouvez placer des conditions sur la taille de la sélection dans la clause where de votre requête:

```
from User usr where size(usr.messages) >= 1
```

Si votre base de données ne supporte pas les sous-requêtes, utilisez la requête suivante :

```
select usr.id, usr.name
from User usr.name
    join usr.messages msg
group by usr.id, usr.name
having count(msg) >= 1
```

Cette solution ne peut pas retourner un `User` avec zéro message à cause de la jointure interne, la forme suivante peut donc être utile :

```
select usr.id, usr.name
from User as usr
    left join usr.messages as msg
group by usr.id, usr.name
having count(msg) = 0
```

Les propriétés d'un JavaBean peuvent être injectées dans les paramètres nommés d'un requête :

```
Query q = s.createQuery("from foo Foo as foo where foo.name=:name and foo.size=:size");
q.setProperties(fooBean); // fooBean has getName() and getSize()
List foos = q.list();
```

Les collections sont paginables via l'utilisation de l'interface `Query` avec un filtre :

```
Query q = s.createFilter( collection, "" ); // the trivial filter
q.setMaxResults(PAGE_SIZE);
q.setFirstResult(PAGE_SIZE * pageNumber);
```

```
List page = q.list();
```

Les éléments d'une collection peuvent être triés ou groupés en utilisant un filtre de requête :

```
Collection orderedCollection = s.filter( collection, "order by this.amount" );
Collection counts = s.filter( collection, "select this.type, count(this) group by this.type" );
```

Vous pouvez récupérer la taille d'une collection sans l'initialiser :

```
( (Integer) session.iterate("select count(*) from ....").next() ).intValue();
```

Chapitre 15. Requêtes par critères

Hibernate offre une API d'interrogation par critères intuitive et extensible.

15.1. Créer une instance de `Criteria`

L'interface `net.sf.hibernate.Criteria` représente une requête sur une classe persistente donnée. La `Session` fournit les instances de `Criteria`.

```
Criteria crit = sess.createCriteria(Cat.class);
crit.setMaxResults(50);
List cats = crit.list();
```

15.2. Restriction du résultat

Un criterion (critère de recherche) est une instance de l'interface `org.hibernate.criterion.Criterion`. La classe `org.hibernate.criterion.Restrictions` définit des méthodes pour obtenir des types de `Criterion` pré-définis.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .add( Restrictions.between("weight", minWeight, maxWeight) )
    .list();
```

Les restrictions peuvent être goupées de manière logique.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .add( Restrictions.or(
        Restrictions.eq( "age", new Integer(0) ),
        Restrictions.isNull("age")
    ) )
    .list();
```

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.in( "name", new String[] { "Fritz", "Izi", "Pk" } ) )
    .add( Restrictions.disjunction()
        .add( Restrictions.isNull("age") )
        .add( Restrictions.eq("age", new Integer(0) ) )
        .add( Restrictions.eq("age", new Integer(1) ) )
        .add( Restrictions.eq("age", new Integer(2) ) )
    ) )
    .list();
```

Il y a plusieurs types de criterion pré-définis (sous classes de `Restriction`), mais l'une d'entre elle particulièrement utile vous permet de spécifier directement du SQL.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.sql("lower({alias}.name) like lower(?)", "Fritz%", Hibernate.STRING) )
    .list();
```

La zone `{alias}` sera remplacée par l'alias de colonne de l'entité que l'on souhaite intérroger.

Une autre approche pour obtenir un criterion est de le récupérer d'une instance de `Property`. Vous pouvez créer une `Property` en appelant `Property.forName()`.

```
Property age = Property.forName("age");
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.disjunction()
        .add( age.isNull() )
        .add( age.eq( new Integer(0) ) )
        .add( age.eq( new Integer(1) ) )
        .add( age.eq( new Integer(2) ) )
    ) )
    .add( Property.forName("name").in( new String[] { "Fritz", "Izi", "Pk" } ) )
    .list();
```

15.3. Trier les résultats

Vous pouvez trier les résultats en utilisant `org.hibernate.criterion.Order`.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "F%")
    .addOrder( Order.asc("name") )
    .addOrder( Order.desc("age") )
    .setMaxResults(50)
    .list();
```

```
List cats = sess.createCriteria(Cat.class)
    .add( Property.forName("name").like("F%") )
    .addOrder( Property.forName("name").asc() )
    .addOrder( Property.forName("age").desc() )
    .setMaxResults(50)
    .list();
```

15.4. Associations

Vous pouvez facilement spécifier des contraintes sur des entités liées, par des associations en utilisant `createCriteria()`.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "F%")
    .createCriteria("kittens")
        .add( Restrictions.like("name", "F%")
    .list();
```

Notez que la seconde `createCriteria()` retourne une nouvelle instance de `Criteria`, qui se rapporte aux éléments de la collection `kittens`.

La forme alternative suivante est utile dans certains cas.

```
List cats = sess.createCriteria(Cat.class)
    .createAlias("kittens", "kt")
    .createAlias("mate", "mt")
    .add( Restrictions.eqProperty("kt.name", "mt.name") )
    .list();
```

(`createAlias()` ne crée pas de nouvelle instance de `Criteria`.)

Notez que les collections kittens contenues dans les instances de `Cat` retournées par les deux précédentes requêtes ne sont *pas* pré-filtrées par les critères ! Si vous souhaitez récupérer uniquement les kittens qui correspondent à la criteria, vous devez utiliser `ResultTransformer`.

```
List cats = sess.createCriteria(Cat.class)
    .createCriteria("kittens", "kt")
        .add( Restrictions.eq("name", "F%") )
    .setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP)
    .list();
Iterator iter = cats.iterator();
while ( iter.hasNext() ) {
    Map map = (Map) iter.next();
    Cat cat = (Cat) map.get(Criteria.ROOT_ALIAS);
    Cat kitten = (Cat) map.get("kt");
}
```

15.5. Peuplement d'associations de manière dynamique

Vous pouvez spéficier au moment de l'exécution le peuplement d'une association en utilisant `setFetchMode()` (c'est-à-dire le chargement de celle-ci). Cela permet de surcharger les valeurs "lazy" et "outer-join" du mapping.

```
List cats = sess.createCriteria(Cat.class)
    .add( Restrictions.like("name", "Fritz%") )
    .setFetchMode("mate", FetchMode.EAGER)
    .setFetchMode("kittens", FetchMode.EAGER)
    .list();
```

Cette requête recherchera `mate` et `kittens` via les jointures externes. Voir Section 19.1, « Stratégies de chargement » pour plus d'informations.

15.6. Requêtes par l'exemple

La classe `org.hibernate.criterion.Example` vous permet de construire un critère suivant une instance d'objet donnée.

```
Cat cat = new Cat();
cat.setSex('F');
cat.setColor(Color.BLACK);
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .list();
```

Les propriétés de type version, identifiant et association sont ignorées. Par défaut, les valeurs null sont exclues.

Vous pouvez ajuster la stratégie d'utilisation de valeurs de l'`Exemple`.

```
Example example = Example.create(cat)
    .excludeZeroes()           //exclude zero valued properties
    .excludeProperty("color")  //exclude the property named "color"
    .ignoreCase()              //perform case insensitive string comparisons
    .enableLike();             //use like for string comparisons
List results = session.createCriteria(Cat.class)
    .add(example)
    .list();
```

Vous pouvez utiliser les "exemples" pour des critères sur les objets associés.

```
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .createCriteria("mate")
        .add( Example.create( cat.getMate() ) )
    .list();
```

15.7. Projections, agrégation et regroupement

La classe `org.hibernate.criterion.Projections` est une fabrique d'instances de `Projection`. Nous appliquons une projection sur une requête en appelant `setProjection()`.

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.rowCount() )
    .add( Restrictions.eq("color", Color.BLACK) )
    .list();
```

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount() )
        .add( Projections.avg("weight") )
        .add( Projections.max("weight") )
        .add( Projections.groupProperty("color") )
    )
    .list();
```

Il n'y a pas besoin de "group by" explicite dans une requête par critère. Certains types de projection sont définis pour être des *projections de regroupement*, lesquels apparaissent aussi dans la clause `group by` SQL.

Un alias peut optionnellement être assigné à une projection, ainsi la valeur projetée peut être référencée dans des restrictions ou des tris. Voici deux façons différentes de faire ça :

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.alias( Projections.groupProperty("color"), "colr" ) )
    .addOrder( Order.asc("colr") )
    .list();
```

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.groupProperty("color").as("colr") )
    .addOrder( Order.asc("colr") )
    .list();
```

Les méthodes `alias()` et `as()` enveloppe simplement une instance de projection dans une autre instance (aliasée) de `Projection`. Comme un raccourci, vous pouvez assignez un alias lorsque vous ajoutez la projection à la liste de projections :

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount(), "catCountByColor" )
        .add( Projections.avg("weight"), "avgWeight" )
        .add( Projections.max("weight"), "maxWeight" )
        .add( Projections.groupProperty("color"), "color" )
    )
    .addOrder( Order.desc("catCountByColor") )
    .addOrder( Order.desc("avgWeight") )
    .list();
```

```
List results = session.createCriteria(Domestic.class, "cat")
    .createAlias("kittens", "kit")
    .setProjection( Projections.projectionList()
        .add( Projections.property("cat.name"), "catName" )
        .add( Projections.property("kit.name"), "kitName" )
    )
    .addOrder( Order.asc("catName") )
    .addOrder( Order.asc("kitName") )
    .list();
```

Vous pouvez aussi utiliser `Property.forName()` pour formuler des projections :

```
List results = session.createCriteria(Cat.class)
    .setProjection( Property.forName("name") )
    .add( Property.forName("color").eq(Color.BLACK) )
    .list();
```

```
List results = session.createCriteria(Cat.class)
    .setProjection( Projections.projectionList()
        .add( Projections.rowCount().as("catCountByColor") )
        .add( Property.forName("weight").avg().as("avgWeight") )
        .add( Property.forName("weight").max().as("maxWeight") )
        .add( Property.forName("color").group().as("color" )
    )
    .addOrder( Order.desc("catCountByColor") )
    .addOrder( Order.desc("avgWeight") )
    .list();
```

15.8. Requêtes et sous-requêtes détachées

La classe `DetachedCriteria` vous laisse créer une requête en dehors de la portée de la session, et puis l'exécuter plus tard en utilisant n'importe quelle `Session` arbitraire.

```
DetachedCriteria query = DetachedCriteria.forClass(Cat.class)
    .add( Property.forName("sex").eq('F') );

Session session = ....;
Transaction txn = session.beginTransaction();
List results = query.getExecutableCriteria(session).setMaxResults(100).list();
txn.commit();
session.close();
```

Une `DetachedCriteria` peut aussi être utilisée pour exprimer une sous-requête. Des instances de criterion impliquant des sous-requêtes peuvent être obtenues via `Subqueries` ou `Property`.

```
DetachedCriteria avgWeight = DetachedCriteria.forClass(Cat.class)
    .setProjection( Property.forName("weight").avg() );
session.createCriteria(Cat.class)
    .add( Property.forName("weight).gt(avgWeight) )
    .list();
```

```
DetachedCriteria weights = DetachedCriteria.forClass(Cat.class)
    .setProjection( Property.forName("weight") );
session.createCriteria(Cat.class)
    .add( Subqueries.geAll("weight", weights) )
    .list();
```

Même des requêtes corrélées sont possibles :

```
DetachedCriteria avgWeightForSex = DetachedCriteria.forClass(Cat.class, "cat2")
    .setProjection( Property.forName("weight").avg() )
    .add( Property.forName("cat2.sex").eqProperty("cat.sex") );
session.createCriteria(Cat.class, "cat")
    .add( Property.forName("weight).gt(avgWeightForSex) )
    .list();
```

15.9. Requêtes par identifiant naturel

Pour la plupart des requêtes, incluant les requêtes par critère, le cache de requêtes n'est pas très efficace, parce que l'invalidation du cache de requêtes arrive trop souvent. Cependant, il y a une sorte spéciale de requête où

nous pouvons optimiser l'algorithme d'invalidation du cache : les recherches sur une clef naturelle constante. Dans certaines applications, cette sorte de requête se produit fréquemment. L'API de critère fournit une provision spéciale pour ce cas d'utilisation.

D'abord vous devriez mapper la clef naturelle de votre entité en utilisant `<natural-id>`, et activer l'utilisation du cache de second niveau.

```
<class name="User">
    <cache usage="read-write"/>
    <id name="id">
        <generator class="increment"/>
    </id>
    <natural-id>
        <property name="name"/>
        <property name="org"/>
    </natural-id>
    <property name="password"/>
</class>
```

Notez que cette fonctionnalité n'est pas prévue pour l'utilisation avec des entités avec des clefs naturelles *mutables*.

Ensuite, activez le cache de requête d'Hibernate.

Maintenant `Restrictions.naturalId()` nous permet de rendre l'utilisation de l'algorithme de cache plus efficace.

```
session.createCriteria(User.class)
    .add( Restrictions.naturalId()
        .set("name", "gavin")
        .set("org", "hb")
    ).setCacheable(true)
    .uniqueResult();
```

Chapitre 16. SQL natif

Vous pouvez aussi écrire vos requêtes dans le dialecte SQL natif de votre base de données. Ceci est utile si vous souhaitez utiliser les fonctionnalités spécifiques de votre base de données comme le mot clé `CONNECT` d'Oracle. Cette fonctionnalité offre par ailleurs un moyen de migration plus propre et doux d'une application basée sur SQL/JDBC vers une application Hibernate.

Hibernate3 vous permet de spécifier du SQL écrit à la main (incluant les procédures stockées) pour toutes les opérations de création, mise à jour, suppression et chargement.

16.1. Utiliser une `SQLQuery`

L'exécution des requêtes en SQL natif est contrôlée par l'interface `SQLQuery`, laquelle est obtenue en appelant `Session.createSQLQuery()`. Dans des cas extrêmement simples, nous pouvons utiliser la forme suivante :

16.1.1. Requêtes scalaires

La requête SQL la plus basique permet de récupérer une liste de (valeurs) scalaires.

```
sess.createSQLQuery("SELECT * FROM CATS").list();
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM CATS").list();
```

Ces deux requêtes retourneront un tableau d'objets (Object[]) avec les valeurs scalaires de chacune des colonnes de la table CATS. Hibernate utilisera le ResultSetMetadata pour déduire l'ordre et le type des valeurs scalaires retournées.

Pour éviter l'overhead lié à `ResultSetMetadata` ou simplement pour être plus explicite dans ce qui est retourné, vous pouvez utiliser `addScalar()`.

```
sess.createSQLQuery("SELECT * FROM CATS")
 .addScalar("ID", Hibernate.LONG)
 .addScalar("NAME", Hibernate.STRING)
 .addScalar("BIRTHDATE", Hibernate.DATE)
```

Cette requête spécifie:

- la chaîne de caractère SQL
- les colonnes et les types retournés

Cela retournera toujours un tableau d'objets, mais sans utiliser le `ResultSetMetdata`, mais récupèrera explicitement les colonnes ID, NAME and BIRTHDATE column étant de respectivement de type Long, String et Short, depuis le resultset sous jacent. Cela signifie aussi que seules ces colonnes seront retournées même si la requête utilise `*` et aurait pu retourner plus que les trois colonnes listées.

Il est possible de ne pas définir l'information sur le type pour toutes ou partie des calaires.

```
sess.createSQLQuery("SELECT * FROM CATS")
 .addScalar("ID", Hibernate.LONG)
 .addScalar("NAME")
 .addScalar("BIRTHDATE")
```

Il s'agit essentiellement de la même requête que précédemment, mais le `ResultSetMetaData` est utilisé pour

décider des types de NAME et BIRTHDATE alors que le type de ID est explicitement spécifié.

Les java.sql.Types retournés par le ResultSetMetaData sont mappés aux type Hibernate via le Dialect. Si un type spécifique n'est pas mappé ou est mappé à un type non souhaité, il est possible de personnaliser en invoquant `registerHibernateType` dans le Dialect.

16.1.2. Requêtes d'entités

Les requêtes précédentes ne retournaient que des valeurs scalaires, retournant basiquement que les valeurs brutes du resultset. Ce qui suit montre comment récupérer des entités depuis une requête native SQL, grâce à `addEntity()`.

```
sess.createSQLQuery("SELECT * FROM CATS").addEntity(Cat.class);
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE FROM CATS").addEntity(Cat.class);
```

Cette requête spécifie:

- La chaîne de caractère de requête SQL
- L'entité retournée par la requête

Avec Cat mappé comme classe avec les colonnes ID, NAME et BIRTHDATE, les requêtes précédentes retournent toutes deux une liste où chaque élément est une entité Cat.

Si l'entité est mappée avec un `many-to-one` vers une autre entité, il est requis de retourner aussi cette entité en exécutant la requête native, sinon une erreur "column not found" spécifique à la base de données sera soulevée. Les colonnes additionnelles seront automatiquement retournées en utilisant la notation *, mais nous préférons être explicites comme dans l'exemple suivant avec le `many-to-one` vers `Dog`:

```
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE, DOG_ID FROM CATS").addEntity(Cat.class);
```

Ceci permet à cat.getDog() de fonctionner normalement.

16.1.3. Gérer les associations et collections

Il est possible de charger agressivement `Dog` pour éviter le chargement de proxy qui signifie aller retour supplémentaire vers la base de données. Ceci est faisable via la méthode `addJoin()`, qui vous permet de joindre une association ou collection.

```
sess.createSQLQuery("SELECT c.ID, NAME, BIRTHDATE, DOG_ID, D_ID, D_NAME FROM CATS c, DOGS d WHERE c.DO
 .addEntity("cat", Cat.class)
 .addJoin("cat.dog");
```

Dans cet exemple, les `Cat` retournés auront leur propriété `dog` entièrement initialisées sans aucun aller/retour supplémentaire vers la base de données. Notez que nous avons ajouté un alias ("cat") pour être capable de spécifier la propriété cible de la jointure. Il est possible de faire la même jointure aggressive pour les collections, e.g. si le `Cat` a un one-to-many vers `Dog`.

