Documentazione di Riferimento

Version: 2.1.6

Sommario

Prefazione

WARNING! This is a translated version of the English Hibernate reference documentation. The translated version might not be up to date! However, the differences should only be very minor. Consult the English reference documentation if you are missing information or encounter a translation error. If you like to contribute to a particular translation, contact us on the Hibernate developer mailing list.

Translator(s): Davide Baroncelli <baroncelli@yahoo.com>

Lavorare con linguaggi orientati agli oggetti e database relazionale negli odierni ambienti aziendali può essere difficoltoso e temporalmente impegnativo. Hibernate è uno strumento di mappaggio tra oggetti e relazioni (OR) per gli ambienti basati su java. Il termine "mappaggio oggetto-relazionale" ("object relational mapping" o ORM in inglese) si riferisce alla tecnica di creare una corrispondenza (mappare) tra una rappresentazione di dati secondo il modello a oggetti ed quella secondo il modello relazionale, con uno schema basato su SQL.

Hibernate si occupa non solo del mappaggio dalle classi Java alle tabelle della base di dati (e dai tipi di dato Java a quelli SQL), ma fornisce anche funzionalità di interrogazione e recupero dei dati (query), e può ridurre significativamente i tempi di sviluppo altrimenti impiegati in attività manuali di gestione dei dati in SQL e JDBC.

Lo scopo di Hibernate è di alleviare lo sviluppatore dal 95% dei più comuni compiti di programmazione legati alla persistenza dei dati. Hibernate può non essere la soluzione migliore per le applicazioni data-centriche che usano solo stored-procedures per implementare la logica di business nel database; è principalmente utile con modelli di dominio orientati agli oggetti in cui la logica di business sia collocata nello strato intermedio di oggetti Java (middle-tier). In ogni caso, Hibernate può aiutare senza dubbio a rimuovere o incapsulare codice SQL che sia dipendente dal particolare fornitore, e aiutare con i compiti più comuni di traduzione dei set di risultati (result set) da una rappresentazione tabellare ad un grafo di oggetti.

Se sei un principiante di Hibernate e del mappaggio OR o addirittura di Java, ti preghiamo di seguire questi passi:

1. Leggi Capitolo 1, *Primi passi con Tomcat*: è un'introduzione di mezz'ora che usa Tomcat.
2. Leggi Capitolo 2, *Architettura* per capire in quali contesti Hibernate può essere usato.
3. Dai un'occhiata alla cartella `eg/` nella distribuzione di Hibernate, contiene una semplice applicazione "standalone". Copia il tuo driver JDBC nella cartella `lib/` e modifica `src/hibernate.properties`, specificando valori corretti per il tuo database. Da una finestra di terminale aperta nella cartella di distribuzione digita `ant eg` (usando Ant) o, se in Windows, digita `build eg`.
4. Usa questa documentazione di riferimento come la tua fonte principale di informazioni.
5. Sul sito di Hibernate è possibile trovare le risposte alle domande più frequenti (FAQ)
6. Sempre sul sito, è possibile trovare esempi, demo di terze parti e tutorial.
7. L'area di community sul sito di Hibernate è una buona fonte di pattern di design e varie soluzioni di utilizzo integrato di Hibernate (con Tomcat, Jboss, Spring, Struts, EJB, ecc.).

Se hai domande, usa il forum di cui trovi il link sul sito di Hibernate. È anche disponibile un sistema di "issue tracking" (tracciamento dei problemi) JIRA per segnalare bachi e richiedere funzionalità. Se sei interessato nello sviluppo di Hibernate, iscriviti alla mailing list degli sviluppatori.

Sono disponibili contratti di sviluppo su base commerciale, supporto per sistemi in produzione e formazione su

Hibernate tramite il JBoss Inc. (vedi http://www.hibernate.org/SupportTraining/). Hibernate è un progetto della suite di prodotti "JBoss Professional Open Source".

Capitolo 1. Primi passi con Tomcat

1.1. Iniziare a lavorare con Hibernate

Questo corso introduttivo mostra come installare Hibernate 2.1 nel servlet container Apache Tomcat per realizzare una applicazione web. Hibernate funziona bene sia in ambienti gestiti dai principali server J2EE, sia in applicazioni java a sé stanti. Il sistema di gestione di basi di dati (DBMS) usato in questa introduzione è PostgreSQL 7.3, ma per farla funzionare su altri database è solo necessario modificare la configurazione del dialetto SQL che viene usato da Hibernate.

Prima di tutto, dobbiamo copiare tutte le librerie richieste nell'installazione di Tomcat. Usiamo un contesto web separato (`webapps/quickstart`) per questa introduzione, e quindi dobbiamo considerare sia il percorso di ricerca globale delle librerie (`TOMCAT/common/lib`) sia il classloader al livello del contesto in `webapps/quickstart/WEB-INF/lib` (per i file JAR) e `webapps/quickstart/WEB-INF/classes`. Ci riferiremo ai due livelli di classloader con i termini di "classpath globale" e "classpath di contesto".

Ora, copiate le librerie nei due classpath:

1. Copiate il driver JDBC per il database nel classpath globale. Questo è richiesto per il software di gestione dei "lotti di connessioni" (connection pool) DBCP che è preinstallato con Tomcat. Hibernate usa le connessioni JDBC per eseguire l'SQL sul database, così gli si deve fornire connessioni JDBC provenienti da un pool o configurare Hibernate in modo tale da usare uno dei pool supportati direttamente (C3P0, Proxool). Per questo particolare tutoriale, copiate la libreria `pg73jdbc3.jar` (per ostgreSQL 7.3 e il JDK 1.4) nel percorso globale. Se voleste usare un database differente, copiate semplicemente il driver JDBC appropriato.

2. Non copiate mai niente altro nel classpath globale di Tomcat, o avrete problemi con vari tool, compreso log4j, commons-logging e altri. Usate sempre il classpath di contesto per ogni applicazione web, cioè copiate le librerie in `WEB-INF/lib` e le vostre classi e file di configurazione/proprietà in `WEB-INF/classes`. Entrambe le cartelle sono situate per default nel classpath a livello di contesto.

3. Hibernate è distribuito come una libreria JAR. Il file `hibernate2.jar` dovrebbe venire copiato nel classpath di contesto insieme con le altre classi dell'applicazione. Hibernate durante l'esecuzione richiede alcune librerie fornite da terze parti: queste sono fornite con la distribuzione di Hibernate nella cartella `lib/`; vedete Tabella 1.1, “ Librerie esterne richieste da Hibernate ”. Copiate ora le librerie richieste nel classpath di contesto.

Tabella 1.1. Librerie esterne richieste da Hibernate

Libreria	Descrizione
dom4j (obbligatoria)	Hibernate usa dom4j per fare il parsing dei file di configurazione XML, così come i file di metadati (sempre in XML).
CGLIB (obbligatoria)	Hibernate usa questa libreria di generazione del codice per potenziare le classi all'avvio (in combinazione con la "reflection" di Java)
Commons Collections, Commons Logging (obbligatorie)	Hibernate usa varie librerie di utilità provenienti dal progetto Apache Jakarta Commons.
ODMG4 (obbligatoria)	Hibernate fornisce un gestore di persistenza opzionale compatibile con la specifica ODMC. Se volete mappare delle collezioni, è obbligatorio

Libreria	Descrizione
	anche se non intendete usare l'API ODMG. Non mapperemo collezioni, in questo articolo introduttivo, ma è comunque una buona idea copiare il JAR.
EHCache (obbligatoria)	Hibernate può usare diversi fornitori di cache per la cache di secondo livello. EHCache è la cache predefinita, se non viene cambiata nella configurazione.
Log4j (opzionale)	Hibernate usa l'API di Commons Logging, che a sua volta può usare Log4j come meccanismo sottostante di gestione delle tracce di esecuzione (logging). Se la libreria di Log4j è disponibile nella cartella di contesto, Commons Logging userà Log4j e il file di configurazione `log4j.properties` situato nel classpath di contesto. Un file di esempio per la configurazione di Log4j è fornito con la distribuzione di Hibernate. Quindi, copiate log4j,jar ed il file di configurazione (da `src/`) nel vostro classpath di contesto se volete vedere cosa succede dietro al sipario.
Richiesta o no?	Date un'occhiata al file `lib/README.txt` nella distribuzione di Hibernate. È una lista aggiornata delle librerie di terze parti distribuite con Hibernate. Vi troverete elencate tutte le librerie opzionali ed obbligatorie.

Ora configureremo il pooling delle connessioni e la condivisione sia in Tomcat sia in Hibernate. Questo significa che tomcat fornirà connessioni JDBC estratte da un pool (usando le funzionalità offerte dalla libreria DBCP inclusa), e Hibernate richiederà queste connessioni via JNDI. Tomcat collega il pool di connessioni al JNDI se aggiungiamo la dichiarazione della risorsa al file di configurazione principale di Tomcat, `TOMCAT/conf/server.xml`:

```
<Context path="/quickstart" docBase="quickstart">
    <Resource name="jdbc/quickstart" scope="Shareable" type="javax.sql.DataSource"/>
    <ResourceParams name="jdbc/quickstart">
        <parameter>
            <name>factory</name>
            <value>org.apache.commons.dbcp.BasicDataSourceFactory</value>
        </parameter>

        <!-- DBCP database connection settings -->
        <parameter>
            <name>url</name>
            <value>jdbc:postgresql://localhost/quickstart</value>
        </parameter>
        <parameter>
            <name>driverClassName</name><value>org.postgresql.Driver</value>
        </parameter>
        <parameter>
            <name>username</name>
            <value>quickstart</value>
        </parameter>
        <parameter>
            <name>password</name>
            <value>secret</value>
        </parameter>

        <!-- DBCP connection pooling options -->
        <parameter>
            <name>maxWait</name>
            <value>3000</value>
        </parameter>
        <parameter>
            <name>maxIdle</name>
            <value>100</value>
        </parameter>
```

```
        <parameter>
            <name>maxActive</name>
            <value>10</value>
        </parameter>
    </ResourceParams>
</Context>
```

Il contesto che configuriamo in questo esempio di chiamaquickstart, ed ha base nella cartella `TOMCAT/webapp/quickstart`. Per accedere dei servlet, bisogna chiamare il percorso `http://localhost:8080/quickstart` nel browser (naturalmente, aggiungendo il nome del servlet così come è mappato nel file `web.xml`). Potete anche procedere e creare un semplice servlet, che abbia un metodo `process()` vuoto

Tomcat usa il pool di connessioni DBCP con questa configurazione e fornisce `Connessioni` JDBC da un pool reperito tramite JNDI all'indirizzo `java:comp/env/jdbc/quickstart`. Se avete problemi a far funzionare il pool di connessioni, fate riferimento alla documentazione di Tomcat. Se ricevete messaggi di eccezione dal driver JDBC, provate prima ad impostare il pool senza Hibernate. Sul web è possibile trovare degli articoli introduttivi sia su Tomcat sia su JDBC.

Il prossimo passo è configurare Hibernate, usando le connessioni dal pool collegato al JNDI. Usiamo la configurazione XML di Hibernate. L'approccio più semplice, che usa file di proprietà, è equivalente in funzionalità ma non offre nessun vantaggio. Usiamo quindi la configurazione XML perché di solito è più conveniente: il file di configurazione XML si trova nel classpath di contesto (`WEB-INF/classes`), come `hibernate.cfg.xml`:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration
    PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">

<hibernate-configuration>

    <session-factory>

        <property name="connection.datasource">java:comp/env/jdbc/quickstart</property>
        <property name="show_sql">false</property>
        <property name="dialect">net.sf.hibernate.dialect.PostgreSQLDialect</property>

        <!-- Mapping files -->
        <mapping resource="Cat.hbm.xml"/>

    </session-factory>

</hibernate-configuration>
```

Disattiviamo il tracciamento dei comandi SQL e diciamo ad Hibernate quale dialetto SQL deve venire usato, e dove prendere le connessioni JDBC (dichiarando l'indirizzo del datasource che dà accesso al pool collegato in Tomcat). Il dialetto è un'impostazione necessaria, poiché i database differiscono nella loro interpretazione dello "standard" SQL. Hibernate gestisce le differenze e viene fornito con dialetti per tutti i principali database commerciali e open source.

La `SessionFactory` è il concetto che in Hibernate corrisponde ad un contenitore di dati (datastore) univoco. Database multipli possono essere usati creando file di configurazione XML multipli e creando più oggetti`Configuration` e `SessionFactory` nella vostra applicazione.

L'ultimo elemento del file `hibernate.cfg.xml` dichiara `Cat.hbm.xml` come nome di un file di mappaggio XML di Hibernate per la classe persistente `Cat`. Questo file contiene i metadati per il mappaggio delle classi POJO (acronimo che sta per "plain old java object", ovvero più o meno "oggetto java puro e semplice", contrapposto ad oggetti che implementano interfacce particolari come gli Enterprise Javabeans) verso una tabella di database (o verso tabelle multiple). Torneremo presto su questo file, prima però scriviamo la classe POJO e poi dichiariamo i suoi metadati di mappaggio.

1.2. La prima classe persistente

Hibernate funziona meglio con il modello degli oggetti POJO per le classi persistenti. Un POJO è più o meno come un JavaBean, con proprietà accessibili tramite metodi "getter" e "setter" (rispettivametne per recuperare e impostare le proprietà), mascherando la rappresentazione interna tramite l'interfaccia pubblicamente visibile:

```
package net.sf.hibernate.examples.quickstart;

public class Cat {

    private String id;
    private String name;
    private char sex;
    private float weight;

    public Cat() {
    }

    public String getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public char getSex() {
        return sex;
    }

    public void setSex(char sex) {
        this.sex = sex;
    }

    public float getWeight() {
        return weight;
    }

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

}
```

Hibernate non è limitato nel suo uso dei tipi di proprietà Java, tutti i tipi ed i tipi primitivi del JDK (come`String`, `char` e `Date`) possono essere mappati, comprese le classi dal framework delle collezioni. Potete mapparle come valori, collezioni di valori o associazioni verso altre entità. La proprietà `id` è una proprietà speciale che rappresenta l'identificatore principale nel database per quella classe (chiave primaria), ed è fortemente raccomandato per entità come `Cat`. Hibernate può anche usare identificatori gestiti solo internamente, ma perderemmoe una parte della flessibilità nella nostra architettura applicativa.

Per le classi persistenti non è richiesta l'implementazione di alcuna interfaccia particolare, né dobbiamo ereditare da una speciale classe persistente radice. Hibernate non usa neppure alcun tipo di computazione in fase di "build" (costruzione del software), come manipolazione del codice binario (byte-code): si appoggia esclusivamente su reflection Java e su potenziamento in fase di esecuzione delle classi (tramite CGLIB). Così, possiamo

mappare le classi POJO sul database senza alcuna dipendenza da Hibernate.

1.3. Mappare il gatto

Il file di mappaggio `Cat.hbm.xml` contiene i metadati richiesti per il mappaggio oggetto-relazione. I metadati includono la dichiarazione delle classi persistenti e il mappaggio delle proprietà sulle tabelle del database (tramite le colonne e le relazioni con altre entità gestite da chiavi esterne)

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>

    <class name="net.sf.hibernate.examples.quickstart.Cat" table="CAT">

        <!-- A 32 hex character is our surrogate key. It's automatically
            generated by Hibernate with the UUID pattern. -->
        <id name="id" type="string" unsaved-value="null" >
            <column name="CAT_ID" sql-type="char(32)" not-null="true"/>
            <generator class="uuid.hex"/>
        </id>

        <!-- A cat has to have a name, but it shouldn' be too long. -->
        <property name="name">
            <column name="NAME" length="16" not-null="true"/>
        </property>

        <property name="sex"/>

        <property name="weight"/>

    </class>

</hibernate-mapping>
```

Ogni classe persistente dovrebbe avere un attributo di identificazione (in realtà, solo classi che rappresentano entità, e non gli oggetti dipendenti, che sono mappati come componenti di un'entità). Questa proprità viene usata per distinguere oggetti persistenti: due gatti sono uguali se `catA.getId().equals(catB.getId())` è vera: questo concetto è chiamata *identità del database*. Hibernate è fornito con vari generatori di identificatori per scenari differenti (compresi generatori nativi per "sequence" del database, tabelle hi/lo sul database, e identificatori assegnati dall'applicazione. Usiamo il generator UUID generator (raccomandato solo per il testing, poiché chiavi surrogate intere generate dal database dovrebbero venire preferite) e specifichiamo anche la colonna `CAT_ID` della tabella `CAT` per il valore dell'identificatore generato da Hibernate (come chiave primaria della tabella).