```
sess.createSQLQuery("SELECT ID, NAME, BIRTHDATE, D_ID, D_NAME, CAT_ID FROM CATS c, DOGS d WHERE c.ID =
 .addEntity("cat", Cat.class)
 .addJoin("cat.dogs");
```

Nous arrivons aux limites de ce qui est possible avec les requêtes natives sans les modifier pour les rendre utilisables par Hibernate; les problèmes surviennent lorsque nous essayons de retourner des entités du même type ou lorsque les alias/colonnes par défaut ne sont plus suffisants..

16.1.4. Retour d'entités multiples

Jusqu'à présent, les colonnes du resultset sont supposées être les mêmes que les colonnes spécifiées dans les fichiers de mapping. Ceci peut être problématique pour les requêtes SQL qui effectuent de multiples jointures vers différentes tables, puisque les mêmes colonnes peuvent apparaître dans plus d'une table.

L'injection d'alias de colonne est requis pour la requête suivante (qui risque de ne pas fonctionner):

```
sess.createSQLQuery("SELECT c.*, m.*  FROM CATS c, CATS m WHERE c.MOTHER_ID = c.ID")
 .addEntity("cat", Cat.class)
 .addEntity("mother", Cat.class)
```

Le but de cette requête est de retourner deux instances de Cat par ligne, un chat et sa mère. Cela échouera puisqu'il y a conflit de nom puisqu'ils sont mappés au même nom de colonne et que sur certaines base de données, les alias de colonnes retournés seront plutôt de la forme "c.ID", "c.NAME", etc. qui ne sont pas égaux aux colonnes spécifiées dans les mappings ("ID" and "NAME").

La forme suivante n'est pas vulnérable à la duplication des noms de colonnes:

```
sess.createSQLQuery("SELECT {cat.*}, {mother.*}  FROM CATS c, CATS m WHERE c.MOTHER_ID = c.ID")
 .addEntity("cat", Cat.class)
 .addEntity("mother", Cat.class)
```

Cette requête spécifie:

- la requête SQL, avec des réceptacles pour qu'Hibernate injecte les alias de colonnes
- les entités retournés par la requête

Les notations {cat.*} et {mother.*} utilisées sont un équivalent à 'toutes les propriétés'. Alternativement, vous pouvez lister les colonnes explicitement, mais même pour ce cas, nous laissons Hibernate injecter les alias de colonne pour chaque propriété. Le réceptable pour un alias de colonne est simplement le nom de la propriété qualifié par l'alias de la table. Dans l'exemple suivant, nous récupérons les chats et leur mère depuis une table différentes (cat_log) de celle déclarée dans les mappings. Notez que nous pouvons aussi utiliser les alias de propriété dans la clause where si nous le voulons.

```
String sql = "SELECT ID as {c.id}, NAME as {c.name}, " +
         "BIRTHDATE as {c.birthDate}, MOTHER_ID as {c.mother}, {mother.*} " +
         "FROM CAT_LOG c, CAT_LOG m WHERE {c.mother} = c.ID";

List loggedCats = sess.createSQLQuery(sql)
        .addEntity("cat", Cat.class)
        .addEntity("mother", Cat.class).list()
```

16.1.4.1. Références d'alias et de propriété

Pour la plupart des cas précédents, l'injection d'alias est requis, mais pour les requêtes relatives à des mappings plus complexes, comme les propriétés composite, les discriminants d'héritage, les collections etc., il y a des alias spécifiques à utiliser pour permettre à Hibernate l'injection des bons alias.

Le tableau suivant montre les diverses possiblités d'utilisation d'injection d'alias. Note: les noms d'alias dans le résultat sont des exemples, chaque alias aura un nom unique et probablement différent lorsqu'ils seront utilisés.

Tableau 16.1. Nom d'injection d'alias

Description	Syntaxe	Exemple
Une propriété simple	`{[aliasname].[propertyname]}`	`A_NAME as {item.name}`
Une propriété composite	`{[aliasname].[componentname].[propertyname]}`	`CURRENCY as {item.amount.currency}, VALUE as {item.amount.value}`
Discriminateur d'une entité	`{[aliasname].class}`	`DISC as {item.class}`
Toutes les propriétés d'une entité	`{[aliasname].*}`	`{item.*}`
La clé d'une collection	`{[aliasname].key}`	`ORGID as {coll.key}`
L'id d'une collection	`{[aliasname].id}`	`EMPID as {coll.id}`
L'élément d'une collection	`{[aliasname].element}`	`XID as {coll.element}`
Propriété d'un élément de collection	`{[aliasname].element.[propertyname]}`	`NAME as {coll.element.name}`
Toutes les propriétés d'un élément de collection	`{[aliasname].element.*}`	`{coll.element.*}`
Toutes les propriétés d'une collection	`{[aliasname].*}`	`{coll.*}`

16.1.5. Retour d'objet n'étant pas des entités

Il est possible d'appliquer un ResultTransformer à une requête native SQL. Ce qui permet, par exemple, de retourner des entités non gérées.

```
sess.createSQLQuery("SELECT NAME, BIRTHDATE FROM CATS")
        .setResultTransformer(Transformers.aliasToBean(CatDTO.class))
```

Cette requête spécifie:

- une requête SQL
- un transformateur de résultat

La requête précédente retournera une liste de `CatDTO` qui auront été instanciés et dans lesquelles les valeurs de NAME et BIRTHNAME auront été injectées dans les propriétés ou champs correspondants.

16.1.6. Gérer l'héritage

Les requêtes natives SQL pour les entités prenant part à un héritage doivent inclure toutes les propriétés de la classe de base et de toutes ses sous classes.

16.1.7. Paramètres

Les requêtes natives SQL supportent aussi les paramètres nommés:

```
Query query = sess.createSQLQuery("SELECT * FROM CATS WHERE NAME like ?").addEntity(Cat.class);
List pusList = query.setString(0, "Pus%").list();

query = sess.createSQLQuery("SELECT * FROM CATS WHERE NAME like :name").addEntity(Cat.class);
List pusList = query.setString("name", "Pus%").list();
```

16.2. Requêtes SQL nommées

Les requêtes SQL nommées peuvent être définies dans le document de mapping et appelées exactement de la même manière qu'un requête HQL nommée. Dans ce cas, nous *n'avons pas besoin* d'appeler `addEntity()`.

```
<sql-query name="persons">
    <return alias="person" class="eg.Person"/>
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex}
    FROM PERSON person
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

```
List people = sess.getNamedQuery("persons")
    .setString("namePattern", namePattern)
    .setMaxResults(50)
    .list();
```

Les éléments `<return-join>` et `<load-collection>` sont respectivement utilisés pour lier des associations et définir des requêtes qui initialisent des collections.

```
<sql-query name="personsWith">
    <return alias="person" class="eg.Person"/>
    <return-join alias="address" property="person.mailingAddress"/>
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex},
           adddress.STREET AS {address.street},
           adddress.CITY AS {address.city},
           adddress.STATE AS {address.state},
           adddress.ZIP AS {address.zip}
    FROM PERSON person
    JOIN ADDRESS adddress
        ON person.ID = address.PERSON_ID AND address.TYPE='MAILING'
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

Une requête SQL nommée peut retourner une valeur scalaire. Vous devez spécifier l'alias de colonne et le type Hibernate utilisant l'élément `<return-scalar>` :

```
<sql-query name="mySqlQuery">
    <return-scalar column="name" type="string"/>
    <return-scalar column="age" type="long"/>
    SELECT p.NAME AS name,
           p.AGE AS age,
    FROM PERSON p WHERE p.NAME LIKE 'Hiber%'
</sql-query>
```

Vous pouvez externaliser les informations de mapping des résultats dans un élément `<resultset>` pour soit les réutiliser dans différentes requêtes nommées, soit à travers l'API `setResultSetMapping()`.

```
<resultset name="personAddress">
```

```
    <return alias="person" class="eg.Person"/>
    <return-join alias="address" property="person.mailingAddress"/>
</resultset>

<sql-query name="personsWith" resultset-ref="personAddress">
    SELECT person.NAME AS {person.name},
           person.AGE AS {person.age},
           person.SEX AS {person.sex},
           adddress.STREET AS {address.street},
           adddress.CITY AS {address.city},
           adddress.STATE AS {address.state},
           adddress.ZIP AS {address.zip}
    FROM PERSON person
    JOIN ADDRESS adddress
        ON person.ID = address.PERSON_ID AND address.TYPE='MAILING'
    WHERE person.NAME LIKE :namePattern
</sql-query>
```

16.2.1. Utilisation de return-property pour spécifier explicitement les noms des colonnes/alias

Avec `<return-property>` vous pouvez explicitement dire à Hibernate quels alias de colonne utiliser, plutot que d'employer la syntaxe `{}` pour laisser Hibernate injecter ses propres alias.

```
<sql-query name="mySqlQuery">
    <return alias="person" class="eg.Person">
        <return-property name="name" column="myName"/>
        <return-property name="age" column="myAge"/>
        <return-property name="sex" column="mySex"/>
    </return>
    SELECT person.NAME AS myName,
           person.AGE AS myAge,
           person.SEX AS mySex,
    FROM PERSON person WHERE person.NAME LIKE :name
</sql-query>
```

`<return-property>` fonctionne aussi avec de multiple colonnes. Cela résout une limitation de la syntaxe `{}` qui ne peut pas permettre une bonne granularité des propriétés multi-colonnes.

```
<sql-query name="organizationCurrentEmployments">
    <return alias="emp" class="Employment">
        <return-property name="salary">
            <return-column name="VALUE"/>
            <return-column name="CURRENCY"/>
        </return-property>
        <return-property name="endDate" column="myEndDate"/>
    </return>
        SELECT EMPLOYEE AS {emp.employee}, EMPLOYER AS {emp.employer},
        STARTDATE AS {emp.startDate}, ENDDATE AS {emp.endDate},
        REGIONCODE as {emp.regionCode}, EID AS {emp.id}, VALUE, CURRENCY
        FROM EMPLOYMENT
        WHERE EMPLOYER = :id AND ENDDATE IS NULL
        ORDER BY STARTDATE ASC
</sql-query>
```

Notez que dans cet exemple nous avons utilisé `<return-property>` en combinaison avec la syntaxe `{}` pour l'injection. Cela autorise les utilisateurs à choisir comment ils veulent référencer les colonnes et les propriétés.

Si votre mapping a un discriminant vous devez utiliser `<return-discriminator>` pour spécifier la colonne discriminante.

16.2.2. Utilisation de procédures stockées pour les requêtes

Hibernate 3 introduit le support des requêtes via procédures stockées et les fonctions. La documentation suivante est valable pour les deux. Les procédures stockées/fonctions doivent retourner l'ensemble de résultats en tant que premier paramètre sortant (NdT: "out-parameter") pour être capable de fonctionner avec Hibernate. Un exemple d'une telle procédure stockée en Oracle 9 et version supérieure :

```
CREATE OR REPLACE FUNCTION selectAllEmployments
    RETURN SYS_REFCURSOR
AS
    st_cursor SYS_REFCURSOR;
BEGIN
    OPEN st_cursor FOR
 SELECT EMPLOYEE, EMPLOYER,
 STARTDATE, ENDDATE,
 REGIONCODE, EID, VALUE, CURRENCY
 FROM EMPLOYMENT;
      RETURN  st_cursor;
 END;
```

Pour utiliser cette requête dans Hibernate vous avez besoin de la mapper via une requête nommée.

```
<sql-query name="selectAllEmployees_SP" callable="true">
    <return alias="emp" class="Employment">
        <return-property name="employee" column="EMPLOYEE"/>
        <return-property name="employer" column="EMPLOYER"/>
        <return-property name="startDate" column="STARTDATE"/>
        <return-property name="endDate" column="ENDDATE"/>
        <return-property name="regionCode" column="REGIONCODE"/>
        <return-property name="id" column="EID"/>
        <return-property name="salary">
            <return-column name="VALUE"/>
            <return-column name="CURRENCY"/>
        </return-property>
    </return>
    { ? = call selectAllEmployments() }
</sql-query>
```

Notez que les procédures stockées retournent, pour le moment, seulement des scalaires et des entités. `<return-join>` et `<load-collection>` ne sont pas supportés.

16.2.2.1. Règles/limitations lors de l'utilisation des procédures stockées

Pur utiliser des procédures stockées avec Hibernate, les procédures doivent suivre certaines règles. Si elles ne suivent pas ces règles, elles ne sont pas utilisables avec Hibernate. Si vous voulez encore utiliser ces procédures vous devez les exécuter via `session.connection()`. Les règles sont différentes pour chaque base de données, puisque les vendeurs de base de données ont des sémantiques/syntaxes différentes pour les procédures stockées.

Les requêtes de procédures stockées ne peuvent pas être paginées avec `setFirstResult()/setMaxResults()`.

Pour Oracle les règles suivantes s'appliquent :

- La procédure doit retourner un ensemble de résultats. Le prmeier paramètre d'une procédure doit être un `OUT` qui retourne un ensemble de résultats. Ceci est fait en retournant un `SYS_REFCURSOR` dans Oracle 9 ou 10. Dans Oracle vous avez besoin de définir un type `REF CURSOR`.

Pour Sybase ou MS SQL server les règles suivantes s'appliquent :

- La procédure doit retourner un ensemble de résultats. Notez que comme ces serveurs peuvent retourner de multiples ensembles de résultats et mettre à jour des compteurs, Hibernate itérera les résultats et prendra le

premier résultat qui est un ensemble de résultat comme valeur de retour. Tout le reste sera ignoré.
- Si vous pouvez activer `SET NOCOUNT ON` dans votre procédure, elle sera probablement plus efficace, mais ce n'est pas une obligation.

16.3. SQL personnalisé pour créer, mettre à jour et effacer

Hibernate3 peut utiliser des expression SQL personnalisées pour des opérations de création, de mise à jour, et de suppression. Les objets persistants les classes et les collections dans Hibernate contiennent déjà un ensemble de chaînes de caractères générées lors de la configuration (insertsql, deletesql, updatesql, etc). Les tages de mapping `<sql-insert>`, `<sql-delete>`, et `<sql-update>` surchargent ces chaînes de caractères :

```
<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <sql-insert>INSERT INTO PERSON (NAME, ID) VALUES ( UPPER(?), ? )</sql-insert>
    <sql-update>UPDATE PERSON SET NAME=UPPER(?) WHERE ID=?</sql-update>
    <sql-delete>DELETE FROM PERSON WHERE ID=?</sql-delete>
</class>
```

Le SQL est directement exécuté dans votre base de données, donc vous êtes libre d'utiliser le dialecte que vous souhaitez. Cela réduira bien sûr la portabilité de votre mapping si vous utilisez du SQL spécifique à votre base de données.

Les procédures stockées sont supportées si l'attribut `callable` est paramétré :

```
<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <sql-insert callable="true">{call createPerson (?, ?)}</sql-insert>
    <sql-delete callable="true">{? = call deletePerson (?)}</sql-delete>
    <sql-update callable="true">{? = call updatePerson (?, ?)}</sql-update>
</class>
```

L'ordre des paramètres positionnels est actuellement vital, car ils doivent être dans la même séquence qu'Hibernate les attend.

Vous pouvez voir l'ordre attendu en activant les journaux de debug pour le niveau `org.hibernate.persister.entity` level. Avec ce niveau activé, Hibernate imprimera le SQL statique qui est utilisé pour créer, mettre à jour, supprimer, etc. des entités. (Pour voir la séquence attendue, rappelez-vous de ne pas inclure votre SQL personnalisé dans les fichiers de mapping de manière à surcharger le SQL statique généré par Hibernate.)

Les procédures stockées sont dans la plupart des cas (lire : il vaut mieux le faire) requises pour retourner le nombre de lignes insérées/mises à jour/supprimées, puisque Hibernate fait quelques vérifications de succès lors de l'exécution de l'expression. Hibernate inscrit toujours la première expression comme un paramètre de sortie numérique pour les opérations CUD :

```
CREATE OR REPLACE FUNCTION updatePerson (uid IN NUMBER, uname IN VARCHAR2)
    RETURN NUMBER IS
BEGIN

    update PERSON
    set
        NAME = uname,
    where
```

```
        ID = uid;

    return SQL%ROWCOUNT;

END updatePerson;
```

16.4. SQL personnalisé pour le chargement

Vous pouvez aussi déclarer vos propres requêtes SQL (ou HQL) pour le chargement d'entité :

```
<sql-query name="person">
    <return alias="pers" class="Person" lock-mode="upgrade"/>
    SELECT NAME AS {pers.name}, ID AS {pers.id}
    FROM PERSON
    WHERE ID=?
    FOR UPDATE
</sql-query>
```

Ceci est juste une déclaration de requête nommée, comme vu plus tôt. Vous pouvez référencer cette requête nommée dans un mapping de classe :

```
<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <loader query-ref="person"/>
</class>
```

Ceci fonctionne même avec des procédures stockées.

Vous pouvez même définir une requête pour le chargement d'une collection :

```
<set name="employments" inverse="true">
    <key/>
    <one-to-many class="Employment"/>
    <loader query-ref="employments"/>
</set>
```

```
<sql-query name="employments">
    <load-collection alias="emp" role="Person.employments"/>
    SELECT {emp.*}
    FROM EMPLOYMENT emp
    WHERE EMPLOYER = :id
    ORDER BY STARTDATE ASC, EMPLOYEE ASC
</sql-query>
```

Vous pourriez même définir un chargeur d'entité qui charge une collection par jointure :