Tutte le altre proprietà di `Cat` sono mappate sulla stessa tabella. Nel caseo della proprietà `name`, l'abbiamo mappata con una colonna del database dichiarata esplicitamente. Questo è particolarmente utile quando si voglia che lo schema del database venga generato automaticamente (sotto forma di istruzioni SQL DDL) a partire dai file di mappaggio tramite lo strumento di Hibernate *SchemaExport*. Tutte le altre proprietà vengono mappate usando le impostazioni predefinite di Hibernate, che è ciò di cui si ha bisogno la maggior parte delle volte. La tabella `CAT` nel database appare così:

```
 Column |         Type          | Modifiers
--------+-----------------------+-----------
 cat_id | character(32)         | not null
 name   | character varying(16) | not null
 sex    | character(1)          |
 weight | real                  |
```

```
Indexes: cat_pkey primary key btree (cat_id)
```

Dovreste ora creare manualmente la tabella nel database, e più tardi leggere Capitolo 15, *Guida degli strumenti* se volete automatizzare questo passo con lo strumento SchemaExport. Questo strumento può generare un DDL SQL completo, comprendente definizioni delle tabelle, vincoli sui tipi delle colonne, vincoli di unicità e indici.

1.4. Giochiamo con i gatti

Ora siamo pronti per lanciare la `Session` Hibernate. È l'interfaccia di *gestione della persistenza*, la usiamo per memorizzare e recuperare istanze della classe `Cat` sul e dal database. Prima però dobbiamo recuperare una istanza di `Session` (che è l'unità di lavoro di Hibernate) dalla `SessionFactory`:

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

La `SessionFactory` è responsabile per un singolo database, e può usare un file di configurazione solo (`hibernate.cfg.xml`). Potete impostare altre proprietà (e anche cambiare i metadati di mappaggio) accedendo all'oggetto `Configuration` *prima* di costruire la `SessionFactory` (è immutabile). Ma dove creiamo la `SessionFactory` e come vi accediamo nella nostra applicazione?

Una `SessionFactory` viene solitamente costruita una volta sola, ad esempio all'avvio con un servlet impostato con il parametro *load-on-startup*. Questo significa anche che non dovreste tenerlo in una variabile di itanza nei vostri sevlet, ma in qualche altra posizione. Abbiamo bisogno di qualche tipo di *Singleton*, in modo tale da poter accedere facilmente alla `SessionFactory`. L'approccio che viene mostrato nel seguito mostra entrambi i problemi: configurazione e accesso facile ad una `SessionFactory`.

Implementiamo una classe di utilità `HibernateUtil`:

```
import net.sf.hibernate.*;
import net.sf.hibernate.cfg.*;

public class HibernateUtil {

    private static final SessionFactory sessionFactory;

    static {
        try {
            // Create the SessionFactory
            sessionFactory = new Configuration().configure().buildSessionFactory();
        } catch (HibernateException ex) {
            throw new RuntimeException("Configuration problem: " + ex.getMessage(), ex);
        }
    }

    public static final ThreadLocal session = new ThreadLocal();

    public static Session currentSession() throws HibernateException {
        Session s = (Session) session.get();
        // Open a new Session, if this Thread has none yet
        if (s == null) {
            s = sessionFactory.openSession();
            session.set(s);
        }
        return s;
    }

    public static void closeSession() throws HibernateException {
        Session s = (Session) session.get();
        session.set(null);
        if (s != null)
            s.close();
```

```
    }
}
```

Questa classe non si occupa solo della `SessionFactory` con il suo attributo statico, ma ha anche un `ThreadLocal` per mantenere la `Session` per il thread che è in esecuzione. Assicuratevi di capire il concetto di variabile thread-local in java prima di provare ad usare questa classe.

Una `SessionFactory` è "threadsafe", ovvero vari thread possono accedervi concorrentemente e richiedere oggetti `Session`. Una `Session` è un oggetto non-threadsafe, che rappresenta una singola unità di lavoro con il database. Le `Session` vengono aperte da una `SessionFactory` e vengono chiuse quando tutto il lavoro è completato.

```
Session session = HibernateUtil.currentSession();

Transaction tx= session.beginTransaction();

Cat princess = new Cat();
princess.setName("Princess");
princess.setSex('F');
princess.setWeight(7.4f);

session.save(princess);
tx.commit();

HibernateUtil.closeSession();
```

In una `Session`, ogni operazione sul database avviene in una transazione che isola le operazioni (anche quelle di sola lettura). Usiamo l'API `Transaction` di Hibernate per astrarre dalla strategia transazionale sottostante (nel nostro caso, transazioni JDBC). Questo consente di mettere in esecuzione il nostro codice con transazioni gestite dal contenitore (usando JTA) senza alcun cambiamento. Notate che l'esempio sopra non gestisce alcuna eccezione.

Notate anche che potete chiamare `HibernateUtil.currentSession();` tutte le volte che volete, e otterrete sempre la `Session` corrente per il thread. Dovete assicurarvi che la `Session` venga chiusa dopo che l'unità di lavoro si completi, o nel servlet o in un servlet filter prima che la risposta HTTP venga inviata. L'effetto collaterale piacevole dell'ultimo approccio è un facile accesso ad un meccanismo di inizializzazione "lazy" (pigro, ovvero che carica i dati solo quando servono): la `Session` è ancora aperta quando la pagina viene generata, così Hibernate può caricare oggetti non inizializzati quando navigate nel grafo.

Hibernate ha vari metodi che possono essere usati per recuperare oggetti dal database. Il più flessibile è usare il linguaggio di query di HIbernate(HQL), che è facile da imparare ed una estensinoe potente ed orientata agli oggetti dell'SQL:

```
Transaction tx = session.beginTransaction();

Query query = session.createQuery("select c from Cat as c where c.sex = :sex");
query.setCharacter("sex", 'F');
for (Iterator it = query.iterate(); it.hasNext();) {
    Cat cat = (Cat) it.next();
    out.println("Female Cat: " + cat.getName() );
}

tx.commit();
```

Hibernate offre anche un'API ad oggetti di *query per criteri* che può essere usata per formulare query "type-safe" (ovvero il cui tipo viene verificato in fase di compilazione). Hibernate usa naturalmente oggetti `PreparedStatement` e binding di parametri per tutte le comunicazioni SQL con il database. Potete anche usare le funzionalità di interrogazione diretta via SQL di Hibernate, o ricevere una connessione JDBC da una `Session`.

1.5. Infine

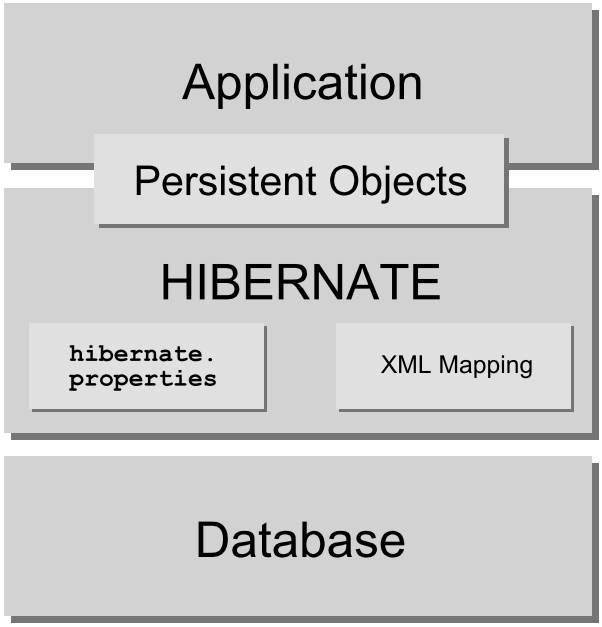
Abbiamo solo sfiorato la superficie di Hibernate in questo breve articolo. Notate che non includiamo alcun codice specifico per i servlet nei nostri esempi: dovete creare un serlvet voi stessi ed inserire il codice di Hibernate come preferite.

Ricordate che Hibernate, come strato di accesso ai dati, è strettamente integrato nella vostra applicazione. Solitamente, tutti gli altri strati dipendono dal meccanismo di persistenza: siate certi di comprendere le implicazioni di questo design.

Capitolo 2. Architettura

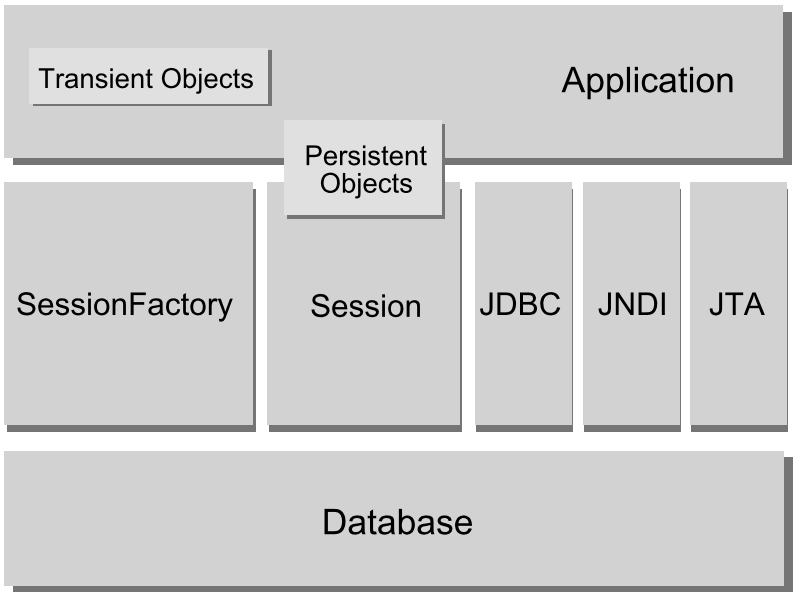
2.1. Introduzione

Una visione (molto) di alto livello sull'architettura di Hibernate:



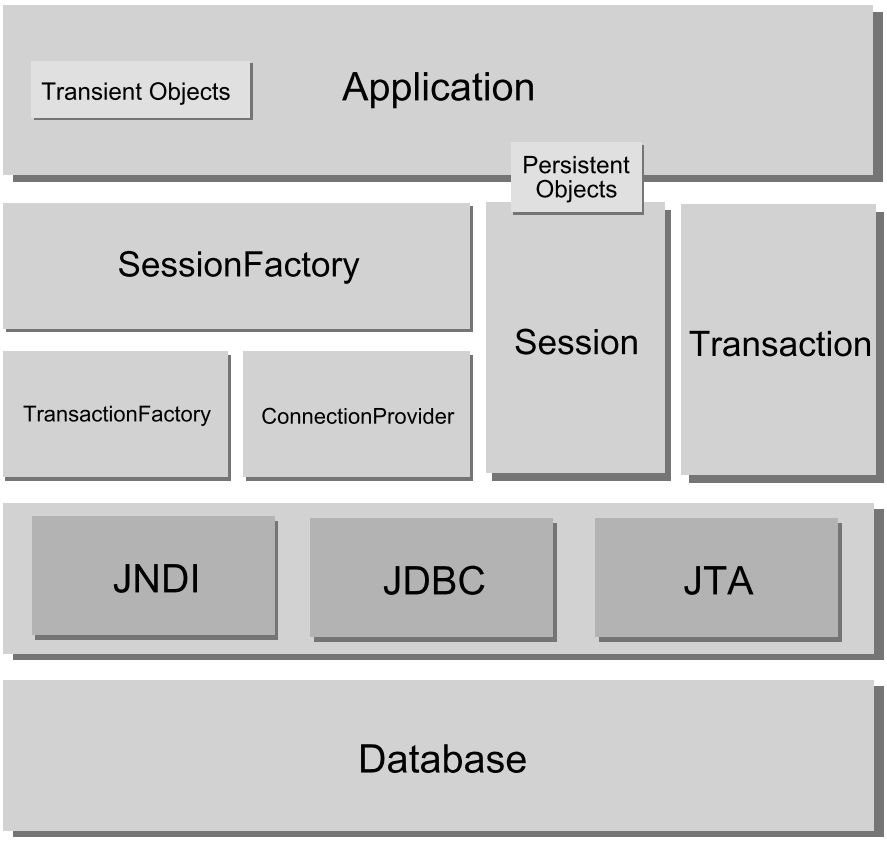
Questo diagramma mostra come Hibernate usa il database e i dati di configurazione per fornire servizi di persistenza (e oggetti persistenti) all'applicazione.

Vorremmo mostrare una vista più dettagliata dell'architettura di runtime. Sfortunatamente Hibernate è flessibile, e rende possibili diversi approcci: mostreremo quindi i due estremi. L'architettura "leggera" è quella in cui l'applicazione fornisce le sue connessioni JDBC e gestisce le transazioni. Questo approccio usa un sottoinsieme minimale delle API di Hibernate.



L'architettura "completa" di Hibernate, permette all'applicazione di astrarre dai dettagli delle API JDBC/JTA e

lascia che se ne occupi Hibernate.



Ecco alcune definizioni degli oggetti nei diagrammi:

SessionFactory (`net.sf.hibernate.SessionFactory`)
: È una cache immutabile e "thread-safe" di mappaggi compilati per un database singolo. Allo stesso tempo è un factory per oggetti `Session` e un client di `ConnectionProvider`. Potrebbe contenere una cache di secondo livello opzionale riutilizzabile tra le transazioni, sia a livello di processo, sia a livello di cluster.

Session (`net.sf.hibernate.Session`)
: È un oggetto mono-thread, di corta durata, che rappresenta una conversazione tra l'applicazione e il contenitore persistente. Incapsula una connessione JDBC. È un factory per oggetti `Transaction`. Mantiene una cache obbligatoria (di primo livello) per gli oggetti persistenti, usata quando si naviga il grafo degli oggetti o si ricercano oggetti per identificatore.

Oggetti persistenti e collezioni
: Sono oggetti di corta durata, a thread singolo, che contengono stato persistente e funzioni applicative. Potrebbero essere normali oggetti POJO/Javabeans, con l'unica particolarità che in un dato momento sono associati con (esattamente) una `Session`. Nel momento in cui la `Session` viene chiusa, verranno staccati e saranno liberi di essere usati in qualsiasi strato applicativo (ad esempio direttamente come oggetti di trasferimento dei dati da e allo strato di presentazione).

Oggetti transienti e collezioni
: Sono le istanze delle classi persistenti che in un dato momento non sono associate con una `Session`. Possono essere state istanziate dall'applicazione e non (ancora) rese persistenti, o possono essere state istanziate da una `Session` poi chiusa.

Transaction (`net.sf.hibernate.Transaction`)
: (Opzionale) È un oggetto a thread singolo, di corta durata, usato dall'applicazione per specificare unità di

lavoro atomiche. Separa le applicazioni dalla transazione JTA, CORBA o JDBC sottostante. Una `Session` potrebbe estendersi lungo varie `Transaction` in certi casi.

ConnectionProvider (`net.sf.hibernate.connection.ConnectionProvider`)
: (Opzionale) Un factory (e pool) di connessioni JDBC. Astrae le applicazioni dai dettagli dei sottostanti `Datasource` o `DriverManager`. Non viene esposta all'applicazione, ma può essere estesa/implementata dagli sviluppatori.

TransactionFactory (`net.sf.hibernate.TransactionFactory`)
: (Opzionale) Un factory per istanze di `Transaction`. Non viene esposta all'applicazione, ma può essere estesa/implementata dagli sviluppatori.

In un'architettura "leggera", l'applicazione aggira le API `Transaction`/`TransactionFactory` e/o `ConnectionProvider` per parlare direttamente con JTA o JDBC.

2.2. Integrazione con JMX

JMX è lo standard J2EE per lo standard o la gestione di componenti Java. Hibernate può essere gestito tramite un MBean standard JMX, ma poiché molti application server non supportano ancora JMX, Hibernate consente anche di usare alcuni sistemi di configurazione non-standard.

Per cortesia, leggete il sito web di Hibernate per informazioni aggiuntive su come configurare Hibernate in modo tale da funzionare come componente JMX in JBoss.

2.3. Supporto a JCA

Hibernate può anche essere configurato come un connettore JCA. Leggete il sito web per altri dettagli.

Capitolo 3. Configurazione della SessionFactory

Poiché Hibernate è progettato per funzionare in molti ambienti differenti, ci sono un gran numero di parametri di configurazione. Fortunatamente, la maggior parte hanno dei valori predefiniti, e Hibernate viene distribuito con un file `hibernate.properties` di esempio che mostra le differenti opzioni possibili. Solitamente è sufficiente mettere quel file nel classpath e applicare le modifiche necessarie per il proprio ambiente.

3.1. Configurazione programmativa

Una istanza di `net.sf.hibernate.cfg.Configuration` rappresenta un insieme completo di mappaggi dei tipi java di una applicazione verso un database SQL. La `Configuration` viene usata per costruire un oggetto `SessionFactory` (immutabile). I mappaggi vengono compilati dai vari file di configurazione XML.