```
<sql-query name="person">
    <return alias="pers" class="Person"/>
    <return-join alias="emp" property="pers.employments"/>
    SELECT NAME AS {pers.*}, {emp.*}
    FROM PERSON pers
    LEFT OUTER JOIN EMPLOYMENT emp
        ON pers.ID = emp.PERSON_ID
    WHERE ID=?
</sql-query>
```

Chapitre 17. Filtrer les données

Hibernate3 fournit une nouvelle approche innovatrice pour gérer des données avec des règles de "visibilité". Un *filtre Hibernate* est un filtre global, nommé, paramétré qui peut être activé ou désactivé pour une session Hibernate particulière.

17.1. Filtres Hibernate

Hibernate3 ajoute la capacité de prédéfinir des critères de filtre et d'attacher ces filtres à une classe ou à une collection. Un critère de filtre est la faculté de définir une clause de restriction très similaire à l'attribut "where" existant disponible sur une classe et divers éléments d'une collection. Mis à part que ces conditions de filtre peuvent être paramétrées. L'application peut alors prendre la décision à l'exécution si des filtres donnés devraient être activés et quels devraient être leurs paramètres. Des filtres peuvent être utilisés comme des vues de base de données, mais paramétrées dans l'application.

Afin d'utiliser des filtres, ils doivent d'abord être définis, puis attachés aux éléments de mapping appropriés. Pour définir un filtre, utilisez l'élément `<filter-def/>` dans un élément `<hibernate-mapping/>` :

```
<filter-def name="myFilter">
    <filter-param name="myFilterParam" type="string"/>
</filter-def>
```

Puis, ce filtre peut être attaché à une classe :

```
<class name="myClass" ...>
    ...
    <filter name="myFilter" condition=":myFilterParam = MY_FILTERED_COLUMN"/>
</class>
```

ou à une collection :

```
<set ...>
    <filter name="myFilter" condition=":myFilterParam = MY_FILTERED_COLUMN"/>
</set>
```

ou même aux deux (ou à plusieurs de chaque) en même temps.

Les méthodes sur `Session` sont : `enableFilter(String filterName)`, `getEnabledFilter(String filterName)`, et `disableFilter(String filterName)`. Par défaut, les filtres *ne sont pas* activés pour une session donnée ; ils doivent être explicitement activés en appelant la méthode `Session.enabledFilter()`, laquelle retourne une instance de l'interface `Filter`. Utiliser le simple filtre défini au-dessus ressemblerait à :

```
session.enableFilter("myFilter").setParameter("myFilterParam", "some-value");
```

Notez que des méthodes sur l'interface org.hibernate.Filter autorisent le chaînage de beaucoup de méthodes communes d'Hibernate.

Un exemple complet, utilisant des données temporelles avec une structure de date d'enregistrement effectif :

```
<filter-def name="effectiveDate">
    <filter-param name="asOfDate" type="date"/>
</filter-def>

<class name="Employee" ...>
...
```

```
    <many-to-one name="department" column="dept_id" class="Department"/>
    <property name="effectiveStartDate" type="date" column="eff_start_dt"/>
    <property name="effectiveEndDate" type="date" column="eff_end_dt"/>
...
    <!--
        Note that this assumes non-terminal records have an eff_end_dt set to
        a max db date for simplicity-sake
    -->
    <filter name="effectiveDate"
            condition=":asOfDate BETWEEN eff_start_dt and eff_end_dt"/>
</class>

<class name="Department" ...>
...
    <set name="employees" lazy="true">
        <key column="dept_id"/>
        <one-to-many class="Employee"/>
        <filter name="effectiveDate"
                condition=":asOfDate BETWEEN eff_start_dt and eff_end_dt"/>
    </set>
</class>
```

Puis, afin de s'assurer que vous pouvez toujours récupérer les enregistrements actuellement effectifs, activez simplement le filtre sur la session avant de récupérer des données des employés :

```
Session session = ...;
session.enabledFilter("effectiveDate").setParameter("asOfDate", new Date());
List results = session.createQuery("from Employee as e where e.salary > :targetSalary")
         .setLong("targetSalary", new Long(1000000))
         .list();
```

Dans le HQL ci-dessus, bien que nous ayons seulement mentionné une contrainte de salaire sur les resultats, à cause du filtre activé, la requête retournera seulement les employés actuellement actifs qui ont un salaire supérieur à un million de dollars.

A noter : si vous prévoyez d'utiliser des filtres avec des jointures externes (soit à travers HQL, soit par le chargement) faites attention à la direction de l'expression de condition. Il est plus sûr de la positionner pour les jointures externes à gauche ; en général, placez le paramètre d'abord, suivi du(des) nom(s) de colonne après l'opérateur.

Chapitre 18. Mapping XML

Notez que cette fonctionnalité est expérimentale dans Hibernate 3.0 et est en développement extrêmement actif.

18.1. Travailler avec des données XML

Hibernate vous laisse travailler avec des données XML persistantes de la même manière que vous travaillez avec des POJOs persistants. Un arbre XML peut être vu comme une autre manière de représenter les données relationnelles au niveau objet, à la place des POJOs.

Hibernate supporte dom4j en tant qu'API pour la manipulation des arbres XML. Vous pouvez écrire des requêtes qui récupèrent des arbres dom4j à partie de la base de données, et avoir toutes les modifications que vous faites sur l'arbre automatiquement synchronisées dans la base de données. Vous pouvez même prendre un document XML, l'analyser en utilisant dom4j, et l'écrire dans la base de données via les opérations basiques d'Hibernate : `persist()`, `saveOrUpdate()`, `merge()`, `delete()`, `replicate()` (merge() n'est pas encore supporté).

Cette fonctionnalité a plusieurs applications dont l'import/export de données, l'externalisation d'entités via JMS ou SOAP et les rapports XSLT.

Un simple mapping peut être utilisé pour simultanément mapper les propriétés d'une classe et les noeuds d'un document XML vers la base de données, ou, si il n'y a pas de classe à mapper, il peut être utilisé juste pour mapper le XML.

18.1.1. Spécifier le mapping XML et le mapping d'une classe ensemble

Voici un exemple de mapping d'un POJO et du XML simultanément :

```
<class name="Account"
        table="ACCOUNTS"
        node="account">

    <id name="accountId"
            column="ACCOUNT_ID"
            node="@id"/>

    <many-to-one name="customer"
            column="CUSTOMER_ID"
            node="customer/@id"
            embed-xml="false"/>

    <property name="balance"
            column="BALANCE"
            node="balance"/>

    ...

</class>
```

18.1.2. Spécifier seulement un mapping XML

Voici un exemple dans lequel il n'y a pas de class POJO :

```
<class entity-name="Account"
        table="ACCOUNTS"
        node="account">
```

```
    <id name="id"
            column="ACCOUNT_ID"
            node="@id"
            type="string"/>

    <many-to-one name="customerId"
            column="CUSTOMER_ID"
            node="customer/@id"
            embed-xml="false"
            entity-name="Customer"/>

    <property name="balance"
            column="BALANCE"
            node="balance"
            type="big_decimal"/>

    ...

</class>
```

Ce mapping vous permet d'accéder aux données comme un arbre dom4j, ou comme un graphe de paire nom de propriété/valeur (`Maps` java). Les noms des propriétés sont des constructions purement logiques qui peuvent être référées des dans requêtes HQL.

18.2. Métadonnées du mapping XML

Plusieurs éléments du mapping Hibernate acceptent l'attribut `node`. Ceci vous permet de spécifier le nom d'un attribut XML ou d'un élément qui contient la propriété ou les données de l'entité. Le format de l'attribut `node` doit être un des suivants :

- `"element-name"` - mappe vers l'élément XML nommé
- `"@attribute-name"` - mappe vers l'attribut XML nommé
- `"."` - mappe vers le parent de l'élément
- `"element-name/@attribute-name"` - mappe vers l'élément nommé de l'attribut nommé

Pour des collections et de simples associations valuées, il y a un attribut `embed-xml` supplémentaire. Si `embed-xml="true"`, qui est la valeur par défaut, l'arbre XML pour l'entité associée (ou la collection des types de valeurs) sera embarquée directement dans l'arbre XML pour l'entité qui possède l'association. Sinon, si `embed-xml="false"`, alors seule la valeur de l'identifiant référencé apparaîtra dans le XML pour de simples associations de points, et les collections n'appraîtront simplement pas.

Vous devriez faire attention à ne pas laisser `embed-xml="true"` pour trop d'associations, puisque XML ne traite pas bien les liens circurlaires.

```
<class name="Customer"
        table="CUSTOMER"
        node="customer">

    <id name="id"
            column="CUST_ID"
            node="@id"/>

    <map name="accounts"
            node="."
            embed-xml="true">
        <key column="CUSTOMER_ID"
                not-null="true"/>
        <map-key column="SHORT_DESC"
                node="@short-desc"
                type="string"/>
```

```
        <one-to-many entity-name="Account"
                embed-xml="false"
                node="account"/>
    </map>

    <component name="name"
            node="name">
        <property name="firstName"
                node="first-name"/>
        <property name="initial"
                node="initial"/>
        <property name="lastName"
                node="last-name"/>
    </component>

    ...

</class>
```

dans ce cas, nous avons décidé d'embarquer la collection d'identifiants de compte, mais pas les données actuelles du compte. La requête HQL suivante :

```
from Customer c left join fetch c.accounts where c.lastName like :lastName
```

devrait retourner l'ensemble de données suivant :

```
<customer id="123456789">
    <account short-desc="Savings">987632567</account>
    <account short-desc="Credit Card">985612323</account>
    <name>
        <first-name>Gavin</first-name>
        <initial>A</initial>
        <last-name>King</last-name>
    </name>
    ...
</customer>
```

Si vous positionnez `embed-xml="true"` sur le mapping `<one-to-many>`, les données pourraient ressembler plus à ça :

```
<customer id="123456789">
    <account id="987632567" short-desc="Savings">
        <customer id="123456789"/>
        <balance>100.29</balance>
    </account>
    <account id="985612323" short-desc="Credit Card">
        <customer id="123456789"/>
        <balance>-2370.34</balance>
    </account>
    <name>
        <first-name>Gavin</first-name>
        <initial>A</initial>
        <last-name>King</last-name>
    </name>
    ...
</customer>
```

18.3. Manipuler des données XML

Relisons et mettons à jour des documents XML dans l'application. Nous faisons ça en obtenant une session dom4j :

```
Document doc = ....;

Session session = factory.openSession();
Session dom4jSession = session.getSession(EntityMode.DOM4J);
Transaction tx = session.beginTransaction();

List results = dom4jSession
    .createQuery("from Customer c left join fetch c.accounts where c.lastName like :lastName")
    .list();
for ( int i=0; i<results.size(); i++ ) {
    //add the customer data to the XML document
    Element customer = (Element) results.get(i);
    doc.add(customer);
}

tx.commit();
session.close();
```

```
Session session = factory.openSession();
Session dom4jSession = session.getSession(EntityMode.DOM4J);
Transaction tx = session.beginTransaction();

Element cust = (Element) dom4jSession.get("Customer", customerId);
for ( int i=0; i<results.size(); i++ ) {
    Element customer = (Element) results.get(i);
    //change the customer name in the XML and database
    Element name = customer.element("name");
    name.element("first-name").setText(firstName);
    name.element("initial").setText(initial);
    name.element("last-name").setText(lastName);
}

tx.commit();
session.close();
```

Il est extrêmement utile de combiner cette fonctionnalité avec l'opération `replicate()` d'Hibernate pour implémenter des imports/exports de données XML.

Chapitre 19. Améliorer les performances

19.1. Stratégies de chargement

Une *stratégie de chargement* est une stratégie qu'Hibernate va utiliser pour récupérer des objets associés si l'application à besoin de naviguer à travers une association. Les stratégies de chargement peuvent être déclarées dans les méta-données de l'outil de mapping objet relationnel ou surchargées par une requête de type HQL ou `Criteria` particulière.

Hibernate3 définit les stratégies de chargement suivantes :

- *Chargement par jointure* - Hibernate récupère l'instance associée ou la collection dans un même `SELECT`, en utilisant un `OUTER JOIN`.
- *Chargement par select* - Un second `SELECT` est utilisé pour récupérer l'instance associée ou la collection. A moins que vous ne désactiviez explicitement le chargement tardif en spécifiant `lazy="false"`, ce second select ne sera exécuté que lorsque vous accéderez réellement à l'association.
- *Chargement par sous-select* - Un second `SELECT` est utilisé pour récupérer les associations pour toutes les entités récupérées dans une requête ou un chargement préalable. A moins que vous ne désactiviez explicitement le chargement tardif en spécifiant `lazy="false"`, ce second select ne sera exécuté que lorsque vous accéderez réellement à l'association.
- *Chargement par lot* - Il s'agit d'une stratégie d'optimisation pour le chargement par select - Hibernate récupère un lot d'instances ou de collections en un seul `SELECT` en spécifiant une liste de clé primaire ou de clé étrangère.

Hibernate fait également la distinction entre :

- *Chargement immédiat* - Une association, une collection ou un attribut est chargé immédiatement lorsque l'objet auquel appartient cet élément est chargé.
- *Chargement tardif d'une collection* - Une collection est chargée lorque l'application invoque une méthode sur cette collection (il s'agit du mode de chargement par défaut pour les collections).
- *Chargement "super tardif" d'une collection* - les éléments de la collection sont récupérés individuellement depuis la base de données lorsque nécessaire. Hibernate essaie de ne pas charger toute la collection en mémoire sauf si cela est absolument nécessaire (bien adapté aux très grandes collections).
- *Chargement par proxy* - une association vers un seul objet est chargée lorsqu'une méthode autre que le getter sur l'identifiant est appelée sur l'objet associé.
- *Chargement "sans proxy"* - une association vers un seul objet est chargée lorsque l'on accède à cet objet. Par rapport au chargement par proxy, cette approche est moins tardif (l'association est quand même chargée même si on n'accède qu'à l'identifiant) mais plus transparente car il n'y a pas de proxy visible dans l'application. Cette approche requiert une instrumentation du bytecode à la compilation et est rarement nécessaire.
- *Chargement tardif des attributs* - Un attribut ou un objet associé seul est chargé lorsque l'on y accède. Cette approche requiert une instrumentation du bytecode à la compilation et est rarement nécessaire.

Nous avons ici deux notions orthogonales : *quand* l'association est chargée et *comment* (quelle requête SQL est utilisée). Il ne faut pas confondre les deux. Le mode de chargement est utilisé pour améliorer les performances. On peut utiliser le mode tardif pour définir un contrat sur quelles données sont toujours accessibles sur une instance détachée d'une classe particulière.

19.1.1. Travailler avec des associations chargées tardivement

Par défaut, Hibernate3 utilise le chargement tardif par select pour les collections et le chargement tardif par proxy pour les associations vers un seul objet. Ces valeurs par défaut sont valables pour la plupart des associations dans la plupart des applications.

Note : si vous définissez `hibernate.default_batch_fetch_size`, Hibernate va utiliser l'optimisation du chargement par lot pour le chargement tardif (cette optimisation peut aussi être activée à un niveau de granularité plus fin).

Cependant, le chargement tardif pose un problème qu'il faut connaitre. L'accès à une association définie comme "tardive", hors du contexte d'une session hibernate ouverte, va conduire à une exception. Par exemple :

```
s = sessions.openSession();
Transaction tx = s.beginTransaction();

User u = (User) s.createQuery("from User u where u.name=:userName")
    .setString("userName", userName).uniqueResult();
Map permissions = u.getPermissions();

tx.commit();
s.close();

Integer accessLevel = (Integer) permissions.get("accounts");  // Error!
```

Etant donné que la collection des permissions n'a pas été initialisée avant que la `Session` soit fermée, la collection n'est pas capable de se charger. *Hibernate ne supporte pas le chargement tardif pour des objets détachés*. La solution à ce problème est de déplacer le code qui lit la collection avant le "commit" de la transaction.

Une autre alternative est d'utiliser une collection ou une association non "tardive" en spécifiant `lazy="false"` dans le mapping de l'association. Cependant il est prévu que le chargement tardif soit utilisé pour quasiment toutes les collections ou associations. Si vous définissez trop d'associtions non "tardives" dans votre modèle objet, Hibernate va finir par devoir charger toute la base de données en mémoire à chaque transaction !

D'un autre côté, on veut souvent choisir un chargement par jointure (qui est par défaut non tardif) à la place du chargement par select dans une transaction particulière. Nous allons maintenant voir comment adapter les stratégies de chargement. Dans Hibernate3 les mécanismes pour choisir une stratégie de chargement sont identiques que l'on ait une association vers un objet simple ou vers une collection.

19.1.2. Personnalisation des stratégies de chargement

Le chargement par select (mode par défaut) est très vulnérable au problème du N+1 selects, du coup vous pouvez avoir envie d'activer le chargement par jointure dans les fichiers de mapping :

```
<set name="permissions"
            fetch="join">
    <key column="userId"/>
    <one-to-many class="Permission"/>
</set
```

```
<many-to-one name="mother" class="Cat" fetch="join"/>
```

La stratégie de chargement définie à l'aide du mot `fetch` dans les fichiers de mapping affecte :

- La récupération via `get()` ou `load()`
- La récupération implicite lorsque l'on navigue à travers une association
- Les requêtes de type `Criteria`
- Les requêtes HQL si l'on utilise le chargement par `subselect`

Quelle que soit la stratégie de chargement que vous utilisez, la partie du graphe d'objets qui est définie comme non "tardive" sera chargée en mémoire. Cela peut mener à l'exécution de plusieurs selects successifs pour une seule requête HQL.

On n'utilise pas souvent les documents de mapping pour adapter le chargement. Au lieu de cela, on conserve le comportement par défaut et on le surcharge pour une transaction particulière en utilisant `left join fetch` dans les requêtes HQL. Cela indique à hibernate à Hibernate de charger l'association de manière agressive lors du premier select en utilisant une jointure externe. Dans l'API Criteria vous pouvez utiliser la méthode `setFetchMode(FetchMode.JOIN)`

Si vous ne vous sentez pas prêt à modifier la stratégie de chargement utilisé par `get()` ou `load()`, vous pouvez juste utiliser une requête de type `Criteria` comme par exemple :

```
User user = (User) session.createCriteria(User.class)
                .setFetchMode("permissions", FetchMode.JOIN)
                .add( Restrictions.idEq(userId) )
                .uniqueResult();
```

(Il s'agit de l'équivalent pour Hibernate de ce que d'autres outils de mapping appellent un "fetch plan" ou "plan de chargement")

Une autre manière complètement différente d'éviter le problème des N+1 selects est d'utiliser le cache de second niveau.

19.1.3. Proxys pour des associations vers un seul objet

Le chargement tardif des collections est implémenté par Hibernate en utilisant ses propres implémentations pour des collections persistantes. Si l'on veut un chargement tardif pour des associations vers un seul objet métier il faut utiliser un autre mécanisme. L'entité qui est pointée par l'association doit être masquée derrière un proxy. Hibernate implémente l'initialisation tardive des proxys sur des objets persistents via une mise à jour à chaud du bytecode (à l'aide de l'excellente librairie CGLIB).

Par défaut, Hibernate génère des proxys (au démarrage) pour toutes les classes persistantes et les utilise pour activer le chargement tardif des associations `many-to-one` et `one-to-one`.

Le fichier de mapping peut déclarer une interface qui sera utilisée par le proxy d'interfaçage pour cette classe à l'aide de l'attribut `proxy`. Par défaut Hibernate utilises une sous classe de la classe persistante. *Il faut que les classes pour lesquelles on ajoute un proxy implémentent un constructeur par défaut de visibilité au moins package. Ce constructeur est recommandé pour toutes les classes persistantes !*

Il y a quelques précautions à prendre lorsque l'on étend cette approche à des classes polymorphiques, exemple :

```
<class name="Cat" proxy="Cat">
```

```
        ......
        <subclass name="DomesticCat" proxy="DomesticCat">
            .....
        </subclass>
    </class>
```

Tout d'abord, les instances de `Cat` ne pourront jamais être "castées" en `DomesticCat`, même si l'instance sous jacente est une instance de `DomesticCat` :

```
Cat cat = (Cat) session.load(Cat.class, id);  // instancie un proxy (n'interroge pas la base de donnée
if ( cat.isDomesticCat() ) {                  // interroge la base de données pour initialiser le prox
    DomesticCat dc = (DomesticCat) cat;       // Erreur !
    ....
}
```

Deuxièmement, il est possible de casser la notion d'== des proxy.