Potete ottenere una istanza di `Configuration` istanziandola direttamnete. Qui di seguito c'è un esempio di impostazione di un contenitore di dati a partire da dei mappaggi definiti in due file di configurazione XML (che si trovano sul classpath):

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

Una maniera alternativa (e in certi casi migliore), è di fare in modo che Hibernate carichi un file di mappaggio usando `getResourceAsStream()`:

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

In questo caso Hibernate cercherà file di mappaggio che si chiamano `/org/hibernate/autcion/Item.hbm.xml` e `/org/hibernate/autcion/Bid.hbm.xml` sul classpath. Questo approccio elimina qualsiasi nome di file cablato nel codice.

Un oggetto `Configuration` specifica anche alcune proprietà opzionali:

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

Una `Configuration` viene considerata un oggetto "da fase di configurazione", che va cioè scartato una volta che una `SessionFactory` sia stata costruita.

3.2. Ottenere una SessionFactory

Quando tutti i mappaggi sono stati interpretati dalla `Configuration`, l'applicazione deve costruire una factory per le istanze di `Session` instances. Questa factory è fatta in modo tale da essere condivisa da tutti i flussi esecutivi (thread) dell'applicazione:

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

Comunque, Hibernate consente alla vostra applicazione di istanziare più di una `SessionFactory`. Questo è utile in particolare se state usando più di un database.

3.3. Connessioni JDBC fornite dall'utente

Una `SessionFactory` può aprire una `Session` su una connessione JDBC fornita dall'utente. Quesa scelta di design dà la libertà all'applicazione di ottenere le connessioni JDBC in qualunque modo preferisca:

```
java.sql.Connection conn = datasource.getConnection();
Session session = sessions.openSession(conn);

// do some data access work
```

L'applicazione deve essere molto attenta a non aprire due `Session`i concorrenti sulla stessa connessione JDBC!

3.4. Connessioni JDBC fornite da Hibernate

In alternativa potete fare in modo che la `SessionFactory` apra le connessioni per voi. La `SessionFactory` deve ricevere le proprietà per le connessioni JDBC in una delle maniere seguenti:

1. Passate una istanza di `java.util.Properties` al metodo `Configuration.setProperties()`.
2. Mettete il file `hibernate.properties` in una directory che si trovi alla radice del classpath.
3. Impostate le proprietà `System` usando `java -Dproperty=value` all'avvio.
4. Includete elementi `<property>` nel file `hibernate.cfg.xml`.

Se seguite questo approccio, aprire una `Session` non è più difficile di così:

```
Session session = sessions.openSession(); // apre una nuova sessione
// fate del lavoro di accesso ai dati, una connessione JDBC sarà usata se ce ne sarà bisogno
```

Tutti i nomi e le semantiche delle proprietà di Hibernate sono definiti nella classe `net.sf.hibernate.cfg.Environment`. Ora descriveremo le impostazioni più importanti per la configurazione delle connessioni JDBC.

Hibernate otterrà le connessioni usando `java.sql.DriverManager` (e le manterrà in un lotto) se impostate le proprietà seguenti:

Tabella 3.1. Proprietà JDBC di Hibernate

Nome della proprietà	Scopo
`hibernate.connection.driver_class`	*classe del driver jdbc*
`hibernate.connection.url`	*URL jdbc*
`hibernate.connection.username`	*nome utente database*
`hibernate.connection.password`	*chiave di accesso al database per l'utente*
`hibernate.connection.pool_size`	*numero massimo di connessioni nel lotto*

L'algoritmo di "pooling" (mantenimento nel lotto) di Hibernate è abbastanza rudimentale. Ha lo scopo di aiutarvi a cominciare a lavorare, ma *non è fatto per l'uso in un sistema in produzione*, o anche solo per dei test di per-

formance. Usate un'altra libreria di pooling per le migliori performance e la stabilità, ovvero sostituite la proprietà `hibernate.connection.pool_size` con le proprietà specifiche per il settaggio del pool.

C3P0 è una libreria open source di pooling per connessioni JDBC che viene distribuita insieme ad Hibernate nella directory `lib`. Hibernate userà il `C3P0ConnectionProvider` integrato per il pooling delle connessioni se settate le proprietà `hibernate.c3p0.*`. C'è anche un supporto integrato per Apache DBCP e per Proxool. Dovete in questo caso impostare le proprietà `hibernate.dbcp.*` per abilitare il `DBCPConnectionProvider`. Il caching dei prepared statement è abilitato se sono impostate le porprietà `hibernate.dbcp.ps.*` (caldamente consigliato). Riferitevi alla documentazione di Apache commons-pool per l'interpretazione di queste proprietà. Se invece volete usare Proxool, settate le proprietà `hibernate.proxool.*`.

Questo è un esempio usando 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.minPoolSize=5
hibernate.c3p0.maxPoolSize=20
hibernate.c3p0.timeout=1800
hibernate.c3p0.max_statement=50
hibernate.dialect = net.sf.hibernate.dialect.PostgreSQLDialect
```

Per l'uso all'interno di un application server, Hibernate può ottenere connessioni da un `javax.sql.Datasource` registrato nel JNDI. Impostate per questo le proprietà seguenti:

Tabella 3.2. Hibernate Datasource Properties

Nome della proprietà	Scopo
`hibernate.connection.datasource`	*Nome JNDI del datasource*
`hibernate.jndi.url`	*URL del provider JNDI* (optional)
`hibernate.jndi.class`	*classe dell'`InitialContextFactory` JNDI* (optional)
`hibernate.connection.username`	*utente del database* (opzionale)
`hibernate.connection.password`	*chiave di accesso al database per l'utente* (opzionale)

Questo è un esempio usando un datasource fornito dal JNDI dell'application server:

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

Connessioni JDBC ottenute da un datasource JNDI parteciperanno automaticamete alle transazioni gestite dal container dell'application server.

Altre proprietà per la connessione possono venire impostate facendo precedere "`hibernate.connnection`" al nome della proprietà. Ad esempio, potete specificare un `charSet` usando `hibernate.connnection.charSet`.

Potete definire la vostra strategia "plugin" per ottenere le connessioni JDBC implementando l'interfaccia `net.sf.hibernate.connection.ConnectionProvider`. Potete selezionare una implementazionee custom impostando `hibernate.connection.provider_class`.

3.5. Configurazione di proprietà opzionali

C'è un certo numero di altre proprietà che controllano il funzionamento in fase di esecuzione di Hibernate. Sono tutte opzionali, e hanno dei valori predefiniti ragionevoli.

Le proprietà a livello di sistema possono essere impostate eslusivamente tramite `java -Dproperty=value` o essere definite in `hibernate.properties` e non con una istanza di `Properties` passata alla classe `Configuration`.

Tabella 3.3. Proprietà per la configurazione di Hibernate

Nome della proprietà	Scopo
`hibernate.dialect`	Il nome della classe di un `Dialect` di Hibernate - attiva alcune funzionalità dipendenti dalla piattaforma di database. *e.g.* `nome.completo.del.Dialect`
`hibernate.default_schema`	Specificate il nome dello schema/tablespace nei nomi delle tabelle nell'SQL che viene generato. *e.g.* `NOME_DELLO_SCHEMA`
`hibernate.session_factory_name`	Il `SessionFactory` verrà automaticamente pubblicato sul JNDI sotto questo nome, se è stato specificato. *e.g.* `nome/composito/jndi`
`hibernate.use_outer_join`	Attiva il reperimento via join esterno. Deprecato, usate `max_fetch_depth`. *e.g.* `true` | `false`
`hibernate.max_fetch_depth`	Imposta una "profondità" massima per l'albero di join esterno che risolve le associazioni ad una singola estremità (uno-a-uno, molti-a-uno). Uno `0` disabilita la risoluzione via join esterno (che invece è attivata per default). *e.g.* i valori raccomandati sono tra `0` e `3`
`hibernate.jdbc.fetch_size`	Un valore non nullo determina le dimensioni di raccolta del JDBC (chiama `Statement.setFetchSize()`).
`hibernate.jdbc.batch_size`	Un valore non nullo abilita l'uso dei "batch update" di JDBC 2 da parte di Hibernate (aggiornamenti in blocco). *e.g.* I valori raccomandati sono tra `5` e `30`
`hibernate.jdbc.use_scrollable_resultset`	Consente l'uso di resultset scrollabili JDBC2 da parte di Hibernate. Questa proprietà è necessaria solo quando vengono usate connessioni JDBC fornite dall'utente: Hibernate usa i metadati di connessione,

Nome della proprietà	Scopo
	altrimenti. *e.g.* `true` | `false`
`hibernate.jdbc.use_streams_for_binary`	Se abilitata, Hibernate usa gli stream quando scrive/legge tipi `binary` o `serializable` da/a JDBC (proprietà di livello di sistema). *e.g.* `true` | `false`
`hibernate.cglib.use_reflection_optimizer`	Abilita l'uso di CGLIB invece di "reflection" in fase di esecuzione (è una proprietà a livello di sistema, il default è usare CGLIB quando possibile). La "reflection" può a volte essere usata in fase di risoluzione dei problemi. *e.g.* `true` | `false`
`hibernate.jndi.<propertyName>`	Passa la proprietà `propertyName` all' `InitialContextFactory` JNDI.
`hibernate.connection.isolation`	Setta il livello di isolamento transazionale JDBC. Consultate la documentazione di `java.sql.Connection` per ottenere i valori significativi, ma tenete presente che la maggior parte dei database non supportano tutti i livelli di isolamento. *eg.* `1, 2, 4, 8`
`hibernate.connection.<propertyName>`	Passa il nome di proprietà JDBC `propertyName` a `DriverManager.getConnection()`.
`hibernate.connection.provider_class`	Il nome della classe di un `ConnectionProvider` definito dall'utente. *e.g.* `nomediclasse.del.ConnectionProvider`
`hibernate.cache.provider_class`	Il nome di classe di un `CacheProvider` fornito dall'utente. *eg.* `nomediclasse.del.CacheProvider`
`hibernate.cache.use_minimal_puts`	Ottimizza le operazioni della cache di secondo livello in modo tale da minimizzare le scritture, al costo di letture più frequenti (usato per cache in cluster). *e.g.* `true|false`
`hibernate.cache.use_query_cache`	Attiva il caching delle interrogazioni, le query singole vanno comunque impostate come "cacheabili". *e.g.* `true|false`
`hibernate.cache.region_prefix`	Il prefisso da usare per i nomi delle regioni della cache di secondo livello. *eg.* `prefisso`

Nome della proprietà	Scopo
`hibernate.transaction.factory_class`	Il nome di classe di una `TransactionFactory` da usare con l'API `Transaction` di Hibernate (il valore predefinito è `JDBCTransactionFactory`). *eg.* `nomediclasse.della.TransactionFactory`
`jta.UserTransaction`	Il nome di classe usato da `JTATransactionFactory` per ottenere la `UserTransaction` JTA dall'application server. *eg.* `nome/composito/jndi`
`hibernate.transaction.manager_lookup_class`	Il nome di classe di un `TransactionManagerLookup` - richiesto quando il caching a livello di JVM è abilitato in un contesto JTA. *eg.* `nomediclasse.del.TransactionManagerLookup`
`hibernate.query.substitutions`	Mappaggio da alcune etichette nelle query di Hibernate a dei valori di sostituzione per le query SQL (potrebbero essere funzioni o letterali, ad esempio). *e.g.* `hqlLiteral=SQL_LITERAL, hqlFunction=SQLFUNC`
`hibernate.show_sql`	Scrive tutte le istruzioni SQL sulla console. *e.g.* `true` | `false`
`hibernate.hbm2ddl.auto`	Esporta automaticamente lo schema DDL sul database quando viene creata la `SessionFactory`. Con `create-drop`, lo schema di database verrà eliminato subito dopo che la `SessionFactory` verrà chiusa esplicitamente. *e.g.* `update` | `create` | `create-drop`

3.5.1. Dialetti SQL

Dovreste sempre impostare la proprietà `hibernate.dialect` al valore della sottoclasse di `net.sf.hibernate.dialect.Dialect` giusta per il vostro database. Questo non è strettamente necessario, a meno che desideriate usare la generazione di chiavi primaria `native` o `sequence` o il locking pessimistico (con, ad esempio, `Session.lock()` o `Query.setLockMode()`). Comunque, se specificate un dialetto, Hibernate imposterà dei default adatti per alcune delle proprietà elencate in precedenza, risparmiandovi lo sforzo di specificarle manualmente.

Tabella 3.4. Dialetti SQL di HIbernate (`hibernate.dialect`)

RDBMS	Dialetto
DB2	`net.sf.hibernate.dialect.DB2Dialect`
MySQL	`net.sf.hibernate.dialect.MySQLDialect`

RDBMS	Dialetto
SAP DB	`net.sf.hibernate.dialect.SAPDBDialect`
Oracle (any version)	`net.sf.hibernate.dialect.OracleDialect`
Oracle 9	`net.sf.hibernate.dialect.Oracle9Dialect`
Sybase	`net.sf.hibernate.dialect.SybaseDialect`
Sybase Anywhere	`net.sf.hibernate.dialect.SybaseAnywhereDialect`
Progress	`net.sf.hibernate.dialect.ProgressDialect`
Mckoi SQL	`net.sf.hibernate.dialect.MckoiDialect`
Interbase	`net.sf.hibernate.dialect.InterbaseDialect`
Pointbase	`net.sf.hibernate.dialect.PointbaseDialect`
PostgreSQL	`net.sf.hibernate.dialect.PostgreSQLDialect`
HypersonicSQL	`net.sf.hibernate.dialect.HSQLDialect`
Microsoft SQL Server	`net.sf.hibernate.dialect.SQLServerDialect`
Ingres	`net.sf.hibernate.dialect.IngresDialect`
Informix	`net.sf.hibernate.dialect.InformixDialect`
FrontBase	`net.sf.hibernate.dialect.FrontbaseDialect`

3.5.2. Reperimento via join esterno

Se il vostro database supporta i join esterni ANSI o Oracle, il *reperimento via join esterno* può aumentare le performance limitando il numero di accessi al database (al costo di un lavoro maggiore probabilmente effettuato dal database stesso). Il reperimento via join esterno consente ad un grafo di oggetti connessi da associazioni molti-a-uno, uno-a-molti o uno-a-uno di essere caricati in una singola `SELECT` SQL.

Per default, il grafo inizializzato quando si carica un oggetto termina agli oggetti foglia, alle collezioni, agli oggetti con mediatori (proxy) o dove ci siano delle circolarità.

Per una *associazione particolare*, il caricamento può essere abilitato o disabilitato (e quindi si può modificare il comportamento predefinito) impostando l'attributo `outer-join` nel mapping XML.

Il caricamento via join esterno può essere disabilitato *globalmente* impostando la proprietà `hibernate.max_fetch_depth` a `0`. Un settaggio di `1` o più abilita il join esterno per tutte le associazioni uno-a-uno e molti-a-uno che sono, sempre come impostazione predefinita, impostate ad `auto`. In ogni caso, le associazioni uno-a-molti e le collezioni non vengono mai caricate con un join esterno, a meno che questo non venga esplicitamente dichiarato per ogni particolare associazione. Anche questo comportamento può essere modificato a runtime con delle query di Hibernate.

3.5.3. Flussi (stream) binari

Oracle limita la dimensione degli array di `byte` che possono essere passati da o al suo driver JDBC. Se volete usare istanze di tipi `binari` o `serializzabili` di grandi dimensioni, dovete abilitare `hibernate.jdbc.use_streams_for_binary`. *Questa impostazione viene settata esclusivamente a livello di JVM.*

3.5.4. `CacheProvider` personalizzati

Potete integrare una cache di secondo livello che operi all'interno della virtual machine (JVM-level) o sull'intero cluster (clustered) implementando l'interfaccia `net.sf.hibernate.cache.CacheProvider`. Potete poi impostare l'implementazione personalizzata con l'opzione `hibernate.cache.provider_class`.

3.5.5. Configurazione della strategia transazionale

Se volete usare l'API `Transaction` di Hibernate, dovete specificare una classe factory ("fabbricatore" di istanze) per gli oggetti `Transaction` impostando la proprietà `hibernate.transaction.factory_class`. L'API `Transaction` maschera il meccanismo transazionale sottostante e consente al codice di Hibernate di eseguirsi in contesti gestiti dal container (managed) e non.

Ci sono due scelte possibili (pre-installate in Hibernate):

`net.sf.hibernate.transaction.JDBCTransactionFactory`
: delega al meccanismo transazionale JDBC transactions (impostazione predefinita)

`net.sf.hibernate.transaction.JTATransactionFactory`
: delega a JTA (se esiste una transazione attiva, la `Session` esegue il suo lavoro in quel contesto, altrimenti una nuova transazione viene attivata)

Potete anche definire le vostre strategie transazionali (per usare un servizio transazionale CORBA, ad esempio).

Se volete usare caching a livello di JVM per dati che siano modificabili in un contesto JTA, dovete specificare una strategia per ottenere il `TransactionManager` JTA, poiché non esiste un metodo standardizzato nei container J2EE per farlo:

Tabella 3.5. TransactionManager JTA

Transaction Factory	Application Server
`net.sf.hibernate.transaction.JBossTransactionManagerLookup`	JBoss
`net.sf.hibernate.transaction.WeblogicTransactionManagerLookup`	Weblogic
`net.sf.hibernate.transaction.WebSphereTransactionManagerLookup`	WebSphere
`net.sf.hibernate.transaction.OrionTransactionManagerLookup`	Orion
`net.sf.hibernate.transaction.ResinTransactionManagerLookup`	Resin
`net.sf.hibernate.transaction.JOTMTransactionManagerLookup`	JOTM
`net.sf.hibernate.transaction.JOnASTransactionManagerLookup`	JOnAS
`net.sf.hibernate.transaction.JRun4TransactionManagerLookup`	JRun4

3.5.6. `SessionFactory` pubblicata sul JNDI

Una `SessionFactory` di Hibernate pubblicata sul JNDI può semplificare il reperimento della factory stessa, e la creazione di nuove `Session`i.

Se volete che la `SessionFactory` venga pubblicata su uno spazio di nomi JNDI, specificate un nome (ad esem-

pio. `java:comp/env/hibernate/SessionFactory`) usando la proprietà `hibernate.session_factory_name`. Se questa proprietà viene omessa, la `SessionFactory` non verrà pubblicata sul JNDI. (Questo è particolarmente utile in ambienti in cui l'implementazione standard del JNDI sia di sola lettura, come ad esempio in Tomcat)

Quando Hibernate pubblicherà la `SessionFactory` sul JNDI, userà i valori di `hibernate.jndi.url` e `hibernate.jndi.class` per istanziare il contesto iniziale JNDI (InitialContext). Se queste proprietà non vengono specificate, verrà usato l'`InitialContext` predefinito.

Se scegliete di usare il JNDI, un EJB o qualsiasi altra classe di utilità può ottenere la `SessionFactory` con una ricerca (lookup) sull'albero JNDI.

3.5.7. Sostituzioni per il linguaggio di interrogazione

Potete definire nuove etichette per le interrogazioni di Hibernate usando la proprietà `hibernate.query.substitutions`. Ad esempio:

```
hibernate.query.substitutions true=1, false=0
```

farebbe sì che le etichette `true` e `false` venissero tradotti in letterali interi nell'SQL generato.

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

permetterebbe di rinominare la funzione `LOWER` dell'SQL.

3.6. Traccia di esecuzione (logging)

Hibernate traccia ("logs") vari eventi utilizzando la libreria commons-logging di Apache.

Il servizio di commons-logging indirizzerà l'uscita a Apache Log4j (se includete `log4j.jar` nel vostro classpath) o al logging nativo di JDK1.4 (se l'applicazione sta funzionando sotto JDK1.4 o superiori). Potete scaricare Log4j da `http://jakarta.apache.org`. Per usare Log4j avrete bisogno di un file `log4j.properties` sul classpath: un file di proprietà di esempio viene distribuito con Hibernate nella directory `src/`.

Raccomandiamo vivamente che vi familiarizziate con i messaggi di log di Hibernate. È stato fatto un grande sforzo per far sì che le tracce lasciate da Hibernate siano il più dettagliate possibile senza per questo renderle illeggibili. È uno strumento di risoluzione dei problemi fondamentale. Non dimenticate anche di abilitare la traccia dell'SQL come descritto in precedenza (`hibernate.show_sql`), perché è il primo passo quando si lavori alla risoluzione di problemi di performance.

3.7. Implementazione di una strategia di denominazione (`NamingStrategy`)

L'interfaccia `net.sf.hibernate.cfg.NamingStrategy` vi consente di specificare uno "standard di denominazione" per gli oggetti del database e gli elementi dello schema.

Potete fornire regole per generare automaticamente identificatori di database da identificatori java, o per ricavare nomi "fisici" di tabella e colonna dai nomi "logici" dati nel file di mappaggio. Questa funzionalità consente di ridurre la prolissità del documento di mappaggio, eliminando il rumore ripetitivo (come ad esempio i prefissi `TBL_`). La strategia base è abbastanza minimale.

Potete specificare una strategia differente chiamando `Configuration.setNamingStrategy()` prima di aggiun-

gere i mappaggi alla configurazione:

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

`net.sf.hibernate.cfg.ImprovedNamingStrategy` è una strategia che viene distribuita con Hibernate e che potrebbe essere un punto di partenza utile per alcune applicazioni.

3.8. File di configurazione XML

Un approccio alternativo è specificare una configurazione completa in un file chiamato `hibernate.cfg.xml`. Questo file può essere usato come un'alternativa al file `hibernate.properties` o, se sono presenti entrambi, per ridefinirne le proprietà.

Il file di configurazione XML viene caricato da Hibernate dalla radice del `CLASSPATH`. Ecco un esempio:

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 2.0//EN"

 "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">

<hibernate-configuration>

    <!-- una istanza di SessionFactory indicata con il suo /nome/jndi -->
    <session-factory
        name="java:comp/env/hibernate/SessionFactory">

        <!-- proprietà -->
        <property name="connection.datasource">my/first/datasource</property>
        <property name="dialect">net.sf.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">false</property>
        <property name="use_outer_join">true</property>
        <property name="transaction.factory_class">
            net.sf.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"/>

    </session-factory>

</hibernate-configuration>
```

La configurazione di Hibernate richiede solo di scrivere

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

È comunque possibile specificare un differente file di configurazione XML usando

```
SessionFactory sf = new Configuration()
    .configure("catdb.cfg.xml")
    .buildSessionFactory();
```

Capitolo 4. Le classi persistenti

Le classi persistenti sono quelle che in un'applicazione implementano le entità del problema di business (ad esempio Customer e Order in una applicazione di e-commerce). Le classi persistenti hanno, come implica il nome, istanze transienti ed istanze persistenti memorizzate nel database.

Hibernate funziona meglio se queste classi seguono alcune semplici regole, conosciute anche come il modello di programmazione dei "cari vecchi oggetti java" (in inglese e nel seguito si usa l'acronimo POJO che sta per "Plain Old Java Object").

4.1. Un semplice esempio POJO

La maggior parte delle applicazioni java richiede una classe persistente che rappresenti dei felini...

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

public class Cat {
    private Long id; // identificatore
    private String name;
    private Date birthdate;
    private Cat mate;
    private Set kittens
    private Color color;
    private char sex;
    private float weight;

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

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

    void setMate(Cat mate) {
        this.mate = mate;
    }
    public Cat getMate() {
        return mate;
    }

    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 setKittens(Set kittens) {
        this.kittens = kittens;
    }
    public Set getKittens() {
        return kittens;
    }
    // addKitten non è richiesto da Hibernate
    public void addKitten(Cat kitten) {
        kittens.add(kitten);
    }
    void setSex(char sex) {
        this.sex=sex;
    }
    public char getSex() {
        return sex;
    }
}
```

Ci sono quattro regole principali da seguire, qui:

4.1.1. Dichiarate metodi di accesso e di impostazione (get e set) per i campi persistenti

`Cat` dichiara metodi di accesso per tutti i suoi campi persistenti. Molti altri strumenti di mappaggio OR rendono direttamente persistenti le variabili di istanza. Crediamo che sia molto meglio disaccoppiare questo dettaglio implementativo dal meccanismo di persistenza. Hibernate rende persistenti le proprietà nello stile dei JavaBeans, e riconosce i nomi di metodo nella forma `getFoo`, `isFoo` e `setFoo`.

Non è necessario che le proprietà siano dichiarate "public" - Hibernate può rendere persistenti proprietà con coppie di metodi get/ set a visibilità default, `protected` o `private`.

4.1.2. Implementate un costruttore di default

`Cat` ha un costruttore di default (senza argomenti) implicito. Tutte le classi persistenti devono avere un costruttore di default (che può non essere pubblico), in modo tale che Hibernate possa costruirle usando `Constructor.newInstance()`.

4.1.3. Fornite una proprietà identificatore (opzionale)

`Cat` ha una proprietà chiamata `id`. Questa proprietà contiene il valore della chiave primaria di una tabella di database. La proprietà avrebbe potuto essere chiamata in un modo qualunque, e il suo tipo poteva essere un qualsiasi tipo primitivo, un "incapsulatore" ("wrapper") di tipi primitivi, `java.lang.String` o `java.util.Date`. (Se la vostra tabella di database preesistente ha chiavi composte, potete anche usare una classe definita da voi con le proprietà dei tipi delle colonne usate nella chiave - leggete la sezione sugli identificatori composti più avanti.)

La proprietà identificatore è opzionale. Potete farne a meno, e Hibernate terrà traccia internamente degli identificatori degli oggetti. In ogni caso, per molte applicazioni è comunque una buona (e molto popolare) decisione di progetto.

In più, alcune funzionalità sono possibili solo per classi che dichiarano una proprietà identificatore:

- Aggiornamenti a cascata (vedete "oggetti del ciclo di vita")
- `Session.saveOrUpdate()`

Vi raccomandiamo di dichiarare proprietà identificatore con nomi coerenti per le classi persistenti, e che usiate un tipo annullabile (cioè non primitivo).

4.1.4. Preferite classi non-final (opzionale)

Una funzionalità centrale di Hibernate, ovvero i *mediatori* ("proxy" in inglese), dipende dal fatto che la classe persistente sia non-final o che sia l'implementazione di un'interfaccia che dichiara tutti i suoi metodi pubblici.

Potete rendere persistenti classi `final` che non implementino un'interfaccia, con hibernate, ma non potrete usare i mediatori - cosa che limiterà in qualche modo le vostre opzioni per l'ottimizzazione delle prestazioni.

4.2. Utilizzo dell'ereditarietà

Anche una sottoclasse deve osservare la prima e la seconda regola. Eredita la sua proprietà di identificazione da `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. Implementate `equals()` e `hashCode()`

Dovete sovrascrivere i metodi `equals()` e `hashCode()` se volete mischiare oggetti di classi persistenti (ad esempio in un `Set`).

Questo vale solo se questi oggetti vengono caricati in due `Session`i differenti, poiché Hibernate garantisce l'uguaglianza degli oggetti (`a == b` , l'implementazione di default di `equals()`) solo all'interno di una singola `Session`!

Anche se entrambi gli oggetti `a` e `b` sono la stessa riga di database (hanno come identificatore lo stesso valore della chiave primaria), al di fuori del contesto di una particolare `Session` non possiamo garantire che siano la stessa istanza di oggetto Java.

La maniera più ovvia è di implementare `equals()`/`hashCode()` confrontando il valore di identificazione di entrambi gli oggetti. Se il valore è lo stesso, deve trattarsi della stessa riga di database, e quindi sono uguali (cioè se vengono entrambe aggiunte ad un `Set`, avremo solo un elemento al suo interno, dopo). Sfortunatamente, non possiamo usare questo approccio. Hibernate assegna valori di identificazione solo agli oggetti che sono persistenti, mentre una istanza appena creata non avrà alcun valore di identificatore! Quello che consigliamo, è di implementare `equals()` e `hashCode()` usando un concetto di *chiave di uguaglianza di business*.

"Chiave di uguaglianza di business" significa che il metodo `equals()` confronta solo le proprietà che formano la chiave di business, una chiave che identificherebbe la nostra istanza nel mondo reale (cioè una chiave candi-

data *naturale*):

```
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 (!getName().equals(cat.getName())) return false;
        if (!getBirthday().equals(cat.getBirthday())) return false;

        return true;
    }

    public int hashCode() {
        int result;
        result = getName().hashCode();
        result = 29 * result + getBirthday().hashCode();
        return result;
    }

}
```

Ricordatevi che la nostra chiave candidata (in questo caso si tratta della composizione di nome e data di nascita) deve essere valida solo per una particolare operazione di confronto (magari solo in un singolo caso d'uso). Non abbiamo bisogno dei parametri di stabilità che solitamente si applicano ad una vera chiave primaria!

4.4. Punti di richiamo del ciclo di vita degli oggetti ("lifecycle callbacks")

Una classe persistente può implementare in via opzionale l'interfaccia `Lifecycle` che fornisce alcuni punti di aggancio che consentono all'oggetto persistente di effettuare operazioni di inizializzazione/pulizia dopo un salvataggio o un caricamento, e prima di una cancellazione o un aggiornamento.

La classe `Interceptor` offre comunque una alternativa meno instrusiva, comunque.

```
public interface Lifecycle {
        public boolean onSave(Session s) throws CallbackException;   (1)
        public boolean onUpdate(Session s) throws CallbackException; (2)
        public boolean onDelete(Session s) throws CallbackException; (3)
        public void onLoad(Session s, Serializable id);              (4)
}
```

(1) `onSave` - chiamato subito prima che l'oggetto venga salvato o inserito

(2) `onUpdate` - chiamato subito prima che un oggetto venga aggiornato (quando viene passato a `Session.update()`)

(3) `onDelete` - chiamato subito prima che un oggetto venga cancellato

(4) `onLoad` - chiamato subito dopo che un oggetto è caricatocalled just after an object is loaded

`onSave()`, `onDelete()` e `onUpdate()` possono essere usati per propagare salvataggi e cancellazioni degli oggetti dipendenti. È un'alternativa alla dichiarazione di operazioni di cascata nel file di mappaggio. `onLoad()` può essere usato per inizializzare proprietà dell'oggetto dal suo stato persistente. Non può essere usato per caricare oggetti dipendenti poiché l'interfaccia `Session` non può venire chiamata dall'interno del metodo. L'utilizzo ulteriore di `onLoad()`, `onSave()` e `onUpdate()` è per memorizzare un riferimento alla `Session` corrente per utilizzi successivi.

Notate che `onUpdate()` non viene chiamato ogni volta che lo stato persistente dell'oggetto viene modificato, ma solo quando l'oggetto transiente viene passato a `Session.update()`.

Se `onSave()`, `onUpdate()` o `onDelete()` restituiscono `true`, l'operazione viene silenziosamente impedita. Se viene lanciata una `CallbackException`, l'operazione è proibita e l'eccezione viene restituita all'applicazione.

Notate che `onSave()` viene chiamata dopo che un identificatore sia assegnato all'oggetto, eccetto quando viene usata la strategia di generazione di chiavi nativa.

4.5. Punto di aggancio (callback) Validatable

Se la classe persistente ha bisogno di controllare degli invarianti prima che il suo stato sia reso persistente, può implementare l'interfaccia seguente:

```
public interface Validatable {
        public void validate() throws ValidationFailure;
}
```

L'oggetto dovrebbe lanciare una `ValidationFailure` se è stato violato qualche invariante. Un'istanza di `Validatable` non dovrebbe però cambiare il suo stato, dall'interno di `validate()`.

A differenza dei metodi di richiamo dell'interfaccia `Lifecycle`, `validate()` potrebbe venire chiamata in momenti imprevisti. L'applicazione non dovrebbe affidarsi alle chiamate a `validate()` per implementare funzionalità di business.

4.6. Utilizzo del contrassegno (markup) di XDoclet

Nel prossimo capitolo mostreremo come i mappaggi di Hibernate possano venire espressi usando un formato XML semplice e leggibile. Molti utenti di Hibernate preferiscono inserire l'informazione di mappaggio direttamente nel codice sorgente usando gli `@hibernate.tags` di XDoclet. Non parleremo qui di questo approccio poiché viene considerato strettamente parte di XDoclet. Tuttavia includiamo l'esempio seguente della classe `Cat` con i mappaggi di 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 mate;
    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="MATE_ID"
     */
    public Cat getMate() {
        return mate;
    }
    void setMate(Cat mate) {
        this.mate = mate;
    }

    /**
     * @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
     *  lazy="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;
    }
}
```

Capitolo 5. Mappaggio O/R di base

5.1. Dichiarazione dei mappaggi

I mappaggi oggetto/relazione vengono definiti in un documento XML. Il documento di mappaggio è progettato per essere leggibile e modificabile a mano. Il linguaggio di mappaggio è java-centrico, nel senso che i mappaggi sono costruiti intorno alle dichiarazioni delle classi persistenti, non sulle dichiarazioni delle tabelle.

Notate che anche se molti utenti di Hibernate scelgono di definire i mappaggi XML a mano, esistono un certo numero di strumenti per generare il documento di mappaggio, tra cui XDoclet, Middlegen e AndroMDA.

Ora cominciamo con un mappaggio di esempio:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping package="eg">

        <class name="Cat" table="CATS" discriminator-value="C">
                <id name="id" column="uid" type="long">
                        <generator class="hilo"/>
                </id>
                <discriminator column="subclass" type="character"/>
                <property name="birthdate" type="date"/>
                <property name="color" not-null="true"/>
                <property name="sex" not-null="true" update="false"/>
                <property name="weight"/>
                <many-to-one name="mate" column="mate_id"/>
                <set name="kittens">
                        <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">
                <!-- qui potrebbe stare il mappaggio per Dog -->
        </class>

</hibernate-mapping>
```

Ora discuteremo il contenuto del documento di mappaggio. Descriveremo solo gli elementi e gli attributi del documento che Hibernate usa in fase di esecuzione. Il documento di mappaggio contiene anche alcuni elementi ed attributi opzionali che hanno effetto sugli schemi del database exportati dallo strumento di generazione dello schema (schemaexport). (Ad esempio l'attributo `not-null`.)

5.1.1. Doctype

Tutti i mappaggi XML dovrebbero dichiarare il che abbiamo mostrato nell'esempio. L'effettivo DTD può essere trovato all'URL indicato, nella directory `hibernate-x.x.x/src/net/sf/hibernate` del pacchetto di Hibernate o nel file `hibernate.jar`. Hibernate cercherà sempre per prima cosa il DTD sul classpath.

5.1.2. hibernate-mapping

Questo elemento ha tre attributi opzionali. L'attributo `schema` specifica che le tabelle a cui si fa riferimento nel mappaggio appartengono allo schema indicato. Se viene usato, i nomi delle tabelle saranno completati dal nome dello schema inidicato. Se manca, i nomi delle tabelle non saranno ulteriormente caratterizzati. L'attributo `default-cascade` specifica quale stile di cascata dovrebbe essere assunto per le proprietà e le collezioni che non specificano un attributo `cascade` attribute. L'attributo `auto-import` ci consente di utilizzare nomi di classe non qualificati nel linguaggio di interrogazione come comportamento predefinito.

```
<hibernate-mapping
        schema="schemaName"                         (1)
        default-cascade="none|save-update"          (2)
        auto-import="true|false"                    (3)
        package="package.name"                      (4)
 />
```

(1) `schema` (opzionale): Il nome di uno schema del database.
(2) `default-cascade` (opzionale - il default è `none`): Uno stile di cascata predefinito.
(3) `auto-import` (opzionale - il default è `true`): Specifica se possiamo usare nomi di classe non qualificati (le classi devono essere di questo mappaggio) nel linguaggio di interrogazione.
(4) `package` (opzionale): Specifica un prefisso di package da assumere per i nomi di classi non qualificati nel documento di mappaggio.

Se avete due classi persistenti con lo stesso nome (non qualificato), dovreste impostare `auto-import="false"`. Hibernate lancerà un'eccezione se tentate di assegnare due classi diversse allo stesso nome "importato".

5.1.3. class

L'elemento `class` si usa per dichiarare una classe persistente:

```
<class
        name="ClassName"                            (1)
        table="tableName"                           (2)
        discriminator-value="discriminator_value"   (3)
        mutable="true|false"                        (4)
        schema="owner"                              (5)
        proxy="ProxyInterface"                      (6)
        dynamic-update="true|false"                 (7)
        dynamic-insert="true|false"                 (8)
        select-before-update="true|false"           (9)
        polymorphism="implicit|explicit"            (10)
        where="arbitrary sql where condition"       (11)
        persister="PersisterClass"                  (12)
        batch-size="N"                              (13)
        optimistic-lock="none|version|dirty|all"    (14)
        lazy="true|false"                           (15)
/>
```

(1) `name`: il nome di classe java completamente qualificato della classe persistente (o l'interfaccia).
(2) `table`: il nome della sua tabella di database.
(3) `discriminator-value` (opzionale - il default è il nome della classe): un valore che distingue sottoclassi individuali, usato per il comportamento polimorfico. I valori accettabili includono `null` e `not null`.
(4) `mutable` (opzionale, il default è `true`): specifica che le istanze della classe (non) sono mutabili.
(5) `schema` (opzionale): sovrascrive il nome dello schema specificato dall'elemento radice `<hibernate-mapping>`.
(6) `proxy` (opzionale): specifica una interfaccia da usare per i mediatori (proxy) ad inizializzazione ritardata. Potete specificare il nome della classe stessa.
(7) `dynamic-update` (opzionale, il default è `false`): specifica che una `UPDATE` SQL dovrebbe venire generata in fase di esecuzione e contenere solo i nomi delle colonne di cui sono cambiati i valori.

(8) `dynamic-insert` (opzionale, il default è `false`): specifica che le `INSERT` SQL dovrebbero venire generate in fase di esecuzione e contenere solo i nomi delle colonne i cui valori sono non nulli.

(9) `select-before-update` (opzionale, il default è `false`): specifica che Hibernate non dovrebbe *mai* eseguire una `UPDATE` a meno che non sia certo che un oggetto non sia davvero stato modificato. In certi casi (in realtà solo quando un oggetto transiente sia stato associato ad una nuova sessione usando `update()`), questo significa che Hibernate effettuerà una istruzione SQL `SELECT` in più per determinare se `UPDATE` sia realmente richiesto.

(10) `polymorphism` (opzionale, il default è `implicit`): determina se deve essere usato un polimorfismo di interrogazione implicito o esplicito.

(11) `where` (opzionale) specifica una condizione `WHERE` dell'SQL arbitraria da usare quando si recuperano oggetti di questa classe

(12) `persister` (opzionale): specifica un `ClassPersister` personalizzato.

(13) `batch-size` (opzionale, il default è `1`) specifica una "dimensione di blocco" (batch) per il caricamento di istanze di questa classe per identificatore.

(14) `optimistic-lock` (opzionale, il default è `version`): Determina la strategia di locking ottimistico.

(15) `lazy` (opzionale): impostare `lazy="true"` è una scorciatoia equivalente a specificare il nome stesso della classe come interfaccia `proxy`.

È perfettamente accettabile che il nome della classe persistente sia un'interfaccia. In questo caso si dichiarano le classi di implementazione di quell'interfaccia utilizzando l'elemento `<subclass>`. Potete persistere anche classi interne *static*. In questo caso dovete specificare il nome della classe usando la forma standard , cioè `eg.Foo$Bar`.

Le classi immutabili, `mutable="false"` non possono essere aggiornate o cancellate dall'applicazione. Questo consente ad Hibernate di effettuare alcune ottimizzazioni di performance minori.

L'attributo `proxy` opzionale consente l'inizializzazione ritardata delle istanze persistenti della classe. Hibernate inizialmente restituirà dei mediatori (proxy) CGLIB che implementano l'interfaccia indicata. Il vero oggetto persistente sarà caricato quando si invocherà un metodo del mediatore. Leggete più oltre il paragrafo "Mediatori per l'inizializzazione ritardata".

Il polimorfismo *implicito* significa che interrogazioni che indicheranno i nomi di una qualsiasi superclasse o interfaccia implementata da una classe potranno restituire istanze di quella classe stessa, e che una query che indichi il nome della classe stessa potrà restituire anche istanze di una qualsiasi sottoclasse. Il polimorfismo *Explicit* significa che le istanze di una classe verranno restituite esclusivamente da interrogazioni che indichino esplicitamente il nome di quella classe, e che interrogazioni che indichino il nome di quella classe restituiranno esclusivamente nomi di sottoclassi mappati all'interno di questa dichiarazione `<class>` come `<subclass>` o `<joined-subclass>`. Per la maggior parte degli scopi, l'impostazione predefinita, ovvero `polymorphism="implicit"`, è appropriata. Il polimorfismo esplicito è utile quando due classi diverse vengano mappate sulla stessa tabella (questo consente di avere una classe "leggera" che contiene un sottoinsieme delle colonne della tabella).

L'attributo `persister` vi consente di personalizzare la strategia di persistenza utilizzata per la classe. potete, ad esempio, specificare la vostra sottoclasse di `net.sf.hibernate.persister.EntityPersister` o potete addirittura fornire una implementazione completamente diversa dell'interfaccia `net.sf.hibernate.persister.ClassPersister` che implementi la persistenza via, ad esempio, chiamate a procedure memorizzate (stored procedure), serializzazione su file piatti o LDAP. Andate a vedere il codice di `net.sf.hibernate.test.CustomPersister` per un esempio semplice (di "persistenza" su una `Hashtable`).

Notate che le impostazioni `dynamic-update` e `dynamic-insert` non vengono ereditate dalle sottoclassi, e quindi potrebbero venire anche specificate sugli elementi `<subclass>` o `<joined-subclass>`. Queste impostazioni possono aumentare le performance, in certi casi, ma potrebbero in realtà diminuire le performance in altri. Usatele con giudizio.

L'uso di `select-before-update` di solito diminuirà le performance. È molto utile però per evitare che dei trigger sul database associati all'update vengano chiamati inutilmente.

Se abilitate `dynamic-update`, avrete una scelta fra strategie di locking ottimistico:

- `version` controlla le colonne di versione/marca di tempo
- `all` controlla tutte le colonne
- `dirty` controlla le colonne cambiate
- `none` non usa il locking ottimistico

Raccomandiamo *molto* che usiate le colonne di versione/marca di tempo per il locking ottimistico con Hibernate. Si tratta della strategia ottimale rispetto alle performance, ed è la sola strategia che gestisca correttamente le modifiche fatte al di fuori della sessione, (ad esempio quando venga usato `Session.update()`). Ricordatevi che una proprietà di versione o marca di tempo non dovrebbe mai essere nulla, indipendentemente da quale sia la strategia `unsaved-value`, o un'istanza verrà individuata come transiente.

5.1.4. id

Le classi mappate *devono* dichiarare la colonna di chiave primaria della tabella sul database. La maggior parte delle classi avrà anche una proprietà nello stile dei javabean (cioè con metodi "getter" e "setter") che manterrà l'identificatore unico di un'istanza. L'elemento `<id>` definisce il mappaggio da quella proprietà alla colonna di chiave primaria.

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

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

(1) `name` (opzionale): il nome della proprietà identificatore.
(2) `type` (opzionale): un nome che indica il tipo di Hibernate.
(3) `column` (opzionale - il default è il nome della proprietà): il nome della colonna di chiave primaria.
(4) `unsaved-value` (opzionale - il default è `null`): un valore di proprietà di identificazione che indichi che un'istanza è appena stata istanziata (è "unsaved"), distinguendola da istanze transienti che siano state salvate o caricate in una sessione precedente.
(5) `access` (opzionale - il default è `property`): la strategia che Hibernate dovrebbe usare per accedere al valore della proprietà.

Se l'attributo `name` manca, si assume che la classe non abbia proprietà identificatore.

L'attributo `unsaved-value` è importante! Se la proprietà identificatore della vostra classe non ha `null` come valore iniziale, allora dovreste specificare il valore.

C'è una dichiarazione alternativa, `<composite-id>`, per consentire accesso a dati preesistenti con chiavi composite. Scoraggiamo fortemente il suo uso per qualsiasi altro motivo.

5.1.4.1. generator

L'elemento figlio obbligatorio `<generator>` indica una classe Java utilizzata per generare identificatori unici per istanze di questa classe persistente. Se l'istanza del generatore richiedesse di essere configurata o inizializzata con dei parametri, questi possono essere passati usando l'elemento `<param>`.

```
<id name="id" type="long" column="uid" unsaved-value="0">
        <generator class="net.sf.hibernate.id.TableHiLoGenerator">
                <param name="table">uid_table</param>
                <param name="column">next_hi_value_column</param>
        </generator>
</id>
```

Tutti i generatori implementano l'interfaccia `net.sf.hibernate.id.IdentifierGenerator`. È un'interfaccia molto semplice; alcune applicazioni potrebbero scegliere di fornire le loro implementazioni specializzate. In ogni caso, Hibernate fornisce un certo numero di implementazioni preinstallate. Ci sono anche dei nomi abbreviati per i generatori preinstallati:

`increment`
: genera identificatori di tipo `long`, `short` o `int` che sono unici solo quando nessun altro processo inserisce dati nella stessa tabella. *Da non usare in un cluster.*

`identity`
: supporta le colonne "identity" in DB2, MySQL, MS SQL Server, Sybase e HypersonicSQL. L'identificatore restituito è di tipo `long`, `short` o `int`.

`sequence`
: usa una "sequence" in DB2, PostgreSQL, Oracle, SAP DB, McKoi o un generatore in Interbase. L'identificatore restituito è di tipo `long`, `short` o `int`.

`hilo`
: usa un algoritmo hi/lo per generare efficientemente identificatori di tipo `long`, `short` o `int`, date una tabella e una colonna (per default `hibernate_unique_key` e `next` rispettivamente) come sorgente di valori "hi". L'algoritmo hi/lo genera identificatori che sono unici solo per un particolare database. *Non usate questo generatore con connessioni iscritte con JTA o con una connessione fornita da voi stessi.*

`seqhilo`
: usa un algoritmo hi/lo per generare efficientemente identificatori di tipo `long`, `short` o `int`, dato il nome di una sequenza sul database.

`uuid.hex`
: usa un algoritmo UUID a 128-bit per generare identificatori di tipo stringa, unici all'interno di una rete (viene usato l'indirizzo IP). L'UUID è codificato come una stringa di 32 caratteri esadecimali.

`uuid.string`
: usa lo stesso algoritmo UUID. L'UUID è codificato come una stringa di lunghezza 16 che consiste di (qualsiasi) carattere ASCII. *Non usare con PostgreSQL.*

`native`
: usa `identity`, `sequence` o `hilo` a seconda delle capacità del database sottostante.

`assigned`
: lascia all'applicazione il compito di assegnare un identificatore all'oggetto prima che venga chiamato `save()`.

`foreign`
: usa l'identificatore di un altro oggetto associato. Solitamente è usato insieme a una associazione di chiave

primaria <one-to-one>.

5.1.4.2. Algoritmo Hi/Lo

I generatori hilo e seqhilo forniscono due implementazioni alternative dell'algoritmo hi/lo, un approccio importante per la generazione di identificatori. La prima implementazione richiede una tabella "speciale" del database per mantenere il prossimo valore "hi" disponibile. La seconda usa una "sequence" nello stile di Oracle (dove sia supportata).

```
<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>
```

Sfortunatamente non è possibile usare hilo quando fornite le vostre Connection a Hibernate, o quando Hibernate sta usando il datasource di un application server per ottenere connessioni iscritte con il JTA. Hibernate deve essere in grado di raccogliere il valore "hi" in una nuova transazione. Un approccio standard in un ambiente EJB è di implementare l'algoritmo hi/lo usando un session bean senza stato.

5.1.4.3. Algoritmo UUID

Gli UUIDs contengono: indirizzo IP, tempo di partenza della JVM (accurato al quarto di secondo), il tempo di sistema e il valore di un contatore (unico all'interno della JVM). Non è possibile ottenere un indirizzo MAC o un indirizzo di memoria da del codice java, quindi questo è il massimo che possiamo fare senza usare JNI.

Non tentate di usare uuid.string in PostgreSQL.

5.1.4.4. Colonne "Identity" e "Sequence"

Per i database che supportano le colonne "identity" (DB2, MySQL, Sybase, MS SQL), potete usare la generazione di chiave identity. Per i database che supportano le sequenze (DB2, Oracle, PostgreSQL, Interbase, McKoi, SAP DB) potete usare la generazione di chiave nello stile sequence. Entrambe queste strategie richiedono due istruzioni SQL per inserire un nuovo oggetto.

```
<id name="id" type="long" column="uid">
        <generator class="sequence">
                <param name="sequence">uid_sequence</param>
        </generator>
</id>
```

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

Per lo sviluppo cross-piattaforma, la strategia native sceglierà dalle strategieidentity, sequence e hilo, in maniera dipendente dalle capacità del database sottostante.

5.1.4.5. Identificatori assegnati

Se volete che sia l'applicazione ad assegnare gli identificatori (rispetto ad una situazione in cui è Hibernate che li genera), potete usare il generatore `assigned`. Questo generatore speciale userà il valore di identificatore già assegnato alla proprietà di identificazione dell'oeggetto. State molto attenti quando usate questa funzionalità a non assegnare chiavi con significato di business (quasi sempre una terribile decisione di design).

A causa della sua natura implicita, le entità che usano questo generatore non possono essere salvate usando il metodo saveOrUpdate() della Session. Invece dovete specificare esplicitamente ad Hibernate se l'oggetto dovrebbe essere salvato o aggiornato chiamando o il metodo `save()` o il metodo `update()` della Session.

5.1.5. composite-id

```
<composite-id
        name="propertyName"
        class="ClassName"
        unsaved-value="any|none"
        access="field|property|ClassName">

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

Per una tabella con una chiave composita, potete mappare proprietà multiple della classe come proprietà identificatore. L'elemento `<composite-id>` accetta mappaggi di proprietà `<key-property>` e `<key-many-to-one>` come elementi figli.

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

La vostra classe persistente *deve* sovrascrivere `equals()` e `hashCode()` per implementare l'uguaglianza degli identificatori composti. Deve anche implementare `Serializable`.