```
Cat cat = (Cat) session.load(Cat.class, id);           // instancie un proxy Cat
DomesticCat dc =
    (DomesticCat) session.load(DomesticCat.class, id);  // acquiert un nouveau proxy DomesticCat
System.out.println(cat==dc);                           // faux
```

Cette situation n'est pas si mauvaise qu'il n'y parait. Même si nous avons deux références à deux objets proxys différents, l'instance de base sera quand même le même objet :

```
cat.setWeight(11.0);  // interroge la base de données pour initialiser le proxy
System.out.println( dc.getWeight() );  // 11.0
```

Troisièmement, vous ne pourrez pas utiliser un proxy CGLIB pour une classe `final` ou pour une classe contenant la moindre méthode `final`.

Enfin, si votre objet persistant obtient une ressource à l'instanciation (par example dans les initialiseurs ou dans le contructeur par défaut), alors ces ressources seront aussi obtenues par le proxy. La classe proxy est vraiment une sous classe de la classe persistante.

Ces problèmes sont tous dus aux limitations fondamentales du modèle d'héritage unique de Java. Si vous souhaitez éviter ces problèmes, vos classes persistantes doivent chacune implémenter une interface qui déclare ses méthodes métier. Vous devriez alors spécifier ces interfaces dans le fichier de mapping :

```
<class name="CatImpl" proxy="Cat">
    ......
    <subclass name="DomesticCatImpl" proxy="DomesticCat">
        .....
    </subclass>
</class>
```

où `CatImpl` implémente l'interface `Cat` et `DomesticCatImpl` implémente l'interface `DomesticCat`. Ainsi, des proxys pour les instances de `Cat` et `DomesticCat` pourraient être retournées par `load()` ou `iterate()` (Notez que `list()` ne retourne généralement pas de proxy).

```
Cat cat = (Cat) session.load(CatImpl.class, catid);
Iterator iter = session.iterate("from CatImpl as cat where cat.name='fritz'");
Cat fritz = (Cat) iter.next();
```

Les relations sont aussi initialisées tardivement. Ceci signifie que vous devez déclarer chaque propriété comme étant de type `Cat`, et non `CatImpl`.

Certaines opérations ne nécessitent pas l'initialisation du proxy

- `equals()`, si la classe persistante ne surcharge pas `equals()`
- `hashCode()`, si la classe persistante ne surcharge pas `hashCode()`
- Le getter de l'identifiant

Hibernate détectera les classes qui surchargent `equals()` ou `hashCode()`.

Eh choisissant `lazy="no-proxy"` au lieu de `lazy="proxy"` qui est la valeur par défaut, il est possible d'éviter les problèmes liés au transtypage. Il faudra alors une instrumentation du bytecode à la compilation et toutes les opérations résulterons immédiatement en une initialisation du proxy.

19.1.4. Initialisation des collections et des proxys

Une exception de type `LazyInitializationException` sera renvoyée par hibernate si une collection ou un proxy non initialisé est accédé en dehors de la portée de la `Session`, e.g. lorsque l'entité à laquelle appartient la collection ou qui a une référence vers le proxy est dans l'état "détachée".

Parfois, nous devons nous assurer qu'un proxy ou une collection est initialisée avant de fermer la `Session`. Bien sûr, nous pouvons toujours forcer l'initialisation en appelant par exemple `cat.getSex()` ou `cat.getKittens().size()`. Mais ceci n'est pas très lisible pour les personnes parcourant le code et n'est pas très générique.

Les méthodes statiques `Hibernate.initialize()` et `Hibernate.isInitialized()` fournissent à l'application un moyen de travailler avec des proxys ou des collections initialisés. `Hibernate.initialize(cat)` forcera l'initialisation d'un proxy de `cat`, si tant est que sa `Session` est ouverte. `Hibernate.initialize( cat.getKittens() )` a le même effet sur la collection kittens.

Une autre option est de conserver la `Session` ouverte jusqu'à ce que toutes les collections et tous les proxys aient été chargés. Dans certaines architectures applicatives, particulièrement celles ou le code d'accès aux données via hiberante et le code qui utilise ces données sont dans des couches applicatives différentes ou des processus physiques différents, il peut devenir problématique de garantir que la `Session` est ouverte lorsqu'une collection est initialisée. Il y a deux moyens de traiter ce problème :

- Dans une application web, un filtre de servlet peut être utilisé pour fermer la `Session` uniquement lorsque la requête a été entièrement traitée, lorsque le rendu de la vue est fini (il s'agit du pattern *Open Session in View*). Bien sûr, cela demande plus d'attention à la bonne gestion des exceptions de l'application. Il est d'une importance vitale que la `Session` soit fermée et la transaction terminée avant que l'on rende la main à l'utilisateur même si une exception survient durant le traitement de la vue. Voir le wiki Hibernate pour des exemples sur le pattern "Open Session in View".

- Dans une application avec une couche métier séparée, la couche contenant la logique métier doit "préparer" toutes les collections qui seront nécessaires à la couche web avant de retourner les données. Cela signifie que la couche métier doit charger toutes les données et retourner toutes les données déjà initialisées à la couche de présentation/web pour un cas d'utilisation donné. En général l'application appelle la méthode `Hibernate.initialize()` pour chaque collection nécessaire dans la couche web (cet appel doit être fait avant la fermeture de la session) ou bien récupère les collections de manière agressive à l'aide d'une requête HQL avec une clause `FETCH` ou à l'aide du mode `FetchMode.JOIN` pour une requête de type `Criteria`. Cela est en général plus facile si vous utilisez le pattern *Command* plutôt que *Session Facade*.

- Vous pouvez également attacher à une `Session` un objet chargé au préalable à l'aide des méthodes `merge()` ou `lock()` avant d'accéder aux collections (ou aux proxys) non initialisés. Non, Hibernate ne fait pas, et ne doit pas faire, cela automatiquement car cela pourrait introduire une sémantique transactionnelle ad hoc.

Parfois, vous ne voulez pas initialiser une grande collection mais vous avez quand même besoin d'informations

sur elle (comme sa taille) ou un sous ensemble de ses données

Vous pouvez utiliser un filtre de collection pour récupérer sa taille sans l'initialiser :

```
( (Integer) s.createFilter( collection, "select count(*)" ).list().get(0) ).intValue()
```

La méthode `createFilter()` est également utilisée pour récupérer de manière efficace des sous ensembles d'une collection sans avoir besoin de l'initialiser dans son ensemble.

```
s.createFilter( lazyCollection, "").setFirstResult(0).setMaxResults(10).list();
```

19.1.5. Utiliser le chargement par lot

Pour améliorer les performances, Hibernate peut utiliser le chargement par lot ce qui veut dire qu'Hibernate peut charger plusieurs proxys (ou collections) non initialisés en une seule requête lorsque l'on accède à l'un de ces proxys. Le chargement par lot est une optimisation intimement liée à la stratégie de chargement tardif par select. Il y a deux moyens d'activer le chargement par lot : au niveau de la classe et au niveau de la collection.

Le chargement par lot pour les classes/entités est plus simple à comprendre. Imaginez que vous ayez la situation suivante à l'exécution : vous avez 25 instances de `Cat` chargées dans une `Session`, chaque `Cat` a une référence à son `owner`, une `Person`. La classe `Person` est mappée avec un proxy, `lazy="true"`. Si vous itérez sur tous les cats et appelez `getOwner()` sur chacun d'eux, Hibernate exécutera par défaut 25 `SELECT`, pour charger les owners (initialiser le proxy). Vous pouvez paramétrer ce comportement en spécifiant une `batch-size` (taille du lot) dans le mapping de `Person` :

```
<class name="Person" batch-size="10">...</class>
```

Hibernate exécutera désormais trois requêtes, en chargeant respectivement 10, 10, et 5 entités.

Vous pouvez aussi activer le chargement par lot pour les collections. Par exemple, si chaque `Person` a une collection chargée tardivement de `Cats`, et que 10 personnes sont actuellement chargées dans la `Session`, itérer sur toutes les persons générera 10 `SELECTs`, un pour chaque appel de `getCats()`. Si vous activez le chargement par lot pour la collection `cats` dans le mapping de `Person`, Hibernate pourra précharger les collections :

```
<class name="Person">
    <set name="cats" batch-size="3">
        ...
    </set>
</class>
```

Avec une taille de lot (`batch-size`) de 8, Hibernate chargera respectivement 3, 3, 3, et 1 collections en quatre `SELECTs`. Encore une fois, la valeur de l'attribut dépend du nombre de collections non initialisées dans une `Session` particulière.

Le chargement par lot de collections est particulièrement utile si vous avez des arborescenses récursives d'éléments (typiquement, le schéma facture de matériels). (Bien qu'un *sous ensemble* ou un *chemin matérialisé* est sans doute une meilleure option pour des arbres principalement en lecture.)

19.1.6. Utilisation du chargement par sous select

Si une collection ou un proxy vers un objet doit être chargé, Hibernate va tous les charger en ré-exécutant la requête orignial dans un sous select. Cela fonctionne de la même manière que le chargement par lot sans la possibilité de fragmenter le chargement.

19.1.7. Utiliser le chargement tardif des propriétés

Hibernate3 supporte le chargement tardif de propriétés individuelles. La technique d'optimisation est également connue sous le nom de *fetch groups* (groupes de chargement). Il faut noter qu'il s'agit principalement d'une fonctionnalité marketing car en pratique l'optimisation de la lecture d'un enregistrement est beaucoup plus importante que l'optimisation de la lecture d'une colonne. Cependant, la restriction du chargement à certaines colonnes peut être pratique dans des cas extrèmes, lorsque des tables "legacy" possèdent des centaines de colonnes et que le modèle de données ne peut pas être amélioré.

Pour activer le chargement tardif d'une propriété, il faut mettre l'attribut `lazy` sur une propriété particulière du mapping :

```
<class name="Document">
       <id name="id">
        <generator class="native"/>
    </id>
    <property name="name" not-null="true" length="50"/>
    <property name="summary" not-null="true" length="200" lazy="true"/>
    <property name="text" not-null="true" length="2000" lazy="true"/>
</class>
```

Le chargement tardif des propriétés requiert une instrumentation du bytecode lors de la compilation ! Si les classes persistantes ne sont pas instrumentées, Hibernate ignorera de manière silencieuse le mode tardif et retombera dans le mode de chargement immédiat.

Pour l'instrumentation du bytecode vous pouvez utiliser la tâche Ant suivante :

```
<target name="instrument" depends="compile">
    <taskdef name="instrument" classname="org.hibernate.tool.instrument.InstrumentTask">
        <classpath path="${jar.path}"/>
        <classpath path="${classes.dir}"/>
        <classpath refid="lib.class.path"/>
    </taskdef>

    <instrument verbose="true">
        <fileset dir="${testclasses.dir}/org/hibernate/auction/model">
            <include name="*.class"/>
        </fileset>
    </instrument>
</target>
```

Une autre façon (meilleure ?) pour éviter de lire plus de colonnes que nécessaire au moins pour des transactions en lecture seule est d'utiliser les fonctionnalités de projection des requêtes HQL ou Criteria. Cela évite de devoir instrumenter le bytecode à la compilation et est certainement une solution préférable.

Vous pouvez forcer le mode de chargement agressif des propriétés en utilisant `fetch all properties` dans les requêts HQL.

19.2. Le cache de second niveau

Une `Session` Hibernate est un cache de niveau transactionnel des données persistantes. Il est possible de configurer un cache de cluster ou de JVM (de niveau `SessionFactory` pour être exact) défini classe par classe et collection par collection. Vous pouvez même utiliser votr choix de cache en implémentant le pourvoyeur (provider) associé. Faites attention, les caches ne sont jamais avertis des modifications faites dans la base de données par d'autres applications (ils peuvent cependant être configurés pour régulièrement expirer les données en cache).

Par défaut, Hibernate utilise EHCache comme cache de niveau JVM (le support de JCS est désormais déprécié et sera enlevé des futures versions d'Hibernate). Vous pouvez choisir une autre implémentation en spécifiant le nom de la classe qui implémente `org.hibernate.cache.CacheProvider` en utilisant la propriété `hibernate.cache.provider_class`.

Tableau 19.1. Fournisseur de cache

Cache	Classe pourvoyeuse	Type	Support en Cluster	Cache de requêtes supporté
Hashtable (ne pas utiliser en production)	`org.hibernate.cache.HashtableCacheProvider`	mémoire		oui
EHCache	`org.hibernate.cache.EhCacheProvider`	mémoire, disque		oui
OSCache	`org.hibernate.cache.OSCacheProvider`	mémoire, disque		oui
SwarmCache	`org.hibernate.cache.SwarmCacheProvider`	en cluster (multicast ip)	oui (invalidation de cluster)	
JBoss TreeCache	`org.hibernate.cache.TreeCacheProvider`	en cluster (multicast ip), transactionnel	oui (replication)	oui (horloge sync. nécessaire)

19.2.1. Mapping de Cache

L'élément `<cache>` d'une classe ou d'une collection à la forme suivante :

```
<cache
    usage="transactional|read-write|nonstrict-read-write|read-only"  (1)
    region="RegionName"                                              (2)
    include="all|non-lazy"                                           (3)
/>
```

(1) `usage` (requis) spécifie la stratégie de cache : `transactionel`, `lecture-écriture`, `lecture-écriture non stricte` ou `lecture seule`

(2) `region` (optionnel, par défaut il s'agit du nom de la classe ou du nom de role de la collection) spécifie le nom de la région du cache de second niveau

(3) `include` (optionnel, par défaut `all`) `non-lazy` spécifie que les propriétés des entités mappées avec `lazy="true"` ne doivent pas être mises en cache lorsque le chargement tardif des attributs est activé.

Alternativement (voir préférentiellement), vous pouvez spécifier les éléments `<class-cache>` et `<collection-cache>` dans `hibernate.cfg.xml`.

L'attribut `usage` spécifie une *stratégie de concurrence d'accès au cache*.

19.2.2. Strategie : lecture seule

Si votre application a besoin de lire mais ne modifie jamais les instances d'une classe, un cache `read-only` peut être utilisé. C'est la stratégie la plus simple et la plus performante. Elle est même parfaitement sûre dans un cluster.

```
<class name="eg.Immutable" mutable="false">
    <cache usage="read-only"/>
    ....
</class>
```

19.2.3. Stratégie : lecture/écriture

Si l'application a besoin de mettre à jour des données, un cache `read-write` peut être approprié. Cette stratégie ne devrait jamais être utilisée si votre application nécessite un niveau d'isolation transactionnelle sérialisable. Si le cache est utilisé dans un environnement JTA, vous devez spécifier `hibernate.transaction.manager_lookup_class`, fournissant une stratégie pour obtenir le `TransactionManager` JTA. Dans d'autres environnements, vous devriez vous assurer que la transation est terminée à l'appel de `Session.close()` ou `Session.disconnect()`. Si vous souhaitez utiliser cette stratégie dans un cluster, vous devriez vous assurer que l'implémentation de cache utilisée supporte le vérrouillage. Ce que ne font *pas* les pourvoyeurs caches fournis.

```
<class name="eg.Cat" .... >
    <cache usage="read-write"/>
    ....
    <set name="kittens" ... >
        <cache usage="read-write"/>
        ....
    </set>
</class>
```

19.2.4. Stratégie : lecture/écriture non stricte

Si l'application besoin de mettre à jour les données de manière occasionnelle (qu'il est très peu probable que deux transactions essaient de mettre à jour le même élément simultanément) et qu'une isolation transactionnelle stricte n'est pas nécessaire, un cache `nonstrict-read-write` peut être approprié. Si le cache est utilisé dans un environnement JTA, vous devez spécifier `hibernate.transaction.manager_lookup_class`. Dans d'autres environnements, vous devriez vous assurer que la transation est terminée à l'appel de `Session.close()` ou `Session.disconnect()`

19.2.5. Stratégie : transactionelle

La stratégie de cache `transactional` supporte un cache complètement transactionnel comme, par exemple, JBoss TreeCache. Un tel cache ne peut être utilisé que dans un environnement JTA et vous devez spécifier `hibernate.transaction.manager_lookup_class`.

Aucun des caches livrés ne supporte toutes les stratégies de concurrence. Le tableau suivant montre quels caches sont compatibles avec quelles stratégies de concurrence.

Tableau 19.2. Stratégie de concurrence du cache

Cache	read-only (lecture seule)	nonstrict-read-write (lecture-écriture non stricte)	read-write (lecture-ériture)	transactional (transactionnel)
Hashtable (ne pas utilser en production)	oui	oui	oui	
EHCache	oui	oui	oui	
OSCache	oui	oui	oui	
SwarmCache	oui	oui		
JBoss TreeCache	oui			oui

19.3. Gérer les caches

A chaque fois que vous passez un objet à la méthode `save()`, `update()` ou `saveOrUpdate()` et à chaque fois que vous récupérez un objet avec `load()`, `get()`, `list()`, `iterate()` or `scroll()`, cet objet est ajouté au cache interne de la `Session`.

Lorsqu'il y a un appel à la méthode `flush()`, l'état de cet objet va être synchronisé avec la base de données. Si vous ne voulez pas que cette synchronisation ait lieu ou si vous traitez un grand nombre d'objets et que vous avez besoin de gérer la mémoire de manière efficace, vous pouvez utiliser la méthode `evict()` pour supprimer l'objet et ses collections dépendantes du cache de la session

```
ScrollableResult cats = sess.createQuery("from Cat as cat").scroll(); //a huge result set
while ( cats.next() ) {
    Cat cat = (Cat) cats.get(0);
    doSomethingWithACat(cat);
    sess.evict(cat);
}
```

La `Session` dispose aussi de la méthode `contains()` pour déterminer si une instance appartient au cache de la session.

Pour retirer tous les objets du cache session, appelez `Session.clear()`

Pour le cache de second niveau, il existe des méthodes définies dans `SessionFactory` pour retirer des instances du cache, la classe entière, une instance de collection ou le rôle entier d'une collection.

```
sessionFactory.evict(Cat.class, catId); //evict a particular Cat
sessionFactory.evict(Cat.class);  //evict all Cats
sessionFactory.evictCollection("Cat.kittens", catId); //evict a particular collection of kittens
sessionFactory.evictCollection("Cat.kittens"); //evict all kitten collections
```

Le `CacheMode` contrôle comme une session particulière interragit avec le cache de second niveau

- `CacheMode.NORMAL` - lit et écrit les items dans le cache de second niveau
- `CacheMode.GET` - lit les items dans le cache de second niveau mais ne les écrit pas sauf dans le cache d'une mise à jour d'une donnée
- `CacheMode.PUT` - écrit les items dans le cache de second niveau mais ne les lit pas dans le cache de second

niveau

- `CacheMode.REFRESH` - écrit les items dans le cache de second niveau mais ne les lit pas dans le cache de second niveau, outrepasse l'effet de`hibernate.cache.use_minimal_puts`, en forçant un rafraîchissement du cache de second niveau pour chaque item lu dans la base

Pour parcourir le contenu du cache de second niveau ou la région du cache dédiée au requêtes, vous pouvez utiliser l'API `Statistics` API:

```
Map cacheEntries = sessionFactory.getStatistics()
        .getSecondLevelCacheStatistics(regionName)
        .getEntries();
```

Vous devez pour cela activer les statistiques et optionnellement forcer Hibernate à conserver les entrées dans le cache sous un format plus compréhensible pour l'utilisateur :

```
hibernate.generate_statistics true
hibernate.cache.use_structured_entries true
```

19.4. Le cache de requêtes

Les résultats d'une requête peuvent aussi être placés en cache. Ceci n'est utile que pour les requêtes qui sont exécutées avec les mêmes paramètres. Pour utiliser le cache de requêtes, vous devez d'abord l'activer :

```
hibernate.cache.use_query_cache true
```

Ce paramètre amène la création de deux nouvelles régions dans le cache, une qui va conserver le résultat des requêtes mises en cache (`org.hibernate.cache.StandardQueryCache`) et l'autre qui va conserver l'horodatage des mises à jour les plus récentes effectuées sur les tables requêtables (`org.hibernate.cache.UpdateTimestampsCache`). Il faut noter que le cache de requête ne conserve pas l'état des entités, il met en cache uniquement les valeurs de l'identifiant et les valeurs de types de base (?). Le cache de requête doit toujours être utilisé avec le cache de second niveau pour être efficace.

La plupart des requêtes ne retirent pas de bénéfice pas du cache, donc par défaut les requêtes ne sont pas mises en cache. Pour activer le cache, appelez `Query.setCacheable(true)`. Cet appel permet de vérifier si les résultats sont en cache ou non, voire d'ajouter ces résultats si la requête est exécutée.

Si vous avez besoin de contrôler finement les délais d'expiration du cache, vous pouvez spécifier une région de cache nommée pour une requête particulière en appelant `Query.setCacheRegion()`.

```
List blogs = sess.createQuery("from Blog blog where blog.blogger = :blogger")
    .setEntity("blogger", blogger)
    .setMaxResults(15)
    .setCacheable(true)
    .setCacheRegion("frontpages")
    .list();
```

Si une requête doit forcer le rafraîchissement de sa région de cache, vous devez appeler `Query.setCacheMode(CacheMode.REFRESH)`. C'est particulièrement utile lorsque les données peuvent avoir été mises à jour par un processus séparé (e.g. elles n'ont pas été modifiées par Hibernate). Cela permet à l'application de rafraîchir de manière sélective les résultats d'une requête particulière. Il s'agit d'une alternative plus efficace à l'éviction d'une région du cache à l'aide de la méthode `SessionFactory.evictQueries()`.

19.5. Comprendre les performances des Collections

Nous avons déjà passé du temps à discuter des collections. Dans cette section, nous allons traiter du comportement des collections à l'exécution.

19.5.1. Classification

Hibernate définit trois types de collections :

- les collections de valeurs
- les associations un-vers-plusieurs
- les associations plusieurs-vers-plusieurs

Cette classification distingue les différentes relations entre les tables et les clés étrangères mais ne nous apprend rien de ce que nous devons savoir sur le modèle relationnel. Pour comprendre parfaitement la structure relationnelle et les caractéristiques des performances, nous devons considérer la structure de la clé primaire qui est utilisée par Hibernate pour mettre à jour ou supprimer les éléments des collections. Celà nous amène aux classifications suivantes :

- collections indexées
- sets
- bags

Toutes les collections indexées (maps, lists, arrays) ont une clé primaire constituée des colonnes clé (`<key>`) et `<index>`. Avec ce type de clé primaire, la mise à jour de collection est en général très performante - la clé primaire peut être indexées efficacement et un élément particulier peut être localisé efficacement lorsqu'Hibernate essaie de le mettre à jour ou de le supprimer.

Les Sets ont une clé primaire composée de `<key>` et des colonnes représentant l'élément. Elle est donc moins efficace pour certains types de collections d'éléments, en particulier les éléments composites, les textes volumineux ou les champs binaires ; la base de données peut ne pas être capable d'indexer aussi efficacement une clé primaire aussi complexe. Cependant, pour les associations un-vers-plusieurs ou plusieurs-vers-plusieurs, spécialement lorsque l'on utilise des entités ayant des identifiants techniques, il est probable que cela soit aussi efficace (note : si vous voulez que `SchemaExport` créé effectivement la clé primaire d'un `<set>` pour vous, vous devez déclarer toutes les colonnes avec `not-null="true"`).

Le mapping à l'aide d'`<idbag>` définit une clé de substitution ce qui leur permet d'être très efficaces lors de la mise à jour. En fait il s'agit du meilleur cas de mise à jour d'une collection

Le pire cas intervient pour les Bags. Dans la mesure où un bag permet la duplications des éléments et n'a pas de colonne d'index, aucune clé primaire ne peut être définie. Hibernate n'a aucun moyen de distinguer des enregistrements dupliqués. Hibernate résout ce problème en supprimant complètement les enregistrements (via un simple `DELETE`), puis en recréant la collection chaque fois qu'elle change. Ce qui peut être très inefficace.

Notez que pour une relation un-vers-plusieurs, la "clé primaire" peut ne pas être la clé primaire de la table en base de données - mais même dans ce cas, la classification ci-dessus reste utile (Elle explique comment Hibernate "localise" chaque enregistrement de la collection).

19.5.2. Les lists, les maps, les idbags et les sets sont les collections les plus efficaces pour la mise à jour

La discussion précédente montre clairement que les collections indexées et (la plupart du temps) les sets, permettent de réaliser le plus efficacement les opérations d'ajout, de suppression ou de modification d'éléments.

Il existe un autre avantage qu'ont les collections indexées sur les Sets dans le cadre d'une association plusieurs vers plusieurs ou d'une collection de valeurs. A cause de la structure inhérente d'un `Set`, Hibernate n'effectue jamais d'`UPDATE` quand un enregistrement est modifié. Les modifications apportées à un `Set` se font via un `INSERT` et `DELETE` (de chaque enregistrement). Une fois de plus, ce cas ne s'applique pas aux associations un vers plusieurs.

Après s'être rappelé que les tableaux ne peuvent pas être chargés tardivement, nous pouvons conclure que les lists, les maps et les idbags sont les types de collections (non inversées) les plus performants, avec les sets pas loin derrières. Les sets son le type de collection le plus courant dans les applications Hibernate. Cela est du au fait que la sémantique des "set" est la plus naturelle dans le modèle relationnel.

Cependant, dans des modèles objet bien conçus avec Hibernate, on voit souvent que la plupart des collections sont en fait des associations "un-vers-plusieurs" avec `inverse="true"`. Pour ces associations, les mises à jour sont gérées au niveau de l'association "plusieurs-vers-un" et les considérations de performance de mise à jour des collections ne s'appliquent tout simplement pas dans ces cas là.

19.5.3. Les Bags et les lists sont les plus efficaces pour les collections inverse

Avant que vous n'oubliez les bags pour toujours, il y a un cas précis où les bags (et les lists) sont bien plus performants que les sets. Pour une collection marquée comme `inverse="true"` (le choix le plus courant pour un relation un vers plusieurs bidirectionnelle), nous pouvons ajouter des éléments à un bag ou une list sans avoir besoin de l'initialiser (fetch) les éléments du sac! Ceci parce que `Collection.add()` ou `Collection.addAll()` doit toujours retourner vrai pour un bag ou une `List` (contrairement au `Set`). Cela peut rendre le code suivant beaucoup plus rapide.

```
Parent p = (Parent) sess.load(Parent.class, id);
    Child c = new Child();
    c.setParent(p);
    p.getChildren().add(c);  //pas besoin de charger la collection !
    sess.flush();
```

19.5.4. Suppression en un coup

Parfois, effacer les éléments d'une collection un par un peut être extrêmement inefficace. Hibernate n'est pas totalement stupide, il sait qu'il ne faut pas le faire dans le cas d'une collection complètement vidée (lorsque vous appellez `list.clear()`, par exemple). Dans ce cas, Hibernate fera un simple `DELETE` et le travail est fait !

Supposons que nous ajoutions un élément dans une collection de taille vingt et que nous enlevions ensuite deux éléments. Hibernate effectuera un `INSERT` puis deux `DELETE` (à moins que la collection ne soit un bag). Ce qui est souhaitable.

Cependant, supposons que nous enlevions dix huit éléments, laissant ainsi deux éléments, puis que nous ajoutions trois nouveaux éléments. Il y a deux moyens de procéder.

- effacer dix huit enregistrements un à un puis en insérer trois

- effacer la totalité de la collection (en un `DELETE` SQL) puis insérer les cinq éléments restant un à un

Hibernate n'est pas assez intelligent pour savoir que, dans ce cas, la seconde méthode est plus rapide (Il plutôt heureux qu'Hibernate ne soit pas trop intelligent ; un tel comportement pourrait rendre l'utilisation de triggers de bases de données plutôt aléatoire, etc...).

Heureusement, vous pouvez forcer ce comportement lorsque vous le souhaitez, en liberant (c'est-à-dire en déréférençant) la collection initiale et en retournant une collection nouvellement instanciée avec les éléments restants. Ceci peut être très pratique et très puissant de temps en temps.

Bien sûr, la suppression en un coup ne s'applique pas pour les collections qui sont mappées avec `inverse="true"`.

19.6. Moniteur de performance

L'optimisation n'est pas d'un grand intérêt sans le suivi et l'accès aux données de performance. Hibernate fournit toute une panoplie de rapport sur ses opérations internes. Les statistiques dans Hibernate sont fournies par `SessionFactory`.

19.6.1. Suivi d'une SessionFactory

Vous pouvez accéder au métriques d'une `SessionFactory` de deux manières. La première option est d'appeler `sessionFactory.getStatistics()` et de lire ou d'afficher les `Statistics` vous même.

Hibernate peut également utiliser JMX pour publier les métriques si vous activez le MBean `StatisticsService`. Vous pouvez activer un seul MBean pour toutes vos `SessionFactory` ou un par factory. Voici un code qui montre un exemple de configuration minimaliste :

```
// MBean service registration for a specific SessionFactory
Hashtable tb = new Hashtable();
tb.put("type", "statistics");
tb.put("sessionFactory", "myFinancialApp");
ObjectName on = new ObjectName("hibernate", tb); // MBean object name

StatisticsService stats = new StatisticsService(); // MBean implementation
stats.setSessionFactory(sessionFactory); // Bind the stats to a SessionFactory
server.registerMBean(stats, on); // Register the Mbean on the server
```

```
// MBean service registration for all SessionFactory's
Hashtable tb = new Hashtable();
tb.put("type", "statistics");
tb.put("sessionFactory", "all");
ObjectName on = new ObjectName("hibernate", tb); // MBean object name

StatisticsService stats = new StatisticsService(); // MBean implementation
server.registerMBean(stats, on); // Register the MBean on the server
```

TODO: Cela n'a pas de sens : dans le premier cs on récupère et on utilise le MBean directement. Dans le second, on doit fournir le nom JNDI sous lequel est retenu la fabrique de session avant de l'utiliser. Pour cela il faut utiliser `hibernateStatsBean.setSessionFactoryJNDIName("my/JNDI/Name")`

Vous pouvez (dés)activer le suivi pour une `SessionFactory`

- au moment de la configuration en mettant `hibernate.generate_statistics` à `false`

- à chaud avec `sf.getStatistics().setStatisticsEnabled(true)` ou `hibernateStatsBean.setStatisticsEnabled(true)`

Les statistiques peuvent être remises à zéro de manière programmatique à l'aide de la méthode `clear()` Un résumé peut être envoyé à un logger (niveau info) à l'aide de la méthode `logSummary()`

19.6.2. Métriques

Hibernate fournit un certain nombre de métriques, qui vont des informations très basiques aux informations très spécialisées qui ne sont appropriées que dans certains scenarii. Tous les compteurs accessibles sont décrits dans l'API de l'interface `Statistics` dans trois catégories :

- Les métriques relatives à l'usage général de la `Session` comme le nombre de sessions ouvertes, le nombre de connexions JDBC récupérées, etc...
- Les métriques relatives aux entités, collections, requêtes et caches dans leur ensemble (métriques globales),
- Les métriques détaillées relatives à une entité, une collection, une requête ou une région de cache particulière.

Par exemple, vous pouvez vérifier l'accès au cache ainsi que le taux d'éléments manquants et de mise à jour des entités, collections et requêtes et le temps moyen que met une requête. Il faut faire attention au fait que le nombre de millisecondes est sujet à approximation en Java. Hibernate est lié à la précision de la machine virtuelle, sur certaines plateformes, cela n'offre qu'une précision de l'ordre de 10 secondes.

Des accesseurs simples sont utilisés pour accéder aux métriques globales (e.g. celles qui ne sont pas liées à une entité, collection ou région de cache particulière). Vous pouvez accéder aux métriques d'une entité, collection, région de cache particulière à l'aide de son nom et à l'aide de sa représentation HQL ou SQL pour une requête. Référez vous à la javadoc des APIS `Statistics`, `EntityStatistics`, `CollectionStatistics`, `SecondLevelCacheStatistics`, and `QueryStatistics` pour plus d'informations. Le code ci-dessous montre un exemple simple :

```
Statistics stats = HibernateUtil.sessionFactory.getStatistics();

double queryCacheHitCount  = stats.getQueryCacheHitCount();
double queryCacheMissCount = stats.getQueryCacheMissCount();
double queryCacheHitRatio =
  queryCacheHitCount / (queryCacheHitCount + queryCacheMissCount);

log.info("Query Hit ratio:" + queryCacheHitRatio);