Sfortunatamente, questo approccio agli identificatori compositi significa che un oggetto persistente è il suo proprio identificatore. Non c'è un "handle" conveniente al di là dell'oggetto stesso. Dovete istanziare un oggetto della classe persistente e popolare le sue proprietà di identificazione, prima che possiate caricare (`load()`) lo stato persistente associato ad una classe composita. Descriveremo un approccio molto più conveniente in cui l'identificatore composito sarà implementato come una classe separata nel paragrafo Sezione 7.4, “Componenti come identificatori composti”. Gli attributi descritti sotto si applicano solo a questo approccio alternativo:

- `name` (opzionale): una proprietà di un tipo di componente che mantiene l'identificatore composito (vedete la prossima sezione).
- `class` (opzionale - il default è il tipo di proprietà ricavato via "reflection"): la classe di componente usata come identificatore composito (vedere la prossima sezione).
- `unsaved-value` (opzionale - il default è `none`): indica che le istanze transienti dovrebbero essere considerate appena istanziate, se impostato ad `any`.

5.1.6. discriminatori

L'elemento `<discriminator>` è richiesto per la persistenza polimorfica quando si usa la strategia di mappaggio "tabella per gerarchia di classi" e dichiara una colonna discriminatore della tabella. La colonna discriminatore contiene valori di indicazione che informano lo strato persistente riguardo alla particolare sottoclasse da istanziare per una riga. Si può usare solo un insieme ristretto di tipi: `string`, `character`, `integer`, `byte`, `short`, `boolean`, `yes_no`, `true_false`.

```
<discriminator
        column="discriminator_column"  (1)
        type="discriminator_type"      (2)
        force="true|false"             (3)
/>
```

(1) `column` (opzionale - il default è `class`) il nome della colonna discriminatore.
(2) `type` (opzionale - il default è `string`) il nome che indica il tipo di Hibernate
(3) `force` (opzionale - il default è `false`) "forza" Hibernate a specificare valori consentiti del discriminatore anche quando si stanno recuperando tutte le istanze della classe radice.

I valori effettivi della colonna discriminatore sono specificati dall'attributo `discriminator-value` degli elementi `<class>` e `<subclass>`.

L'attributo `force` è utile (solo) se la tabella contiene righe con valori di discriminatore "extra" che non sono mappati su una classe persistente. Questo solitamente non è il caso.

5.1.7. versione (opzionale)

L'elemento `<version>` è opzionale, e indica che la tabella contiene dati versionati. È particolarmente utile se progettate di usare *transazioni lunghe* (vedete oltre).

```
<version
        column="version_column"                         (1)
        name="propertyName"                             (2)
        type="typename"                                 (3)
        access="field|property|ClassName"               (4)
        unsaved-value="null|negative|undefined"         (5)
/>
```

(1) `column` (opzionale - il default è il nome di proprietà): il nome della colonna che mantiene il numero di versione.
(2) `name`: il nome di una proprietà della classe persistente.
(3) `type` (opzionale - il default è `integer`): il tipo del numero di versione.
(4) `access` (opzionale - il default è `property`): la strategia che Hibernate deve usare per accedere al valore della proprietà.
(5) `unsaved-value` (opzionale - il default è `undefined`): il valore di una proprietà di versione che indica che un'istanza è appena stata istanziata (è "unsaved"), distinguendola da istanze transienti che erano state salvate o caricate in una sessione precedente. (`undefined` specifica che bisogna usare il valore della proprietà identificatore.)

I numeri di versione possono essere di tipo `long`, `integer`, `short`, `timestamp` o `calendar`.

5.1.8. timestamp (opzionale)

L'elemento opzionale `<timestamp>` indica che la tabella contiene dati con marche di tempo. Si intende come un'alternativa al versionamento. Le marche di tempo sono per natura un'implementazione meno sicura del locking ottimistico. In ogni caso, a volte l'applicazione potrebbe usare le marche di tempo in altri modi.

```
<timestamp
        column="timestamp_column"           (1)
        name="propertyName"                 (2)
        access="field|property|ClassName"   (3)
        unsaved-value="null|undefined"      (4)
/>
```

(1) `column` (opzionale - il default è il nome di proprietà): il nome di una colonna che contiene la marca di tempo.
(2) `name`: il nome di una proprietà in stile JavaBeans del tipo java `Date` o `Timestamp` della classe persistente.
(3) `access` (opzionale - il default è `property`): la strategia che Hibernate dovrebbe usare per accedere ai valori delle proprietà.
(4) `unsaved-value` (opzionale - il default è `null`): il valore di una proprietà di versione che indica che l'istanza è appena stata istanziata (è "unsaved"), distinguendola dalle istanze transienti che sono state savlate o caricate in una sessione precedente. (`undefined` specifica che bisogna usare il valore della proprietà identificatore.)

Notate che `<timestamp>` è equivalente a `<version type="timestamp">`.

5.1.9. property

L'elemento `<property>` dichiara una proprietà persistente in stile JavaBeans della classe.

```
<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)
/>
```

(1) `name`: il nome della proprietà con iniziale minuscola.
(2) `column` (opzionale - il default è usare il nome della proprietà): il nome della colonna mappata della tabella del database.
(3) `type` (opzionale): il nome che indica il tipo di Hibernate.
(4) `update, insert` (opzionale - il default è `true`): specifica che le colonne mappate dovrebbero essere incluse in istruzioni SQL `UPDATE` e/o `INSERT`. Impostare entrambe a `false` consente una proprietà puramente "derivata" il cui valore è inizializzato da qualche altra proprietà che si mappa sulla stessa colonna (o colonne), o da un trigger o da un'altra applicazione.
(5) `formula` (opzionale): una espressione SQL che definisce il valore per una proprietà *calcolata*. Le proprietà calcolate non hanno un mappaggio di colonna proprio.
(6) `access` (opzionale - il default è `property`): la strategia che Hibernate deve usare per accedere al valore della proprietà.

typename potrebbe essere:

1. Il nome di un tipo di base di Hibernate (ad esempio. `integer, string, character, date, timestamp, float, binary, serializable, object, blob`).
2. Il nome di una classe Java con un tipo di default base (e.g. `int, float, char, java.lang.String, java.util.Date, java.lang.Integer, java.sql.Clob`).
3. Il nome di una sottoclasse di `PersistentEnum` (e.g. `eg.Color`).
4. Il nome di una classe java serializzabile.
5. Il nome della classe di un tipo personalizzato (e.g. `com.illflow.type.MyCustomType`).

Se non specificate un tipo, Hibernate userà la "reflection" sul nome della proprietà per indovinare il tipo di Hibernate corretto. Hibernate cercherà di interpretare il nome della classe di ritorno del metodo recuperatore ("getter") usando le regole 2, 3 e 4 in questo ordine. Però, questo non è sempre abbastanza. In certi casi, avrete comunque bisogno dell'attributo `type`. (Ad esempio, per distinguere tra `Hibernate.DATE` e `Hibernate.TIMESTAMP`, o per specificare un tipo personalizzato.)

L'attributo access vi consente di controllare come Hibernate accederà al valore dell'attributo in fase di esecuzione. Il comportamento predefinito di Hibernate è di chiamare la coppia get/set della proprietà. Se però specificate access="field", Hibernate aggirerà la coppia get/set ed accederà direttamente al campo utilizzando la "reflection". Potete anche specificare la vostra strategia per l'accesso alle proprietà indicando una classe che implementi l'interfaccia net.sf.hibernate.property.PropertyAccessor.

5.1.10. many-to-one

Un'associazione ordinaria ad un'altra classe persistente si dichiara usando un elemento many-to-one. Il modello relazionale è un'associazione molti-a-uno. (In realtà si tratta semplicemente di un riferimento ad oggetto.)

```
<many-to-one
        name="propertyName"                                 (1)
        column="column_name"                                (2)
        class="ClassName"                                   (3)
        cascade="all|none|save-update|delete"               (4)
        outer-join="true|false|auto"                        (5)
        update="true|false"                                 (6)
        insert="true|false"                                 (6)
        property-ref="propertyNameFromAssociatedClass"      (7)
        access="field|property|ClassName"                   (8)
/>
```

(1) name: il nome della proprietà.
(2) column (opzionale): il nome della colonna.
(3) class (opzionale - il default è il tipo della proprietà determinato per "reflection"): il nome della classe associata.
(4) cascade (opzionale): specifica quali operazioni dovrebbero andare in cascata dall'oggetto genitore all'oggetto associato.
(5) outer-join (opzionale - il default è auto): consente la raccolta via outer-join per questa associazione se è impostata la proprietà hibernate.use_outer_join.
(6) update, insert (opzionale - il default è true) specifica che la colonna mappata dovrebbe venire inclusa nelle istruzioni SQL UPDATE e/o INSERT. Impostare entrambe a false consente di avere una associazione puramente "derivata" il cui valore è inizializzato da qualche altra proprietà che si mappi sulla stessa colonna o da un trigger o da un'altra applicazione.
(7) property-ref: (opzionale) il nome di una proprietà della classe associata che è messa in join a questa chiave esterna. Se non viene specificata, si usa la chiave primaria della classe associata.
(8) access (opzionale - il default è property): la strategia che Hibernate deve usare per accedere al valore di questa proprietà.

L'attributo cascade accetta i valori seguenti: all, save-update, delete, none. Impostare un valore diverso da none farà sì che certe operazioni si propaghino sull'oggetto associato (figlio). Vedete anche "oggetti a ciclo di vita" più oltre.

L'attributo outer-join accetta tre valori differenti:

- auto (default) recupera l'associazione utilizzando un join esterno se la classe associata non ha proxy
- true carica sempre l'associazione con un join esterno
- false non carica mai l'associazione con un outer join

Una tipica dichiarazione many-to-one appare così semplice:

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

L'attributo property-ref dovrebbe venire usato solo per mappare dati preesistenti in cui una chiave esterna

faccia riferimento ad una chiave unica della tabella associata che sia diversa dalla chiave primaria. Si tratta di un modello relazionale decisamente orrido. Ad esempio, immaginate che la classe `Product` abbia un numero di serie unico che non sia la chiave primaria. (L'attributo `unique` controlla la generazione del DDL da parte di Hibernate con il tool SchemaExport.)

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

A questo punto il mappaggio per `OrderItem` potrebbe usare:

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

In ogni modo questo è di certo non incoraggiato.

5.1.11. one-to-one

Una associazione uno-a-uno con un'altra classe persistente si dichiara usando un elemento `one-to-one` .

```
<one-to-one
        name="propertyName"                                 (1)
        class="ClassName"                                   (2)
        cascade="all|none|save-update|delete"               (3)
        constrained="true|false"                            (4)
        outer-join="true|false|auto"                        (5)
        property-ref="propertyNameFromAssociatedClass"      (6)
        access="field|property|ClassName"                   (7)
/>
```

(1) `name`: il nome della proprietà.

(2) `class` (opzionale - il default è il tipo della proprietà determinato per "reflection"): il nome della classe associata.

(3) `cascade` (opzionale) specifica quali operazioni dovrebbero propagarsi in cascata dall'oggetto genitore all'oggetto associato.

(4) `constrained` (opzionale) specifica che un vincolo di chiave esterna sulla chiave primaria della tabella mappata fa riferimento alla tabella della classe associata. Questa opzione condiziona l'ordine in cui `save()` e `delete()` vengono propagate (ed è anche usata dallo strumento di generazione dello schema).

(5) `outer-join` (opzionale - il default è `auto`): consente la raccolta via join esterno per questa associazione quando viene impostato `hibernate.use_outer_join`.

(6) `property-ref`: (opzionale) il nome di una proprietà della classe associata che è messa in join alla chiave primaria di questa classe. Se non viene specificata, viene usata la chiave primaria della classe associata.

(7) `access` (opzionale - il default è `property`): la strategia che Hibernate dovrebbe usare per accedere al valore della proprietà.

Ci sono due varietà di associazioni uno-a-uno:

- associazioni di chiave primaria
- associazioni di chiave esterna univoca

Le associazioni di chiave primaria non hanno bisogno di una colonna extra nella tabella; se due righe sono messe in relazione dall'associazione, allora le due righe condividono lo stesso valore di chiave primaria. Per questo, se volete che due oggetti siano correlati da un'associazion di chiave primaria, dovete assicurarvi che venga loro assegnato lo stesso valore di identificatore!

Per un'associazione di chiave primaria, si aggiungono i mappaggi seguenti rispettivamente a `Employee` e `Person`.

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

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

Ora ci dobbiamo assicurare che le chiavi primarie delle righe correlate nelle tabelle PERSON e EMPLOYEE siano uguali. Usiamo una strategia di generazione di identificatore speciale di Hibernate, chiamata `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>
```

Ad un'istanza appena salvata di `Person` si assegna poi lo stesso valore di chiave primaria dell'istanza di `Employee` a cui fa riferimento la proprietà `employee` di quella `Person`.

In alternativa, una chiave esterna con un vincolo di unicità da `Employee` a `Person` può essere espressa come:

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

E questa associazione può essere resa bidirezionale aggiungendo quanto segue al mappaggio di `Person`:

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

5.1.12. component, dynamic-component

L'elemento `<component>` mappa proprietà di un oggetto figlio su colonne della tabella di una classe genitore. I componenti possono, a loro volta, dichiarare le proprie proprietà, componenti o collezioni. Vedete "Componenti" più oltre.

```
<component
        name="propertyName"                 (1)
        class="className"                   (2)
        insert="true|false"                 (3)
        upate="true|false"                  (4)
        access="field|property|ClassName">  (5)

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

(1) `name`: il nome della proprietà.

(2) `class` (opzionale - il default è il tipo della proprietà individuato per "reflection"): il nome della classe componente (figlio).

(3) `insert`: le colonne mappate devono apparire nelle `INSERT` SQL?

(4) `update`: le colonne mappate devono apparire nelle `UPDATE` SQL?

(5) `access` (opzionale - il default è `property`): la strategia che Hibernate dovrebbe usare per accedere ai valori delle proprietà.

I tag figli `<property>` mappano proprietà della classe figlio a colonne della tabella.

L'elemento <component> consente un sottoelemento <parent> che mappa una proprietà della classe componente come un riferimento all'entità contenitore.

L'elemento <dynamic-component> consente ad una Map di essere mappata come componente in cui i nomi delle proprietà si riferiscono a chiavi della mappa.

5.1.13. subclass

Infine, la persistenza polimorfica richiede la dichiarazione di ogni sottoclasse della classe persistente radice. Per la strategia di mappaggio (raccomandata) tabella-per-classe, si usa la dichiarazione <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">

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

(1) name: il nomoe di classe completamente qualificato della sottoclasse.

(2) discriminator-value (opzionale - il default è il nome della classe): un valore che distingue le sottoclassi individuali.

(3) proxy (opzionale): specifica una classe o interfaccia da usare per i mediatori ad inizializzazione ritardata ("lazy initializing proxy").

(4) lazy (opzionale): impostare lazy="true" è un'abbreviazione equivalente a specificare il nome della classe stessa come interfaccia proxy.

Ogni sottoclasse dovrebbe dichiarare le sue proprietà persistenti e le sottoclassi. Si assume che le proprietà <version> e <id> siano ereditate dalla classe radice. Ogni sottoclasse in una gerarchia deve definire un valore unico di discriminator-value. Se non viene specificato un valore viene usato il nome della classe java completamente qualificato.

5.1.14. joined-subclass

In alternativa, una sottoclasse che sia resa persistente sulla sua propria tabella (strategia di mappaggio "tabella per sottoclasse") si dichiara usando un elemento <joined-subclass> element.

```
<joined-subclass
        name="ClassName"                    (1)
        proxy="ProxyInterface"              (2)
        lazy="true|false"                   (3)
        dynamic-update="true|false"
        dynamic-insert="true|false">

        <key .... >

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

(1) name: il nome di classe completamente qualificato della sottoclasse.

(2) proxy (opzionale): specifica una classe od interfaccia da usare per i mediatori a inizializzazione ritardata ("lazy initializing proxy").

(3) `lazy` (optional): impostare `lazy="true"` è un'abbreviazione equivalente a specificare il nome della classe stessa come interfaccia `proxy`.

Non viene richiesta alcuna colonna discriminatore per questa strategia di mappaggio. Ogni sottoclasse deve, però, dichiarare una colonna della tabella che contiene l'identificatore dell'oggetto usando l'elemento `<key>`. Il mappaggio all'inizio del capitolo verrebbe riscritto come:

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.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>
```

5.1.15. map, set, list, bag

Le collezioni sono descritte più avanti.

5.1.16. import

Supponete che la vostra applicazione abbia due classi persistenti con lo stesso nome, e non vogliate specificare il nome completamente qualificato (con il package) nelle interrogazioni di Hibernate. Le classi possono essere "importate" esplicitamente invece di fare affidamento a `auto-import="true"`. Potete anche importare classi e interfacce che non sono mappate esplicitamente.

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

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

(1) `class`: il nome completamente qualificato di una classe java qualunque.

(2) `rename` (opzionale - il default è il nome non qualificato della classe): un nome che può essere usato nel linguaggio di interrogazione.