EntityStatistics entityStats =
  stats.getEntityStatistics( Cat.class.getName() );
long changes =
        entityStats.getInsertCount()
        + entityStats.getUpdateCount()
        + entityStats.getDeleteCount();
log.info(Cat.class.getName() + " changed " + changes + "times"  );
```

Pour travailler sur toutes les entités, collections, requêtes et régions de cache, vous pouvez récupérer la liste des noms des entités, collections, requêtes et régions de cache avec les méthodes : `getQueries()`, `getEntityNames()`, `getCollectionRoleNames()`, et `getSecondLevelCacheRegionNames()`.

Chapitre 20. Guide des outils

Des outils en ligne de commande, des plugins Eclipse ainsu que des tâches Ant permettent de gérer de cycles de développement complet de projets utilisant Hibernate.

Les *outils Hibernate* actuels incluent des plugins pour l'IDE Eclipse ainsi que des tâches Ant pour l'ingénierie inverse de bases de données existantes :

- *Mapping Editor :* un éditeur pour les fichiers de mapping XML Hibernate, supportant l'auto-complétion et la mise en valeur de la syntaxe. Il supporte aussi l'auto-complétion automatique pour les noms de classes et les noms de propriété/champ, le rendant beaucoup plus polyvalent qu'un éditeurXML normal.
- *Console :* la console est une nouvelle vue d'Eclipse. En plus de la vue d'ensemble arborescente de vos configurations de console, vous obtenez aussi une vue interactive de vos classes persistantes et de leurs relations. La console vous permet d'exécuter des requête HQL dans votre base de données et de parcourir les résultats directement dans Eclipse.
- *Development Wizards :* plusieurs assistants sont fournis avec les outils d'Hibernate pour Eclipse ; vous pouvez utiliser un assistant pour générer rapidement les fichiers de configuration d'Hibernate (cfg.xml), ou vous pouvez même complètement générer les fichiers de mapping Hibernate et les sources des POJOs à partir d'un schéma de base de données existant. L'assistant d'ingénierie inverse supporte les modèles utilisateur.
- *Tâches Ant :*

Veuillez-vous référer au paquet *outils Hibernate* et sa documentation pour plus d'informations.

Pourtant, le paquet principal d'Hibernate arrive avec un lot d'outils intégrés (il peut même être utilisé de "l'intérieur" d'Hibernate à la volée) : *SchemaExport* aussi connu comme `hbm2ddl`.

20.1. Génération automatique du schéma

La DDL peut être générée à partir de vos fichiers de mapping par un utilitaire d'Hibernate. Le schéma généré inclut les contraintes d'intégrité référentielle (clefs primaires et étrangères) pour les tables d'entités et de collections. Les tables et les séquences sont aussi créées pour les générateurs d'identifiant mappés.

Vous *devez* spécifier un `Dialect` SQL via la propriété `hibernate.dialect` lors de l'utilisation de cet outils, puisque la DDL est fortement dépendante de la base de données.

D'abord, personnalisez vos fichiers de mapping pour améliorer le schéma généré.

20.1.1. Personnaliser le schéma

Plusieurs éléments du mapping hibernate définissent des attributs optionnels nommés `length`, `precision` et `scale`. Vous pouvez paramétrer la longueur, la précision,... d'une colonne avec ces attributs.

```
<property name="zip" length="5"/>
```

```
<property name="balance" precision="12" scale="2"/>
```

Certains éléments acceptent aussi un attribut `not-null` (utilisé pour générer les contraintes de colonnes `NOT`

NULL) et un attribut unique (pour générer une contrainte de colonne UNIQUE).

```
<many-to-one name="bar" column="barId" not-null="true"/>
```

```
<element column="serialNumber" type="long" not-null="true" unique="true"/>
```

Un attribut unique-key peut être utilisé pour grouper les colonnes en une seule contrainte d'unicité. Actuellement, la valeur spécifiée par l'attribut unique-key n'est *pas* utilisée pour nommer la contrainte dans le DDL généré, elle sert juste à grouper les colonnes dans le fichier de mapping.

```
<many-to-one name="org" column="orgId" unique-key="OrgEmployeeId"/>
            <property name="employeeId" unique-key="OrgEmployee"/>
```

Un attribut index indique le nom d'un index qui sera créé en utilisant la ou les colonnes mappées. Plusieurs colonnes peuvent être groupées dans un même index, en spécifiant le même nom d'index.

```
<property name="lastName" index="CustName"/>
<property name="firstName" index="CustName"/>
```

Un attribut foreign-key peut être utilisé pour surcharger le nom des clés étrangères générées.

```
<many-to-one name="bar" column="barId" foreign-key="FKFooBar"/>
```

Plusieurs éléments de mapping acceptent aussi un élément fils <column>. Ceci est utile pour les type multi-colonnes:

```
<property name="name" type="my.customtypes.Name"/>
    <column name="last" not-null="true" index="bar_idx" length="30"/>
    <column name="first" not-null="true" index="bar_idx" length="20"/>
    <column name="initial"/>
</property>
```

L'attribut default vous laisse spécifier une valeur par défaut pour une colonnes (vous devriez assigner la même valeur à la propriété mappée avant de sauvegarder une nouvelle instance de la classe mappée).

```
<property name="credits" type="integer" insert="false">
    <column name="credits" default="10"/>
</property>
```

```
<version name="version" type="integer" insert="false">
    <column name="version" default="0"/>
</property>
```

L'attribut sql-type laisse l'utilisateur surcharger le mapping par défaut du type Hibernate vers un type SQL.

```
<property name="balance" type="float">
    <column name="balance" sql-type="decimal(13,3)"/>
</property>
```

L'attribut check permet de spécifier une contrainte de vérification.

```
<property name="foo" type="integer">
    <column name="foo" check="foo > 10"/>
</property>
```

```
<class name="Foo" table="foos" check="bar < 100.0">
    ...
    <property name="bar" type="float"/>
</class>
```

Tableau 20.1. Summary

Attribut	Valeur	Interprétation
`length`	numérique	taille d'une colonne
`precision`	numérique	précision décimale de la colonne
`scale`	numérique	scale décimale de la colonne
`not-null`	`true|false`	spécifie que la colonne doit être non-nulle
`unique`	`true|false`	spécifie que la colonne doit avoir une contrainte d'unicité
`index`	`index_name`	spécifie le nom d'un index (multi-colonnes)
`unique-key`	`unique_key_name`	spécifie le nom d'une contrainte d'unicité multi-colonnes
`foreign-key`	`foreign_key_name`	spécifie le nom d'une contrainte de clé étrangère générée pour une association, utilisez-la avec les éléments de mapping <one-to-one>, <many-to-one>, <key>, et <many-to-many> Notez que les extrêmités `inverse="true"` se seront pas prises en compte par `SchemaExport`.
`sql-type`	`SQL column_type`	surcharge le type par défaut (attribut de l'élément `<column>` uniquement)
`default`	expression SQL	spécifie une valeur par défaut pour la colonne
`check`	SQL expression	crée une contrainte de vérification sur la table ou la colonne

L'élément `<comment>` vous permet de spécifier un commentaire pour le schéma généré.

```
<class name="Customer" table="CurCust">
    <comment>Current customers only</comment>
    ...
</class>
```

```
<property name="balance">
    <column name="bal">
        <comment>Balance in USD</comment>
    </column>
</property>
```

Ceci a pour résultat une expression `comment on table` ou `comment on column` dans la DDL générée (où supportée).

20.1.2. Exécuter l'outil

L'outil `SchemaExport` génère un script DDL vers la sortie standard et/ou exécute les ordres DDL.

`java -cp` *classpath_hibernate* `net.sf.hibernate.tool.hbm2ddl.SchemaExport` *options fichiers_de_mapping*

Tableau 20.2. `SchemaExport` Options de la ligne de commande

Option	Description
`--quiet`	ne pas écrire le script vers la sortie standard
`--drop`	supprime seuleument les tables
`--create`	ne créé que les tables
`--text`	ne pas exécuter sur la base de données
`--output=my_schema.ddl`	écrit le script ddl vers un fichier
`--naming=eg.MyNamingStrategy`	sélectionne une `NamingStrategy`
`--config=hibernate.cfg.xml`	lit la configuration Hibernate à partir d'un fichier XML
`--properties=hibernate.properties`	lit les propriétés de la base de données à partir d'un fichier
`--format`	formatte proprement le SQL généré dans le script
`--delimiter=x`	paramètre un délimiteur de fin de ligne pour le script

Vous pouvez même intégrer `SchemaExport` dans votre application :

```
Configuration cfg = ....;
new SchemaExport(cfg).create(false, true);
```

20.1.3. Propriétés

Les propriétés de la base de données peuvent être spécifiées

- comme propriétés système avec `-D`<*property*>
- dans `hibernate.properties`
- dans un fichier de propriétés déclaré avec `--properties`

Les propriétés nécessaires sont :

Tableau 20.3. SchemaExport Connection Properties

Nom de la propriété	Description
`hibernate.connection.driver_class`	classe du driver JDBC
`hibernate.connection.url`	URL JDBC
`hibernate.connection.username`	utilisateur de la base de données
`hibernate.connection.password`	mot de passe de l'utilisateur
`hibernate.dialect`	dialecte

20.1.4. Utiliser Ant

Vous pouvez appeler `SchemaExport` depuis votre script de construction Ant :

```
<target name="schemaexport">
    <taskdef name="schemaexport"
```

```
        classname="org.hibernate.tool.hbm2ddl.SchemaExportTask"
        classpathref="class.path"/>

    <schemaexport
        properties="hibernate.properties"
        quiet="no"
        text="no"
        drop="no"
        delimiter=";"
        output="schema-export.sql">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaexport>
</target>
```

20.1.5. Mises à jour incrémentales du schéma

L'outil `SchemaUpdate` mettra à jour un schéma existant en effectuant les changement par "incrément". Notez que `SchemaUpdate` dépends beaucoup de l'API JDBC metadata, il ne fonctionnera donc pas avec tous les drivers JDBC.

`java -cp` *classpath_hibernate* `net.sf.hibernate.tool.hbm2ddl.SchemaUpdate` *options fichiers_de_mapping*

Tableau 20.4. `SchemaUpdate` Options de ligne de commande

Option	Description
`--quiet`	ne pas écrire vers la sortie standard
`--text`	ne pas exporter vers la base de données
`--naming=eg.MyNamingStrategy`	choisit une `NamingStrategy`
`--properties=hibernate.properties`	lire les propriétés de la base de données à partir d'un fichier

Vous pouvez intégrer `SchemaUpdate` dans votre application :

```
Configuration cfg = ....;
new SchemaUpdate(cfg).execute(false);
```

20.1.6. Utiliser Ant pour des mises à jour de schéma par incrément

Vous pouvez appeler `SchemaUpdate` depuis le script Ant :

```
<target name="schemaupdate">
    <taskdef name="schemaupdate"
        classname="org.hibernate.tool.hbm2ddl.SchemaUpdateTask"
        classpathref="class.path"/>

    <schemaupdate
        properties="hibernate.properties"
        quiet="no">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaupdate>
</target>
```

20.1.6.1. Validation du schéma

L'outil `SchemaValidator` validera que le schéma existant correspond à vos documents de mapping. Notez que le `SchemaValidator` dépends de l'API metadata de JDBC, il ne fonctionnera donc pas avec tous les drivers JDBC. Cet outil est extrêmement utile pour tester.

`java -cp` *hibernate_classpaths* `org.hibernate.tool.hbm2ddl.SchemaValidator` *options mapping_files*

Tableau 20.5. `SchemaValidator` Options de ligne de commande

Option	Description
`--naming=eg.MyNamingStrategy`	Indique une `NamingStrategy`
`--properties=hibernate.properties`	lit les propriétés dela base de données depuis un fichier de propriétés
`--config=hibernate.cfg.xml`	indique un fichier `.cfg.xml`

Vous pouvez inclure `SchemaValidator` dans votre application:

```
Configuration cfg = ....;
new SchemaValidator(cfg).validate();
```

20.1.7. Utiliser Ant pour la validation du Schéma

Vous pouvez appeler `SchemaValidator` depuis le script Ant:

```
<target name="schemavalidate">
    <taskdef name="schemavalidator"
        classname="org.hibernate.tool.hbm2ddl.SchemaValidatorTask"
        classpathref="class.path"/>

    <schemavalidator
        properties="hibernate.properties">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
    </schemaupdate>
</target>
```

Chapitre 21. Exemple : Père/Fils

L'une des premières choses que les nouveaux utilisateurs essaient de faire avec Hibernate est de modéliser une relation père/fils. Il y a deux approches différentes pour cela. Pour un certain nombre de raisons, la méthode la plus courante, en particulier pour les nouveaux utilisateurs, est de modéliser les deux relations `Père` et `Fils` comme des classes entités liées par une association `<one-to-many>` du `Père` vers le `Fils` (l'autre approche est de déclarer le `Fils` comme un `<composite-element>`). Il est évident que le sens de l'association un vers plusieurs (dans Hibernate) est bien moins proche du sens habituel d'une relation père/fils que ne l'est celui d'un élément cmposite. Nous allons vous expliquer comment utiliser une association *un vers plusieurs bidirectionnelle avec cascade* afin de modéliser efficacement et élégamment une relation père/fils, ce n'est vraiment pas difficile !

21.1. Une note à propos des collections

Les collections Hibernate sont considérées comme étant une partie logique de l'entité dans laquelle elles sont contenues ; jamais des entités qu'elle contient. C'est une distinction crutiale ! Les conséquences sont les suivantes :

- Quand nous ajoutons / retirons un objet d'une collection, le numéro de version du propriétaire de la collection est incrémenté.
- Si un objet qui a été enlevé d'une collection est une instance de type valeur (ex : élément composite), cet objet cessera d'être persistant et son état sera complètement effacé de la base de données. Par ailleurs, ajouter une instance de type valeur dans une collection aura pour conséquence que son état sera immédiatement persistant.
- Si une entité est enlevée d'une collection (association un-vers-plusieurs ou plusieurs-vers-plusieurs), par défaut, elle ne sera pas effacée. Ce comportement est complètement logique - une modification de l'un des états internes d'une entité ne doit pas causer la disparition de l'entité associée ! De même, l'ajout d'une entité dans une collection n'engendre pas, par défaut, la persistance de cette entité.

Le comportement par défaut est donc que l'ajout d'une entité dans une collection créé simplement le lien entre les deux entités, et qu'effacer une entité supprime ce lien. C'est le comportement le plus approprié dans la plupart des cas. Ce comportement n'est cependant pas approprié lorsque la vie du fils est liée au cycle de vie du père.

21.2. un-vers-plusieurs bidirectionnel

Supposons que nous ayons une simple association `<one-to-many>` de `Parent` vers `Child`.

```
<set name="children">
                <key column="parent_id"/>
                <one-to-many class="Child"/>
            </set>
```

Si nous executions le code suivant

```
Parent p = .....;
Child c = new Child();
p.getChildren().add(c);
session.save(c);
session.flush();
```

Hibernate exécuterait deux ordres SQL:

- un `INSERT` pour créer l'enregistrement pour `c`
- un `UPDATE` pour créer le lien de `p` vers `c`

Ceci est non seuleument inefficace, mais viole aussi toute contrainte `NOT NULL` sur la colonne `parent_id`. Nous pouvons réparer la contrainte de nullité en spécifiant `not-null="true"` dans le mapping de la collection :

```
<set name="children">
    <key column="parent_id" not-null="true"/>
    <one-to-many class="Child"/>
</set>
```

Cependant ce n'est pas la solution recommandée.

La cause sous jacente à ce comportement est que le lien (la clé étrangère `parent_id`) de `p` vers `c` n'est pas considérée comme faisant partie de l'état de l'objet `Child` et n'est donc pas créé par l'`INSERT`. La solution est donc que ce lien fasse partie du mapping de `Child`.

```
<many-to-one name="parent" column="parent_id" not-null="true"/>
```

(Nous avons aussi besoin d'ajouter la propriété `parent` dans la classe `Child`).

Maintenant que l'état du lien est géré par l'entité `Child`, nous spécifions à la collection de ne pas mettre à jour le lien. Nous utilisons l'attribut `inverse`.

```
<set name="children" inverse="true">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

Le code suivant serait utilisé pour ajouter un nouveau `Child`

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
c.setParent(p);
p.getChildren().add(c);
session.save(c);
session.flush();
```

Maintenant, seul un `INSERT` SQL est nécessaire !

Pour alléger encore un peu les choses, nous devrions créer une méthode `addChild()` dans `Parent`.

```
public void addChild(Child c) {
    c.setParent(this);
    children.add(c);
}
```

Le code d'ajout d'un `Child` serait alors

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
session.save(c);
session.flush();
```

21.3. Cycle de vie en cascade

L'appel explicite de `save()` est un peu fastidieux. Nous pouvons simplifier cela en utilisant les cascades.

```
<set name="children" inverse="true" cascade="all">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

Simplifie le code précédent en

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
session.flush();
```

De la même manière, nous n'avons pas à itérer sur les fils lorsque nous sauvons ou effacons un `Parent`. Le code suivant efface `p` et tous ses fils de la base de données.

```
Parent p = (Parent) session.load(Parent.class, pid);
session.delete(p);
session.flush();
```

Par contre, ce code

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
c.setParent(null);
session.flush();
```

n'effacera pas `c` de la base de données, il enlèvera seulement le lien vers `p` (et causera une violation de contrainte `NOT NULL`, dans ce cas). Vous devez explicitement utiliser `delete()` sur `Child`.

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
session.delete(c);
session.flush();
```

Dans notre cas, un `Child` ne peut pas vraiment exister sans son père. Si nous effacons un `Child` de la collection, nous voulons vraiment qu'il soit effacé. Pour cela, nous devons utiliser `cascade="all-delete-orphan"`.

```
<set name="children" inverse="true" cascade="all-delete-orphan">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

A noter : même si le mapping de la collection spécifie `inverse="true"`, les cascades sont toujours assurées par l'itération sur les éléments de la collection. Donc, si vous avez besoin qu'un objet soit enregistré, effacé ou mis à jour par cascade, vous devez l'ajouter dans la colleciton. Il ne suffit pas d'appeler explicitement `setParent()`.

21.4. Cascades et `unsaved-value`

Supposons que nous ayons chargé un `Parent` dans une `Session`, que nous l'ayons ensuite modifié et que voulions persiter ces modifications dans une nouvelle session en appelant `update()`. Le `Parent` contiendra une

collection de fils et, puisque la cascade est activée, Hibernate a besoin de savoir quels fils viennent d'être instanciés et quels fils proviennent de la base de données. Supposons aussi que `Parent` et `Child` ont tous deux des identifiants du type `Long`. Hibernate utilisera la propriété de l'identifiant et la propriété de la version/horodatage pour déterminer quels fils sont nouveaux (vous pouvez aussi utiliser la propriété version ou timestamp, voir ???). *Dans Hibernate3, il n'est plus nécessaire de spécifier une `unsaved-value` explicitement.*

Le code suivant mettra à jour `parent` et `child` et insérera `newChild`.