5.2. Tipi di Hibernate

5.2.1. Entità e valori

Per capire il comportamento di vari oggetti java a livello del linguaggio rispetto al servizio di persistenza, dobbiamo classificarli in due gruppi:

Una *entity* esiste indipendentemente dal fatto che altri oggetti mantengano riferimenti ad essa. Questo è in contrasto con il modello java usuale in cui un oggetto non referenziato è fatto oggetto di garbage collection. Le entità devono essere salvate e cancellate esplicitamente (eccetto il fatto che salvataggi e cancellamenti possono essere *cascaded* ovvero propagati da un'entità genitore ai suoi figli). Questo è differente dal modello ODMG di persistenza per raggiungibilità degli oggetti - e corrisponde più strettamente a come gli oggetti applicativi sono solitamente usati nei grandi sistemi. Le entità supportano riferimenti circolari e condivisi: possono anche essere versionati.

Lo stato persistente di un'entità consiste di riferimenti ad altre entità e di istanze di *tipi di valore*. I valori sono tipi primitivi, collezioni componenti e alcuni oggetti immutabili. A differenza delle entità, i tipi di valore (in particolare le collezioni e i componenti) *sono* resi persistenti e cancellati per raggiungibilità. Poiché gli oggetti di valore (e i primitivi) sono resi persistenti e cancellati insieme all'entità che li contiene, non possono essere versionati indipendentemente. I valori non hanno identità indipendente, e per questo non possono essere condivisi da due entità o collezioni.

Tutti i tipi di Hibernate eccetto le collezioni devono supportare la semantica null.

Fino ad ora, abbiamo usato il termine "classe persistente" per riferirci alle entità. Continueremo a farlo. Parlando esattamente, però, non tutte le classi definite dall'utente con uno stato persistente sono entità. Un componente (*component*) è una classe definita dall'utente con semantica di valore.

5.2.2. Tipi di valore di base

I *tipi di base* possono a grandi linee essere catalogati in

`integer, long, short, float, double, character, byte, boolean, yes_no, true_false`
: Mappaggi di tipo dai primitivi java o dalle classi incapsulatore su tipi di colonna SQL appropriati (e specifici della marca del database). `boolean, yes_no` e `true_false` sono tutte codifiche alternative per un `boolean` o `java.lang.Boolean` di java.

`string`
: Mappaggio di tipo da `java.lang.String` a `VARCHAR` (o `VARCHAR2` di Oracle).

`date, time, timestamp`
: Mappaggi di tipo da `java.util.Date` e le sue sottoclassi ai tipi SQL `DATE`, `TIME` e `TIMESTAMP` (o equivalenti).

`calendar, calendar_date`
: Mappaggi di tipo da `java.util.Calendar` ai tipi SQL `TIMESTAMP` e `DATE` (o equivalenti).

`big_decimal`
: Mappaggio di tipo da `java.math.BigDecimal` a `NUMERIC` (o `NUMBER` in Oracle).

`locale, timezone, currency`
: Mappaggi di tipo da `java.util.Locale`, `java.util.TimeZone` e `java.util.Currency` verso `VARCHAR` (o

VARCHAR2 in Oracle). Istanze di Locale e Currency vengono mappati sui loro codici ISO. Le istanze di TimeZone vengono mappate sul loro ID.

class
: Mappaggio di tipo da java.lang.Class a VARCHAR (o VARCHAR2 in Oracle). Un oggetto Class viene mappato sul suo nome di classe completamente qualificato.

binary
: Mappa array di byte su un tipo binario SQL appropriato.

text
: Mappa stringhe java lunghe su un tipo CLOB or TEXT dell'SQL.

serializable
: Mappa tipi java serializzabili su un tipo binario SQL appropriato. Potete anche indicare il tipo Hibernate serializable con il nome di una classe java serializzabile o un'interfaccia che non abbia come default un tipo di base, o implementare PersistentEnum.

clob, blob
: Mappaggi di tipo per le classi JDBC java.sql.Clob e java.sql.Blob. Questi tipi possono non essere convenienti per alcune applicazioni, perché gli oggetti blob o clob non possono essere riutilizzati al di fuori di una transazione (e per di più il supporto da parte dei driver è approssimativo e inconsistente.)

Gli identificatori unici delle entità e le collezioni possono essere di qualsiasi tipo di base eccetto binary, blob e clob. (Sono permessi anche identificatori compositi, leggete più sotto.)

I tipi di valore di base hanno costanti Type corrispondenti definite nella classe net.sf.hibernate.Hibernate. Ad esempio, Hibernate.STRING rappresenta il tipo string.

5.2.3. Tipi di enumerazione persistente

Un tipo *enumerativo* è un idioma java comune in cui una classe ha un (piccolo) numero costante di istanze immutabili. Potete creare un tipo enumerativo persistente implementando net.sf.hibernate.PersistentEnum, definendo le operazioni toInt() e fromInt():

```
package eg;
import net.sf.hibernate.PersistentEnum;

public class Color implements PersistentEnum {
    private final int code;
    private Color(int code) {
        this.code = code;
    }
    public static final Color TABBY = new Color(0);
    public static final Color GINGER = new Color(1);
    public static final Color BLACK = new Color(2);

    public int toInt() { return code; }

    public static Color fromInt(int code) {
        switch (code) {
            case 0: return TABBY;
            case 1: return GINGER;
            case 2: return BLACK;
            default: throw new RuntimeException("Unknown color code");
        }
    }
}
```

Il nome di tipo di Hibernate in questo caso è semplicemente il nome della classe enumerativa. eg.Color.

5.2.4. Tipi di valore personalizzati

Per gli sviluppatori è relativamente facile creare i prorpi tipi di valore. Ad esempio, potreste desiderare rendere persistenti proprietà di tipo `java.lang.BigInteger` su colonne `VARCHAR`. Hibernate non fornisce un tipo predefinito per questo, ma i tipi personalizzati non sono limitati al mappaggio di una proprietà (o elemento di collezione) su una singola colonna di tabella. Allora, ad esempio, potreste avere una proprietà java `getName()`/`setName()` di tipo `java.lang.String` che sia resa persistente sulle colonne `FIRST_NAME`, `INITIAL`, `SURNAME`.

Per implementare un tipo personalizzato, implementate `net.sf.hibernate.UserType` o `net.sf.hibernate.CompositeUserType` e dichiarate le proprietà usando il nome di classe completamente qualificato del tipo. Guardate il codice di `net.sf.hibernate.test.DoubleStringType` per vedere il genere di cose che sono possibili.

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

Notate l'uso di elementi `<column>` per mappare una proprietà su colonne multiple.

Anche se l'insieme ricco di tipi predefiniti e il supporto per i componenti significa che avrete molto raramente *bisogno* di usare un tipo personalizzato, è comunque considerata una buona norma usare i tipi personalizzati per le classi (non di entità) che si presentino frequentemente nella vostra applicazione. Ad esempio, una classe `MonetaryAmount` è un buon candidato per un `CompositeUserType`, anche se potrebbe essere facilmente mappato come componente. Una ragione per questo è l'astrazione. Con un tipo personalizzato i vostri documenti di mappaggio sarebbero a prova di cambiamenti possibili nella vostra maniera di rappresentare valori monetari in futuro.

5.2.5. Tipi di mappaggio "any"

C'è un tipo ulteriore di mappaggio di proprietà. L'elemento di mappaggio `<any>` definisce una associazione polimorfica alle classi da tabelle multiple. Questo tipo di mappaggio richiede sempre più di una colonna. La prima colonna mantiene il tipo dell'entità associata. Le colonne rimanenti mantengono l'identificatore. È impossibile specificare un vincolo di chiave esterna per questo genere di associazioni, così non si tratta certamente del modo usuale di mappare associazioni (polimorfiche). Dovreste usarlo solo in casi molto speciali (ad esempio registri di auditing, dati delle sessioni utente, ecc.).

```
<any name="anyEntity" id-type="long" meta-type="eg.custom.Class2TablenameType">
    <column name="table_name"/>
    <column name="id"/>
</any>
```

L'attributo `meta-type` consente all'applicazione di specificare un tipo personalizzato che mappi valori di colonne del database su classi persistenti che abbiano proprietà identificatore del tipo specificato da `id-type`. Se il meta-tipo restituisce istanze di `java.lang.Class`, non è richiesto nient'altro. Da un altro punto di vista, se è un tipo basico come `string` o `character`, dovete specificare il mappaggio da valori a classi.

```
<any name="anyEntity" 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="none|all|save-update"          (4)
        access="field|property|ClassName"       (5)
>
        <meta-value ... />
        <meta-value ... />
        .....
        <column .... />
        <column .... />
        .....
</any>
```

(1) `name`: il nome della proprietà.

(2) `id-type`: il nome dell'identificatore.

(3) `meta-type` (opzionale - il default è `class`): un tipo che mappa `java.lang.Class` su una singola colonna del database o, alternativamente, un tipo che sia consentito per un mappaggio a discriminatore.

(4) `cascade` (opzionale - il default è `none`): il tipo di cascata.

(5) `access` (opzionale - il default è `property`): la strategia che Hibernate dovrebbe usare per accedere al valore delle proprietà.

Il vecchio tipo `object` che svolgeva un ruolo simile in Hibernate 1.2 è ancora supportato, ma è oramai semi-deprecato.

5.3. Identificatori SQL tra virgolette

Potete forzare Hibernate a mettere un identificatore tra virgolette nell'SQL generato mettendo il nome della tabella tra "backtick" (virgolette inverse) nel documento di mappaggio. Hibernate userà lo stile di virgolettatura corretta per il `Dialect` SQL (di solito sono virgolette doppie, ma SQL Server usa parentesi quadre, e MySQL usa backtick).

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

5.4. File di mappaggio modulari

È possibile definire mappaggi `subclass` e `joined-subclass` in documenti di mappaggio separati, direttamente sotto a `hibernate-mapping`. Questo vi consente di estendere una gerarchia di classe aggiungendo semplicemente un nuovo file di mappaggio. Dovete specificare un attributo `extends` nel mappaggio di sottoclasse, indicando una superclasse mappata preventivamente: l'uso di questa funzionalità fa sì che l'ordinamento dei documenti di mappaggio sia importante!

```

<hibernate-mapping>
        <subclass name="eg.subclass.DomesticCat" extends="eg.Cat" discriminator-value="D">
             <property name="name" type="string"/>
        </subclass>
</hibernate-mapping>
```

Capitolo 6. Mappaggio delle collezioni

6.1. Collezioni persistenti

Questa sezione non contiene molto codice Java di esempio. Diamo per scontato che conosciate già come usare l'infrastruttura delle collezioni in Java. Se è così, non c'è davvero niente di più da sapere - con un'unica avvertenza, potete usare le collezioni di Java nello stesso modo in cui avete sempre fatto.

Hibernate può rendere persistenti istanze di `java.util.Map`, `java.util.Set`, `java.util.SortedMap`, `java.util.SortedSet`, `java.util.List`, e qualsiasi array di entità persistenti o valori. Le proprietà di tipo `java.util.Collection` or `java.util.List` possono anche essere rese persistenti con semantiche a "sacco" ("bag").

Ed ora l'avvertenza: le collezioni persistenti non mantengono nessuna semantica aggiunta dalla classe che implementa l'interfaccia di base della collezione particolare (come ad esempio l'ordine di iterazione in un `LinkedHashSet`). Le collezioni persistenti in particolare si comportano come `HashMap`, `HashSet`, `TreeMap`, `TreeSet` e `ArrayList` rispettivamente. Inoltre, il tipo di oggetto java di una proprietà che contiene una collezione, deve essere quello dell'interfaccia (ovvero `Map`, `Set` o `List`; mai `HashMap`, `TreeSet` o `ArrayList`). Questa restrizione esiste perché, a vostra insaputa, Hibernate sostituisce le istanze di `Map`, `Set` e `List` con istanze delle sue implementazioni persistenti di queste interfacce. Per questo motivo, dovete anche fare attenzione quando usate `==` sulle collezioni.

```
Cat cat = new DomesticCat();
Cat kitten = new DomesticCat();
....
Set kittens = new HashSet();
kittens.add(kitten);
cat.setKittens(kittens);
session.save(cat);
kittens = cat.getKittens(); //Ok, la collezione "kittens" è un Set
(HashSet) cat.getKittens(); //Errore!
```

Le collezioni rispettano le normali regole per i tipi di valore: niente riferimenti condivisi, vengono create e cancellate insieme all'entità che le contiene. A causa delle caratteristiche del modello relazionale sottostante, non sopportano semantiche a valore nullo: Hibernate non distingue tra il riferimento ad una collezione nulla e una collezione vuota.

Le collezioni sono automaticamente rese persistenti quando sono referenziate da un oggetto persistente, e cancellate automaticamente quando il riferimento viene eliminato. Se una collezione viene passata da un oggetto persistente ad un altro, i suoi elementi vengono spostati da una tabella ad un'altra. Non dovreste preoccuparvi molto di questo: semplicemente usate le collezioni di Hibernate nello stesso modo in cui usate le normali collezioni di Java, ma assicuratevi di comprendere la semantica delle associazioni bidirezionali (discussa più avanti), prima di farlo.

Le istanze di collezione sono distinte nel database da una chiave esterna verso l'entità che le contiene. Questa chiave esterna viene chiamata la *chiave di collezione*. La chiave di collezione è mappata con l'elemento `<key>`.

Le collezioni possono contenere quasi ogni altro tipo di Hibernate, compresi tutti i tipi di base, i tipi personalizzati (custom), i tipi di entità e i componenti. Questa è una definizione importante: un oggetto in una collezione può essere gestito sia con una semantica di "passaggio per valore" (del resto dipende completamente dal proprietario della collezione) o può essere un riferimento ad un'altra entità di Hibernate, con il suo proprio ciclo di vita. Le collezioni non possono contenere altre collezioni. Il tipo contenuto viene chiamato il *tipo di elemento della collezione*. Gli elementi della collezione vengono mappati da`<element>`, `<composite-element>`,

<one-to-many>, <many-to-many> or <many-to-any>. I primi due mappano elementi con semantica di valore, mentre gli altri tre vengono usati per mappare associazioni tra entità.

Tutti i tipi di collezione eccetto Set e bag hanno una colonna *indice* , ovvero una colonna che mappa l'indice di un array o di una List o la chiave di una Map. L'indice di una Map può essere di qualsiasi tipo di base, un tipo di entità o anche un tipo composito (ma non può essere una collezione). L'indice di un array o di una lista è sempre di tipo integer. Gli indici vengono mappati usando <index>, <index-many-to-many>, <composite-index> o <index-many-to-any>.

C'è un insieme abbastanza vario di mappaggi che possono venire generati per le collezioni, e coprono molti modelli relazionali comuni. Suggeriamo che sperimentiate con lo strumento di generazione dello schema per avere un'idea di come i vari tipi di dichiarazione si traducono in tabelle di database.

6.2. Come mappare una collezione

Le collezioni vengono dichiarate tramite gli elementi <set>, <list>, <map>, <bag>, <array> e <primitive-array>. <map> è un buon esempio:

```
<map
    name="propertyName"                                          (1)
    table="table_name"                                           (2)
    schema="schema_name"                                         (3)
    lazy="true|false"                                            (4)
    inverse="true|false"                                         (5)
    cascade="all|none|save-update|delete|all-delete-orphan"      (6)
    sort="unsorted|natural|comparatorClass"                      (7)
    order-by="column_name asc|desc"                              (8)
    where="arbitrary sql where condition"                        (9)
    outer-join="true|false|auto"                                 (10)
    batch-size="N"                                               (11)
    access="field|property|ClassName"                            (12)
>

    <key .... />
    <index .... />
    <element .... />
</map>
```

(1) name il nome della proprietà corrispondente alla collezione

(2) table (opzionale - se assente è uguale al nome della proprietà) il nome della tabella che corrisponde alla collezione (non usato per le associazioni uno-a-molti)

(3) schema (opzionale) il nome di uno schema di tabella che sovrascriva quello dichiarato sull'elemento radice

(4) lazy (opzionale - se assente è false) consente l'inizializzazione differita (non usato per gli array)

(5) inverse (opzionale - se assente vale false) indica che questa collezione è il lato "opposto" di una associazione bidirezionale

(6) cascade (opzionale - se assente vale none) consente che le operazioni si propaghino sugli elementi figli della collezione

(7) sort (opzionale) specifica una collezione ordinata con un metodo di ordinamento naturale, o una classe di comparazione specifica

(8) order-by (opzionale, solo JDK1.4) specifica una colonna della tabella (o più colonne) che indica l'ordine di iterazione della Map, del Set o del bag, con un indicatore asc o desc (ascendente o discendente) opzionale.

(9) where (opzionale) specifica una condizione WHERE opzionale da usare quando si carica o rimuove la collezione (utile se la collezione deve contenere solo un sottoinsieme dei dati presenti)

(10) outer-join (opzionale) specifica che la collezione dovrebbe essere caricata tramite un outer join, quando possibile. Solo una collezione può venire caricata in questo modo in una SELECT SQL.