```
//parent et child ont été chargés dans une session précédente
parent.addChild(child);
Child newChild = new Child();
parent.addChild(newChild);
session.update(parent);
session.flush();
```

Ceci est très bien pour des identifiants générés, mais qu'en est-il des identifiants assignés et des identifiants composés ? C'est plus difficile, puisqu'Hibernate ne peut pas utiliser la propriété de l'identifiant pour distinguer un objet nouvellement instancié (avec un identifiant assigné par l'utilisateur) d'un objet chargé dans une session précédente. Dans ce cas, Hibernate utilisera soit la propriété de version ou d'horodatage, soit effectuera vraiment une requête au cache de second niveau, soit, dans le pire des cas, à la base de données, pour voir si la ligne existe.

21.5. Conclusion

Il y a quelques principes à maîtriser dans ce chapitre et tout cela peut paraître déroutant la première fois. Cependant, dans la pratique, tout fonctionne parfaitement. La plupart des applications Hibernate utilisent le pattern père / fils.

Nous avons évoqué une alternative dans le premier paragraphe. Aucun des points traités précédemment n'existe dans le cas d'un mapping `<composite-element>` qui possède exactement la sémantique d'une relation père / fils. Malheureusement, il y a deux grandes limitations pour les classes éléments composites : les éléments composites ne peuvent contenir de collections, et ils ne peuvent être les fils d'entités autres que l'unique parent.

Chapitre 22. Exemple : application Weblog

22.1. Classes persistantes

Les classes persistantes representent un weblog, et un article posté dans un weblog. Il seront modélisés comme une relation père/fils standard, mais nous allons utiliser un "bag" trié au lieu d'un set.

```
package eg;

import java.util.List;

public class Blog {
    private Long _id;
    private String _name;
    private List _items;

    public Long getId() {
        return _id;
    }
    public List getItems() {
        return _items;
    }
    public String getName() {
        return _name;
    }
    public void setId(Long long1) {
        _id = long1;
    }
    public void setItems(List list) {
        _items = list;
    }
    public void setName(String string) {
        _name = string;
    }
}
```

```
package eg;

import java.text.DateFormat;
import java.util.Calendar;

public class BlogItem {
    private Long _id;
    private Calendar _datetime;
    private String _text;
    private String _title;
    private Blog _blog;

    public Blog getBlog() {
        return _blog;
    }
    public Calendar getDatetime() {
        return _datetime;
    }
    public Long getId() {
        return _id;
    }
    public String getText() {
        return _text;
    }
    public String getTitle() {
        return _title;
    }
    public void setBlog(Blog blog) {
        _blog = blog;
```

```
    }
    public void setDatetime(Calendar calendar) {
        _datetime = calendar;
    }
    public void setId(Long long1) {
        _id = long1;
    }
    public void setText(String string) {
        _text = string;
    }
    public void setTitle(String string) {
        _title = string;
    }
}
```

22.2. Mappings Hibernate

Le mapping XML doit maintenant être relativement simple à vos yeux.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="eg">

    <class
        name="Blog"
        table="BLOGS">

        <id
            name="id"
            column="BLOG_ID">

            <generator class="native"/>

        </id>

        <property
            name="name"
            column="NAME"
            not-null="true"
            unique="true"/>

        <bag
            name="items"
            inverse="true"
            order-by="DATE_TIME"
            cascade="all">

            <key column="BLOG_ID"/>
            <one-to-many class="BlogItem"/>

        </bag>

    </class>

</hibernate-mapping>
```

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="eg">
```

```
    <class
        name="BlogItem"
        table="BLOG_ITEMS"
        dynamic-update="true">

        <id
            name="id"
            column="BLOG_ITEM_ID">

            <generator class="native"/>

        </id>

        <property
            name="title"
            column="TITLE"
            not-null="true"/>

        <property
            name="text"
            column="TEXT"
            not-null="true"/>

        <property
            name="datetime"
            column="DATE_TIME"
            not-null="true"/>

        <many-to-one
            name="blog"
            column="BLOG_ID"
            not-null="true"/>

    </class>

</hibernate-mapping>
```

22.3. Code Hibernate

La classe suivante montre quelques utilisations que nous pouvons faire de ces classes.

```
package eg;

import java.util.ArrayList;
import java.util.Calendar;
import java.util.Iterator;
import java.util.List;

import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import org.hibernate.tool.hbm2ddl.SchemaExport;

public class BlogMain {

    private SessionFactory _sessions;

    public void configure() throws HibernateException {
        _sessions = new Configuration()
            .addClass(Blog.class)
            .addClass(BlogItem.class)
            .buildSessionFactory();
    }
```

```
    public void exportTables() throws HibernateException {
        Configuration cfg = new Configuration()
            .addClass(Blog.class)
            .addClass(BlogItem.class);
        new SchemaExport(cfg).create(true, true);
    }

    public Blog createBlog(String name) throws HibernateException {

        Blog blog = new Blog();
        blog.setName(name);
        blog.setItems( new ArrayList() );

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.persist(blog);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return blog;
    }

    public BlogItem createBlogItem(Blog blog, String title, String text)
                        throws HibernateException {

        BlogItem item = new BlogItem();
        item.setTitle(title);
        item.setText(text);
        item.setBlog(blog);
        item.setDatetime( Calendar.getInstance() );
        blog.getItems().add(item);

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.update(blog);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return item;
    }

    public BlogItem createBlogItem(Long blogid, String title, String text)
                        throws HibernateException {

        BlogItem item = new BlogItem();
        item.setTitle(title);
        item.setText(text);
        item.setDatetime( Calendar.getInstance() );

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            Blog blog = (Blog) session.load(Blog.class, blogid);
            item.setBlog(blog);
```

```
            blog.getItems().add(item);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return item;
    }

    public void updateBlogItem(BlogItem item, String text)
                    throws HibernateException {

        item.setText(text);

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            session.update(item);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
    }

    public void updateBlogItem(Long itemid, String text)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        try {
            tx = session.beginTransaction();
            BlogItem item = (BlogItem) session.load(BlogItem.class, itemid);
            item.setText(text);
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
    }

    public List listAllBlogNamesAndItemCounts(int max)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        List result = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "select blog.id, blog.name, count(blogItem) " +
                "from Blog as blog " +
                "left outer join blog.items as blogItem " +
                "group by blog.name, blog.id " +
                "order by max(blogItem.datetime)"
            );
            q.setMaxResults(max);
            result = q.list();
```

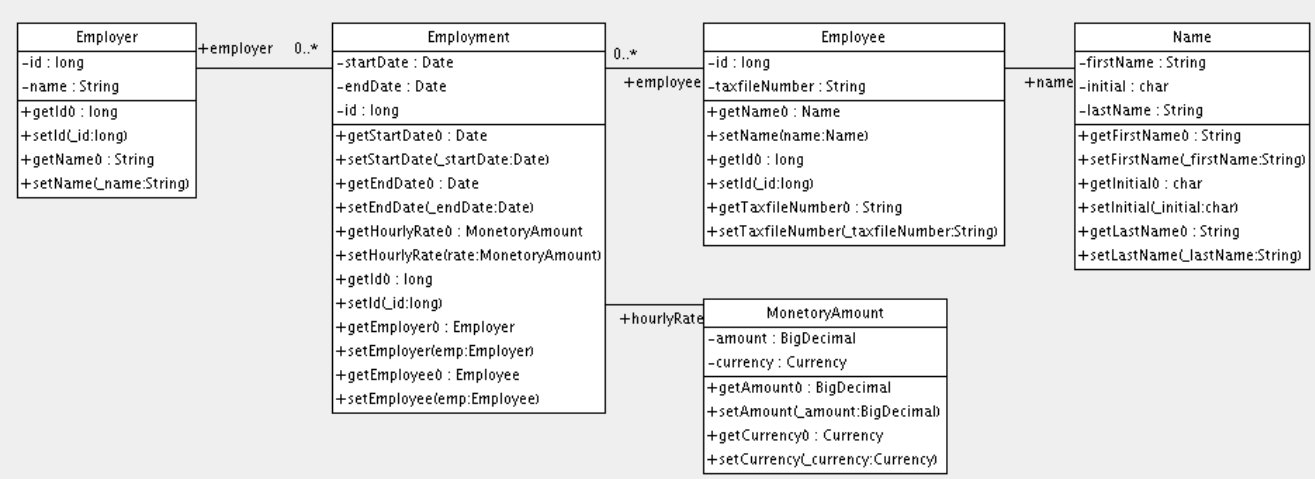
```
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return result;
    }

    public Blog getBlogAndAllItems(Long blogid)
                    throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        Blog blog = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "from Blog as blog " +
                "left outer join fetch blog.items " +
                "where blog.id = :blogid"
            );
            q.setParameter("blogid", blogid);
            blog  = (Blog) q.uniqueResult();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return blog;
    }

    public List listBlogsAndRecentItems() throws HibernateException {

        Session session = _sessions.openSession();
        Transaction tx = null;
        List result = null;
        try {
            tx = session.beginTransaction();
            Query q = session.createQuery(
                "from Blog as blog " +
                "inner join blog.items as blogItem " +
                "where blogItem.datetime > :minDate"
            );

            Calendar cal = Calendar.getInstance();
            cal.roll(Calendar.MONTH, false);
            q.setCalendar("minDate", cal);

            result = q.list();
            tx.commit();
        }
        catch (HibernateException he) {
            if (tx!=null) tx.rollback();
            throw he;
        }
        finally {
            session.close();
        }
        return result;
    }
}
```

Chapitre 23. Exemple : quelques mappings

Ce chapitre montre quelques mappings plus complexes.

23.1. Employeur/Employé (Employer/Employee)

Le modèle suivant de relation entre `Employer` et `Employee` utilise une vraie classe entité (`Employment`) pour représenter l'association. On a fait cela parce qu'il peut y avoir plus d'une période d'emploi pour les deux mêmes parties. Des composants sont utilisés pour modéliser les valeurs monétaires et les noms des employés.



Voici un document de mapping possible :

```
<hibernate-mapping>

    <class name="Employer" table="employers">
        <id name="id">
            <generator class="sequence">
                <param name="sequence">employer_id_seq</param>
            </generator>
        </id>
        <property name="name"/>
    </class>

    <class name="Employment" table="employment_periods">

        <id name="id">
            <generator class="sequence">
                <param name="sequence">employment_id_seq</param>
            </generator>
        </id>
        <property name="startDate" column="start_date"/>
        <property name="endDate" column="end_date"/>

        <component name="hourlyRate" class="MonetaryAmount">
            <property name="amount">
                <column name="hourly_rate" sql-type="NUMERIC(12, 2)"/>
            </property>
            <property name="currency" length="12"/>
        </component>

        <many-to-one name="employer" column="employer_id" not-null="true"/>
        <many-to-one name="employee" column="employee_id" not-null="true"/>

    </class>

    <class name="Employee" table="employees">
        <id name="id">
```

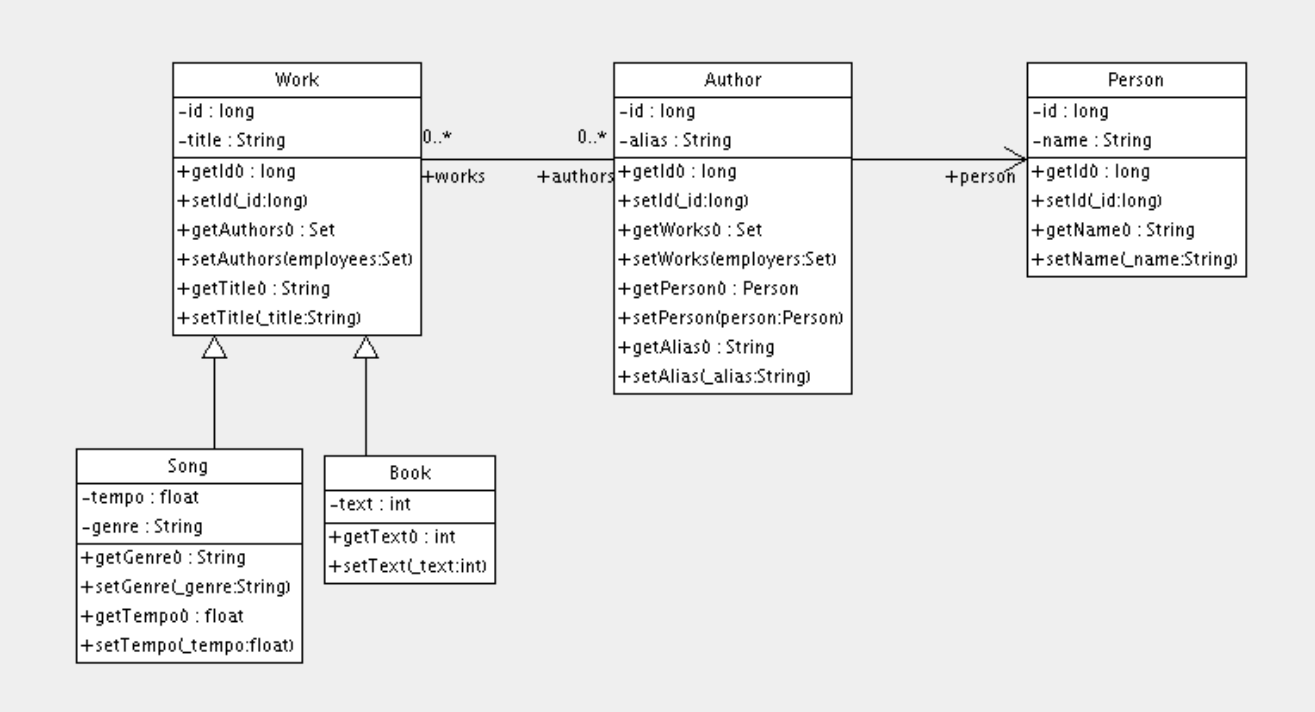
```
            <generator class="sequence">
                <param name="sequence">employee_id_seq</param>
            </generator>
        </id>
        <property name="taxfileNumber"/>
        <component name="name" class="Name">
            <property name="firstName"/>
            <property name="initial"/>
            <property name="lastName"/>
        </component>
    </class>

</hibernate-mapping>
```

Et voici le schéma des tables générées par `SchemaExport`.

```
create table employers (
    id BIGINT not null,
    name VARCHAR(255),
    primary key (id)
)

create table employment_periods (
    id BIGINT not null,
    hourly_rate NUMERIC(12, 2),
    currency VARCHAR(12),
    employee_id BIGINT not null,
    employer_id BIGINT not null,
    end_date TIMESTAMP,
    start_date TIMESTAMP,
    primary key (id)
)

create table employees (
    id BIGINT not null,
    firstName VARCHAR(255),
    initial CHAR(1),
    lastName VARCHAR(255),
    taxfileNumber VARCHAR(255),
    primary key (id)
)

alter table employment_periods
    add constraint employment_periodsFK0 foreign key (employer_id) references employers
alter table employment_periods
    add constraint employment_periodsFK1 foreign key (employee_id) references employees
create sequence employee_id_seq
create sequence employment_id_seq
create sequence employer_id_seq
```

23.2. Auteur/Travail (Author/Work)

Soit le modèle de la relation entre `Work`, `Author` et `Person`. Nous représentons la relation entre `Work` et `Author` comme une association plusieurs-vers-plusieurs. Nous avons choisi de représenter la relation entre `Author` et `Person` comme une association un-vers-un. Une autre possibilité aurait été que `Author` hérite de `Person`.



Le mapping suivant représente exactement ces relations :

```
<hibernate-mapping>

    <class name="Work" table="works" discriminator-value="W">

        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <discriminator column="type" type="character"/>

        <property name="title"/>
        <set name="authors" table="author_work">
            <key column name="work_id"/>
            <many-to-many class="Author" column name="author_id"/>
        </set>

        <subclass name="Book" discriminator-value="B">
            <property name="text"/>
        </subclass>

        <subclass name="Song" discriminator-value="S">
            <property name="tempo"/>
            <property name="genre"/>
        </subclass>

    </class>

    <class name="Author" table="authors">

        <id name="id" column="id">
            <!-- The Author must have the same identifier as the Person -->
            <generator class="assigned"/>
        </id>

        <property name="alias"/>
        <one-to-one name="person" constrained="true"/>

        <set name="works" table="author_work" inverse="true">
            <key column="author_id"/>
            <many-to-many class="Work" column="work_id"/>
        </set>

    </class>
```

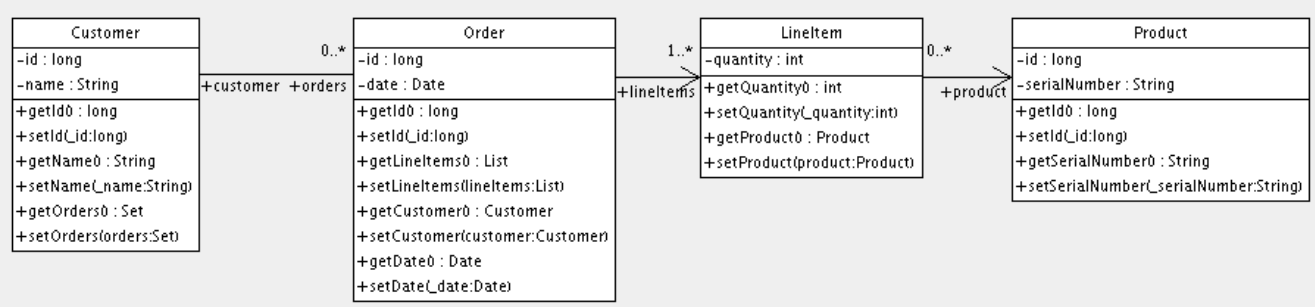
```
    <class name="Person" table="persons">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

Il y a quatre tables dans ce mapping. `works`, `authors` et `persons` qui contiennent respectivement les données de work, author et person. `author_work` est une table d'association qui lie authors à works. Voici le schéma de tables, généré par `SchemaExport`.

```
create table works (
    id BIGINT not null generated by default as identity,
    tempo FLOAT,
    genre VARCHAR(255),
    text INTEGER,
    title VARCHAR(255),
    type CHAR(1) not null,
    primary key (id)
)

create table author_work (
    author_id BIGINT not null,
    work_id BIGINT not null,
    primary key (work_id, author_id)
)

create table authors (
    id BIGINT not null generated by default as identity,
    alias VARCHAR(255),
    primary key (id)
)

create table persons (
    id BIGINT not null generated by default as identity,
    name VARCHAR(255),
    primary key (id)
)

alter table authors
    add constraint authorsFK0 foreign key (id) references persons
alter table author_work
    add constraint author_workFK0 foreign key (author_id) references authors
alter table author_work
    add constraint author_workFK1 foreign key (work_id) references works
```

23.3. Client/Commande/Produit (Customer/Order/Product)

Imaginons maintenant le modèle de relation entre `Customer`, `Order`, `LineItem` et `Product`. Il y a une association un-vers-plusieurs entre `Customer` et `Order`, mais comment devrions nous représenter `Order` / `LineItem` / `Product`? J'ai choisi de mapper `LineItem` comme une classe d'association représentant l'association plusieurs-vers-plusieurs entre `Order` et `Product`. Dans Hibernate, on appelle cela un élément composite.



Le document de mapping :

```
<hibernate-mapping>

    <class name="Customer" table="customers">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
        <set name="orders" inverse="true">
            <key column="customer_id"/>
            <one-to-many class="Order"/>
        </set>
    </class>

    <class name="Order" table="orders">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="date"/>
        <many-to-one name="customer" column="customer_id"/>
        <list name="lineItems" table="line_items">
            <key column="order_id"/>
            <list-index column="line_number"/>
            <composite-element class="LineItem">
                <property name="quantity"/>
                <many-to-one name="product" column="product_id"/>
            </composite-element>
        </list>
    </class>

    <class name="Product" table="products">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="serialNumber"/>
    </class>

</hibernate-mapping>
```

`customers`, `orders`, `line_items` et `products` contiennent les données de customer, order, order line item et product. `line_items` est aussi la table d'association liant orders à products.

```
create table customers (
    id BIGINT not null generated by default as identity,
    name VARCHAR(255),
    primary key (id)
)

create table orders (
    id BIGINT not null generated by default as identity,
    customer_id BIGINT,
    date TIMESTAMP,
    primary key (id)
)

create table line_items (
    line_number INTEGER not null,
```

```
    order_id BIGINT not null,
    product_id BIGINT,
    quantity INTEGER,
    primary key (order_id, line_number)
)

create table products (
    id BIGINT not null generated by default as identity,
    serialNumber VARCHAR(255),
    primary key (id)
)

alter table orders
    add constraint ordersFK0 foreign key (customer_id) references customers
alter table line_items
    add constraint line_itemsFK0 foreign key (product_id) references products
alter table line_items
    add constraint line_itemsFK1 foreign key (order_id) references orders
```

23.4. Divers mappings d'exemple

Ces exemples sont tous pris de la suite de tests d'Hibernate. Vous en trouverez beaucoup d'autres. Regardez dans le dossier `test` de la distribution d'Hibernate.

TODO: put words around this stuff

23.4.1. "Typed" one-to-one association

```
<class name="Person">
    <id name="name"/>
    <one-to-one name="address"
            cascade="all">
        <formula>name</formula>
        <formula>'HOME'</formula>
    </one-to-one>
    <one-to-one name="mailingAddress"
            cascade="all">
        <formula>name</formula>
        <formula>'MAILING'</formula>
    </one-to-one>
</class>

<class name="Address" batch-size="2"
        check="addressType in ('MAILING', 'HOME', 'BUSINESS')">
    <composite-id>
        <key-many-to-one name="person"
                column="personName"/>
        <key-property name="type"
                column="addressType"/>
    </composite-id>
    <property name="street" type="text"/>
    <property name="state"/>
    <property name="zip"/>
</class>
```

23.4.2. Exemple de clef composée

```
<class name="Customer">

    <id name="customerId"
        length="10">
        <generator class="assigned"/>
```

```
    </id>

    <property name="name" not-null="true" length="100"/>
    <property name="address" not-null="true" length="200"/>

    <list name="orders"
            inverse="true"
            cascade="save-update">
        <key column="customerId"/>
        <index column="orderNumber"/>
        <one-to-many class="Order"/>
    </list>

</class>

<class name="Order" table="CustomerOrder" lazy="true">
    <synchronize table="LineItem"/>
    <synchronize table="Product"/>

    <composite-id name="id"
            class="Order$Id">
        <key-property name="customerId" length="10"/>
        <key-property name="orderNumber"/>
    </composite-id>

    <property name="orderDate"
            type="calendar_date"
            not-null="true"/>

    <property name="total">
        <formula>
            ( select sum(li.quantity*p.price)
            from LineItem li, Product p
            where li.productId = p.productId
                and li.customerId = customerId
                and li.orderNumber = orderNumber )
        </formula>
    </property>

    <many-to-one name="customer"
            column="customerId"
            insert="false"
            update="false"
            not-null="true"/>

    <bag name="lineItems"
            fetch="join"
            inverse="true"
            cascade="save-update">
        <key>
            <column name="customerId"/>
            <column name="orderNumber"/>
        </key>
        <one-to-many class="LineItem"/>
    </bag>

</class>

<class name="LineItem">

    <composite-id name="id"
            class="LineItem$Id">
        <key-property name="customerId" length="10"/>
        <key-property name="orderNumber"/>
        <key-property name="productId" length="10"/>
    </composite-id>

    <property name="quantity"/>

    <many-to-one name="order"
            insert="false"
```

```
            update="false"
            not-null="true">
        <column name="customerId"/>
        <column name="orderNumber"/>
    </many-to-one>

    <many-to-one name="product"
            insert="false"
            update="false"
            not-null="true"
            column="productId"/>

</class>

<class name="Product">
    <synchronize table="LineItem"/>

    <id name="productId"
        length="10">
        <generator class="assigned"/>
    </id>

    <property name="description"
        not-null="true"
        length="200"/>
    <property name="price" length="3"/>
    <property name="numberAvailable"/>

    <property name="numberOrdered">
        <formula>
            ( select sum(li.quantity)
            from LineItem li
            where li.productId = productId )
        </formula>
    </property>

</class>
```

23.4.3. Many-to-many avec une clef composée partagée

```
<class name="User" table="`User`">
    <composite-id>
        <key-property name="name"/>
        <key-property name="org"/>
    </composite-id>
    <set name="groups" table="UserGroup">
        <key>
            <column name="userName"/>
            <column name="org"/>
        </key>
        <many-to-many class="Group">
            <column name="groupName"/>
            <formula>org</formula>
        </many-to-many>
    </set>
</class>

<class name="Group" table="`Group`">
    <composite-id>
        <key-property name="name"/>
        <key-property name="org"/>
    </composite-id>
    <property name="description"/>
    <set name="users" table="UserGroup" inverse="true">
        <key>
            <column name="groupName"/>
            <column name="org"/>
        </key>
```

```
        <many-to-many class="User">
            <column name="userName"/>
            <formula>org</formula>
        </many-to-many>
    </set>
</class>
```

23.4.4. Contenu basé sur une discrimination

```
<class name="Person"
    discriminator-value="P">

    <id name="id"
        column="person_id"
        unsaved-value="0">
        <generator class="native"/>
    </id>


    <discriminator
        type="character">
        <formula>
            case
                when title is not null then 'E'
                when salesperson is not null then 'C'
                else 'P'
            end
        </formula>
    </discriminator>

    <property name="name"
        not-null="true"
        length="80"/>

    <property name="sex"
        not-null="true"
        update="false"/>

    <component name="address">
        <property name="address"/>
        <property name="zip"/>
        <property name="country"/>
    </component>

    <subclass name="Employee"
        discriminator-value="E">
            <property name="title"
                length="20"/>
            <property name="salary"/>
            <many-to-one name="manager"/>
    </subclass>

    <subclass name="Customer"
        discriminator-value="C">
            <property name="comments"/>
            <many-to-one name="salesperson"/>
    </subclass>

</class>
```

23.4.5. Associations sur des clefs alternées

```
<class name="Person">

    <id name="id">
        <generator class="hilo"/>
```

```
    </id>

    <property name="name" length="100"/>

    <one-to-one name="address"
        property-ref="person"
        cascade="all"
        fetch="join"/>

    <set name="accounts"
        inverse="true">
        <key column="userId"
            property-ref="userId"/>
        <one-to-many class="Account"/>
    </set>

    <property name="userId" length="8"/>

</class>

<class name="Address">

    <id name="id">
        <generator class="hilo"/>
    </id>

    <property name="address" length="300"/>
    <property name="zip" length="5"/>
    <property name="country" length="25"/>
    <many-to-one name="person" unique="true" not-null="true"/>

</class>

<class name="Account">
    <id name="accountId" length="32">
        <generator class="uuid"/>
    </id>

    <many-to-one name="user"
        column="userId"
        property-ref="userId"/>

    <property name="type" not-null="true"/>

</class>
```

Chapitre 24. Meilleures pratiques

Découpez finement vos classes et mappez les en utilisant `<component>`.

Utilisez une classe `Adresse` pour encapsuler `Rue`, `Region`, `CodePostal`. Ceci permet la réutilisation du code et simplifie la maintenance.

Déclarez des propriétés d'identifiants dans les classes persistantes.

Hibernate rend les propriétés d'identifiants optionnelles. Il existe beaucoup de raisons pour lesquelles vous devriez les utiliser. Nous recommandons que vous utilisiez des identifiants techniques (générés, et sans connotation métier).

Identifiez les clefs naturelles.

Identifiez les clefs naturelles pour toutes les entités, et mappez les avec `<natural-id>`. Implémentez `equals()` et `hashCode()` pour comparer les propriétés qui composent la clef naturelle.

Placez chaque mapping de classe dans son propre fichier.

N'utilisez pas un unique document de mapping. Mappez `com.eg.Foo` dans le fichier `com/eg/Foo.hbm.xml`. Cela prend tout son sens lors d'un travail en équipe.

Chargez les mappings comme des ressources.

Déployez les mappings en même temps que les classes qu'ils mappent.

Pensez à externaliser les chaînes de caractères.

Ceci est une bonne habitude si vos requêtes appellent des fonctions SQL qui ne sont pas au standard ANSI. Cette externalisation dans les fichiers de mapping rendra votre application plus portable.

Utilisez les variables "bindées".

Comme en JDBC, remplacez toujours les valeurs non constantes par "?". N'utilisez jamais la manipulation des chaînes de caractères pour remplacer des valeurs non constantes dans une requête ! Encore mieux, utilisez les paramètres nommés dans les requêtes.

Ne gérez pas vous même les connexions JDBC.

Hibernate laisse l'application gérer les connexions JDBC. Vous ne devriez gérer vos connexions qu'en dernier recours. Si vous ne pouvez pas utiliser les systèmes de connexions livrés, réfléchissez à l'idée de fournir votre propre implémentation de `org.hibernate.connection.ConnectionProvider`.

Pensez à utiliser les types utilisateurs.

Supposez que vous ayez une type Java, de telle bibliothèque, qui a besoin d'être persisté mais qui ne fournit pas les accesseurs nécessaires pour le mapper comme composant. Vous devriez implémenter `org.hibernate.UserType`.Cette approche libère le code de l'application de l'implémentation des transformations vers / depuis les types Hibernate.

Utilisez du JDBC pur dans les goulets d'étranglement.

Dans certaines parties critiques de votre système d'un point de vue performance, quelques opérations peuvent tirer partie d'un appel JDBC natif. Mais attendez de *savoir* que c'est un goulet d'étranglement. Ne supposez jamais qu'un appel JDBC sera forcément plus rapide. Si vous avez besoin d'utiliser JDBC directement, ouvrez une `Session` Hibernate et utilisez la connexion SQL sous-jacente. Ainsi vous pourrez utiliser la même stratégie de transation et la même gestion des connexions.

Comprendre le flush de `Session`.

De temps en temps la Session synchronise ses états persistants avec la base de données. Les performances seront affectées si ce processus arrive trop souvent. Vous pouvez parfois minimiser les flush non nécessaires en désactivant le flush automatique ou même en changeant l'ordre des opérations menées dans

une transaction particulière.

Dans une architecture à trois couches, pensez à utiliser `saveOrUpdate()`.

Quand vous utilisez une architecture à base de servlet / session bean, vous pourriez passer des objets chargés dans le bean session vers et depuis la couche servlet / JSP. Utilisez une nouvelle session pour traiter chaque requête. Utilisez `Session.merge()` ou `Session.saveOrUpdate()` pour synchroniser les objets avec la base de données.

Dans une architecture à deux couches, pensez à utiliser la déconnexion de session.

Les transactions de bases de données doivent être aussi courtes que possible pour une meilleure montée en charge.Cependant, il est souvent nécessaire d'implémenter de longues *transactions applicatives*, une simple unité de travail du point de vue de l'utilisateur. Une transaction applicative peut s'étaler sur plusieurs cycles de requêtes/réponses du client. Il est commun d'utiliser des objets détachés pour implémenter des transactions applicatives. Une alternative, extrêmement appropriée dans une architecture à 2 couches, est de maintenir un seul contact de persistance ouvert (session) pour toute la durée de vie de la transaction applicative et simplement se déconnecter de la connexion JDBC à la fin de chaque requête, et se reconnecter au début de la requête suivante. Ne partagez jamais une seule session avec plus d'une transaction applicative, ou vous travaillerez avec des données périmées.

Considérez que les exceptions ne sont pas rattrapables.

Il s'agit plus d'une pratique obligatoire que d'une "meilleure pratique". Quand une exception intervient, il faut faire un rollback de la `Transaction` et fermer la `Session`. Sinon, Hibernate ne peut garantir l'intégrité des états persistants en mémoire. En particulier, n'utilisez pas `Session.load()` pour déterminer si une instance avec un identifiant donné existe en base de données, utilisez `Session.get()` ou un requête.

Préférez le chargement tardif des associations.

Utilisez le chargement complet avec modération. Utilisez les proxies et les collections chargées tardivement pour la plupart des associations vers des classes qui ne sont pas susceptibles d'être complètement retenues dans le cache de second niveau. Pour les assocations de classes en cache, où il y a une extrêmement forte probabilité que l'élément soit en cache, désactivez explicitement le chargement par jointures ouvertes en utilisant `outer-join="false"`. Lorsqu'un chargement par jointure ouverte est approprié pour un cas d'utilisation particulier, utilisez une requête avec un `left join fetch`.

Utilisez le pattern *d'une ouverture de session dans une vue*, ou une *phase d'assemblage* disciplinée pour éviter des problèmes avec des données non rapatriées.

Hibernate libère les développeurs de l'écriture fastidieuse des *objets de transfert de données (NdT : Data Transfer Objects)* (DTO). Dans une architecture EJB traditionnelle, les DTOs ont deux buts : premièrement, ils contournent le problème des "entity bean" qui ne sont pas sérialisables ; deuxièmement, ils définissent implicitement une phase d'assemblage où toutes les données utilisées par la vue sont rapatriées et organisées dans les DTOs avant de retourner sous le contrôle de la couche de présentation. Hibernate élimine le premier but. Pourtant, vous aurez encore besoin d'une phase d'assemblage (pensez vos méthodes métier comme ayant un contrat strict avec la couche de présentation à propos de quelles données sont disponibles dans les objets détachés) à moins que vous soyez préparés à garder le contexte de persistance (la session) ouvert à travers tout le processus de rendu de la vue.

Pensez à abstraite votre logique métier d'Hibernate.

Cachez le mécanisme d'accès aux données (Hibernate) derrière une interface. Combinez les patterns *DAO* et *Thread Local Session*. Vous pouvez même avoir quelques classes persistées par du JDBC pur, associées à Hibernate via un `UserType` (ce conseil est valable pour des applications de taille respectables ; il n'est pas valable pour une application avec cinq tables).

N'utilisez pas d'associations de mapping exotiques.

De bons cas d'utilisation pour de vraies associations plusieurs-vers-plusieurs sont rares. La plupart du temps vous avez besoin d'informations additionnelles stockées dans la table d'association. Dans ce cas, il est

préférable d'utiliser deux associations un-vers-plusieurs vers une classe de liaisons intermédiaire. En fait, nous pensons que la plupart des associations sont de type un-vers-plusieurs ou plusieurs-vers-un, vous devez être très attentifs lorsque vous utilisez autre chose et vous demander si c'est vraiment nécessaire.

Préférez les associations bidirectionnelles.

Les associations unidirectionnelles sont plus difficiles à questionner. Dans une grande application, la plupart des associations devraient être navigables dans les deux directions dans les requêtes.