(11) batch-size (opzionale, per default vale 1) specifica una dimensione del "batch" (blocco di caricamento)

per il caricamento differito di istanze di questa collezione.

(12) `access` (opzionale - se assente vale `property`): La strategia che Hibernate dovrebbe utilizzare per accedere al valore di questa proprietà.

Il mappaggio di una `List` o di un array richiede una colonna separata della tabella per mantenere l'indice (l'elemento `i` in `foo[i]`). Se il vostro modello relazionale non ha una colonna indice, ad esempio perché state lavorando con dati preesistenti, usate un `Set` non ordinato. Questo sembra deludere le persone che assumono che una `List` sia un modo più conveniente di accedere ad una collezione non ordinata, ma le collezioni di Hibernate obbediscono strettamente alla semantica associata alle interfacce `Set`, `List` e `Map`, e semplicemente gli elementi di una `List` non si ordinano spontaneamente!

Da un altro punto di vista, le persone che immaginavano di usare `List` per emilare la semantica di un *bag* hanno un motivo legittimo di lamentela, qui. Un bag è una collezione non ordinata e non indicizzata di elementi, che può quindi contenere lo stesso elemento più volte. L'infrastruttura delle collezioni di java non specifica un'interfaccia `Bag`, per cui la si deve emulare con una `List`. Hibernate consente di mappare proprietà di tipo `List` o `Collection` con l'elemento `<bag>`. Notate che la semantica del sacco (bag) non sono realmente parte del contratto di `Collection` ed in realtà confliggono con il contratto della `List` contract (anche se, come discusso più avanti nel capitolo, potete ordinare a piacimento il bag).

Nota: bag molto grandi mappati con `inverse="false"` in Hibernate sono inefficienti, e andrebbero evitati; Hiberante non può creare, cancellare o aggiornare righe individualmente perché non c'è una chiave che possa identificare una riga singola.

6.3. Collezioni di valori e associazioni molti-a-molti

Una tabella di collezione è richiesta per ogni collezione di valori o di riferimenti ad altre entità che sia mappada come un'associazione molti-a-molti (la semantica naturale per una collezione java). La tabella richiede colonne di chiave (esterna), colonne di elemento e possibilmente colonne indice.

La chiave esterna dalla tabella di collezione verso la tabella della classe proprietaria è dichiarata usando un elemento `<key>`.

```
<key column="column_name"/>
```

(1) `column` (obbligatorio): Il nome della colonna di chiave esterna.

Per le collezioni indicizzate come le mappe e le liste, è necessario un elemento `<index>`. Per le liste, questa colonna deve contenere interi in sequenza, numerati a partire da zero. Assicuratevi che il vostro indice parta davvero da zero, se dovete avere a che fare con dati preesistenti. Per le mappe, la colonna può contenere valori di un tipo qualsiasi gestito da Hibernate.

```
<index
        column="column_name"                (1)
        type="typename"                     (2)
/>
```

(1) `column` (obbligatorio): Il nome della colonna che contiene i valori dell'indice di collezione.

(2) `type` (opzionale, se assente vale `integer`): Il tipo dell'indice di collezione.

In alternativa, una mappa può essere indicizzata da oggetti di tipo "entità". Usiamo in questo caso l'elemento `<index-many-to-many>`.

```
<index-many-to-many
        column="column_name"                (1)
        class="ClassName"                   (2)
```

```
/>
```

(1) `column` (obbligatorio): Il nome della colonna che contiene la chiave esterna verso i valori di indice della collezione.
(2) `class` (obbligatorio): La classe dell'entità che è usata come indice della collezione.

Per una collezione di valori usiamo l'etichetta `<element>`.

```
<element
        column="column_name"                (1)
        type="typename"                     (2)
/>
```

(1) `column` (obbligatorio): Il nome della colonna che contiene i valori degli elementi della collezione.
(2) `type` (obbligatorio): Il tipo degli elementi della collezione.

Una collezione di entità con la propria tabella corrisponde alla nozione relazionale di *associazione molti-a-molti*. Una associazione di questo tipo è il mappaggio più naturale per una collezione java, ma solitamente non rappresenta il miglior modello relazionale.

```
<many-to-many
        column="column_name"                                (1)
        class="ClassName"                                   (2)
        outer-join="true|false|auto"                        (3)
    />
```

(1) `column` (obbligatorio): Il nome della colonna con la chiave esterna verso l'elemento.
(2) `class` (obbligatorio): Il nome della classe associata.
(3) `outer-join` (opzionale - se assente vale `auto`): quando il parametro `hibernate.use_outer_join` è impostato, consente il caricamento via join esterno per questa associazione .

Alcuni esempio. Prima di tutto, un insieme di stringhe:

```
<set name="names" table="NAMES">
    <key column="GROUPID"/>
    <element column="NAME" type="string"/>
</set>
```

Un "sacco" (bag) contenente interi (con un ordine di iterazione determinato dall'attributo `order-by`):

```
<bag name="sizes" table="SIZES" order-by="SIZE ASC">
    <key column="OWNER"/>
    <element column="SIZE" type="integer"/>
</bag>
```

Un array di entità - in questo caso un'associazione molti-a-molti (notate che le entità vengono gestite nel ciclo di vita dell'entità proprietaria, grazie al settaggio `cascade="all"`):

```
<array name="foos" table="BAR_FOOS" cascade="all">
    <key column="BAR_ID"/>
    <index column="I"/>
    <many-to-many column="FOO_ID" class="org.hibernate.Foo"/>
</array>
```

Una mappa da indici stringa a date:

```
<map name="holidays" table="holidays" schema="dbo" order-by="hol_name asc">
    <key column="id"/>
    <index column="hol_name" type="string"/>
```

```
    <element column="hol_date" type="date"/>
</map>
```

Una lista di componenti (discussi nel prossimo capitolo):

```
<list name="carComponents" table="car_components">
    <key column="car_id"/>
    <index column="posn"/>
    <composite-element class="org.hibernate.car.CarComponent">
            <property name="price" type="float"/>
            <property name="type" type="org.hibernate.car.ComponentType"/>
            <property name="serialNumber" column="serial_no" type="string"/>
    </composite-element>
</list>
```

6.4. Associazioni uno-a-molti

Una *associazione uno a molti* collega *direttamente* le tabelle di due classi, senza che intervenga una apposita tabella di collezione. (ciò corrisponde al modello relazionale *uno-a-molti*.) In questo modello, si perde un po' della semantica delle collezioni di Java:

- Non è possibile avere valori null contenuti in una mappa, un insieme o una lista.
- Una istanza dell'entità contenuta non può appartenere a più di una istanza della collezione.
- Una istanza dell'entità contenuta non può apparire in corrispondenza di più di un valore dell'indice di collezione.

Un'associazione da `Pippo` a `Pluto` richiede l'aggiunta di una colonna chiave, e possibilmente anche di una colonna indice alla tabella della classe di entità contenuta, `Pluto`. Queste colonne vengono mappate usando gli elementi `<key>` e `<index>` già descritti in precedenza.

L'etichetta `<one-to-many>` indica un'associazione uno a molti.

```
<one-to-many class="ClassName"/>
```

(1) `class` (obbligatorio): Il nome della classe associata.

Esempio:

```
<set name="bars">
    <key column="foo_id"/>
    <one-to-many class="org.hibernate.Bar"/>
</set>
```

Notate che l'elemento `<one-to-many>` non ha bisogno di dichiarare alcuna colonna. Non è neppure necessario specificare il nome della tabella.

Nota Molto Importante: se la colonna `<key>` di una associazione `<one-to-many>` viene dichiarata `NOT NULL`, Hibernate può causare violazioni di vincoli quando crea o aggiorna le associazioni. Per prevenire questo problema, *dovete usare una associazione bidirezionale* con l'estremità "many" (l'insieme o il sacco) impostati a `inverse="true"`. Per ulteriori informazioni si legga la discussioni sulle associazioni bidirezionali più avanti in questo capitolo.

6.5. Inizializzazione differita (lazy)

Le collezioni (a differenza degli array) possono essere inizializzate in maniera differita, ovvero possono caricare il proprio stato dal database solo quando l'applicazione ha bisogno di accedervi. L'inizializzazione avviene trasparentemente per l'utente, in modo tale che l'applicazione non ha normalmente bisogno di preoccuparssene (in effetti, l'inizializzazione trasparente e differita è la ragione principale per cui Hibernate ha bisogno di implementazioni proprie delle collezioni). Nonostante ciò, se l'applicazione tenta di fare qualcosa come nel codice seguente:

```
s = sessions.openSession();
User u = (User) s.find("from User u where u.name=?", userName, Hibernate.STRING).get(0);
Map permissions = u.getPermissions();
s.connection().commit();
s.close();

Integer accessLevel = (Integer) permissions.get("accounts");  // Error!
```

Può ritrovarsi di fronte ad una brutta sorpresa. Poiché la collezione dei permessi (permissions) non era stata inizializzata quando la `Session` è stata committata, la collezione non sarà mai capace di caricare il suo stato. La correzione consiste nel muovere la riga che legge dalla collezione subito prima del commit. (In ogni caso ci sono altre maniere più avanzate di risolvere il problema).

In alternativa, potete usare una collezione ad inizializzazione non differita. Poiché l'inizializzazione differita può portare a bachi come nel codice precedente, la "non-lazyness" (il "non differimento" o, letteralmente, la "non pigrizia") è il comportamento predefinito. Comunque, è sottinteso che l'inizializzazione differita venga usata per quasi tutte le collezioni, in particolar modo per le collezioni di entità (per questioni di efficienza).

Le eccezioni che accadono mentre si inizializza in maniera differita le collezioni sono incapsulate in una `LazyInitializationException`.

Potete dichiarare una collezione differita usando l'attributo opzionale `lazy`:

```
<set name="names" table="NAMES" lazy="true">
    <key column="group_id"/>
    <element column="NAME" type="string"/>
</set>
```

In alcune architetture applicative, in particolare quando il codice che accede ai dati con Hibernate e il codice che lo usa sono in differenti livelli applicativi, può essere un problema assicurarsi che la `Session` sia aperta quando una collezione viene inizializzata. Ci sono due maniere principali, per trattare questa questione:

- In una applicazione basata sul web, si può usare un "servlet filter" per chiudere la `Session` solo alla fine della richiesta di un utente, quando la costruzione della vista è completa. Naturalmente, questo impone dei vincoli molto importanti sulla correttezza della gestione delle eccezioni nella vostra infrastruttura applicativa. È vitalmente importante che la `Session` venga chiusa e la transazione conclusa prima di restituire il controllo all'utente, anche quando una eccezione avviene durante la resa della vista. Il servlet filter deve essere in grado di accedere la `Session` perché questo sia possibile. Raccomandiamo l'uso di una variabile `ThreadLocal` per mangenere la `Session` corrente (vedere il capitolo 1, Sezione 1.4, “Giochiamo con i gatti”, per un'implementazione di esempio).

- In una applicazione con uno strato di business separato, la logica applicativa deve "preparare" tutte le collezioni che saranno necessarie per lo strato web prima di ritornare. Questo significa che lo strato di business dovrebbe caricare tutti i dati che siano richiesti per un particolare caso d'uso, e restituirli allo strato web di presentazione una volta inizializzati . Solitamente, l'applicazione chiama `Hibernate.initialize()` per ogni collezione che sarà necessaria nello strato web (la chiamata deve avvenire prima che la sessione venga chiusa), o carica la collezione direttamente usando una query di Hibernate che comprende una clausola `FETCH`.

- Potete anche attaccare un oggetto caricato precedentemente ad una nuova `Session` con `update()` o `lock()` prima di accedere a collezioni non inizializzate (o altri tipi di mediatori). Hibernate non può farlo automaticamente, perché dovrebbe introdurre semantica transazionale ad-hoc!

Potete usare il metodo `filter()` dell'API della classe Session di Hibernate per ottenere la dimensione di una collezione senza doverla inizializzare:

```
( (Integer) s.filter( collection, "select count(*)" ).get(0) ).intValue()
```

`filter()` o `createFilter()` vengono anche usati per caricare efficientemente sottoinsiemi di una collezione senza bisogno di inizializzare l'intera collezione.

6.6. Collezioni ordinate

Hibernate supporta collezioni che implementano `java.util.SortedMap` e `java.util.SortedSet`. Dovete specificare esplicitamente un comparatore nel file di mappaggio:

```
<set name="aliases" table="person_aliases" sort="natural">
    <key column="person"/>
    <element column="name" type="string"/>
</set>

<map name="holidays" sort="my.custom.HolidayComparator" lazy="true">
    <key column="year_id"/>
    <index column="hol_name" type="string"/>
    <element column="hol_date type="date"/>
</map>
```

I valori permessi per l'attributo `sort` sono `unsorted`, `natural` e il nome di una classe che implementi `java.util.Comparator`.

Le collezioni ordinate si comportano in effetti come `java.util.TreeSet` o `java.util.TreeMap`.

Se volete che sia il database stesso ad ordinare gli elementi della collezione, usate l'attributo `order-by` dei mappaggi dei `set`, `bag` o delle `map`. Questa soluzione funziona solo sotto JDK 1.4 o superiori (è implementata usando `LinkedHashSet` o `LinkedHashMap`). Questo fa sì che l'ordinamento avvenga durante l'esecuzione della query SQL, non in memoria.

```
<set name="aliases" table="person_aliases" order-by="name asc">
    <key column="person"/>
    <element column="name" type="string"/>
</set>

<map name="holidays" order-by="hol_date, hol_name" lazy="true">
    <key column="year_id"/>
    <index column="hol_name" type="string"/>
    <element column="hol_date type="date"/>
</map>
```

Notate che il valore dell'attributo `order-by` si riferisce all'SQL, non all'HQL!

Le associazioni possono anche venire ordinate usando criteri arbitrari in fase di esecuzione usando un `filter()`.

```
sortedUsers = s.filter( group.getUsers(), "order by this.name" );
```

6.7. L'uso degli <idbag>

Se avete abbracciato il nostro punto di vista, secondo cui le chiavi composte sono una cattiva cosa, e le entità dovrebbero avere identificatori sintetici (chiavi surrogate), potreste trovare un po' strano che le associazioni molti-a-molti e le collezioni di valori che abbiamo mostrato fin qui, si mappano tutte su tabelle con chiavi composte! Ora, questo punto è abbastanza discutibile; una tabella di pura associazione non sembra trarre molto beneficio da una chiave surrogata (benché una collezione di valori composti *potrebbe*). Ciononostante, Hibern ate fornisce una funzionalità (lievemente sperimentale), che consente di mappare associazioni molti-a-molti e collezioni di valori su una tabella con una chiave surrogata.

L'elemento `<idbag>` vi permette di mappare una `List` (o una `Collection`) con la semantica del sacco ("bag").

```
<idbag name="lovers" table="LOVERS" lazy="true">
    <collection-id column="ID" type="long">
        <generator class="hilo"/>
    </collection-id>
    <key column="PERSON1"/>
    <many-to-many column="PERSON2" class="eg.Person" outer-join="true"/>
</idbag>
```

Come potete vedere, un `<idbag>` ha un generatore di id sintetici essattamente come una classe di entità! Una chiave surrogata diversa viene assegnata ad ogni riga della collezione. Hibernate non fornisce alcun meccanismo per scoprire la chiave surrogata di una particolare riga, però.

Notate che le performance in aggiornamento di un `<idbag>` sono *molto* migliori di un `<bag>` normale! Hibernate può individuare chiavi individuali in maniera efficiente, e aggiornare o cancellarle individualmente esattamente come in una lista, una mappa o un insieme.

Nell'implementazione corrente, la strategia di generazione degli identificatori indicata con `identity` non viene supportata per gli identificatori di collezione `<idbag>`.

6.8. Associazioni bidirezionali

Una *associazione bidirezionale* consente la navigazione da entrambe le estremità dell'associazione. Vengono supportati due stili differenti di associazioni bidirezionali:

one-to-many
: un insieme o un sacco ad una estremità, un valore singolo dall'altra

many-to-many
: entrambe le estremità sono valorizzate con un set o un sacco

Tenete presente che HIbernate non supporta associazioni bidirezionali uno-a-molti con una collezione indicizzata (lista, mappa o array) come estremità "molti": dovete usare un mappaggio "set" o "bag".

Potete specificare una associazione bidirezionale molti-a-molti semplicemente mappando due associazioni molti-a-molti sulla stessa tabella di database e dichiarando una estremità come *inverse* (decidere quale è una scelta che sta a voi). Ecco un esempio di un'associazione bidirezionale molti-a-molti da una classe a se *stessa* (ogni categoria può avere molti elementi, ed ogni elemento può essere in molte categorie):

```
<class name="org.hibernate.auction.Category">
    <id name="id" column="ID"/>
    ...
    <bag name="items" table="CATEGORY_ITEM" lazy="true">
```

```
        <key column="CATEGORY_ID"/>
        <many-to-many class="org.hibernate.auction.Item" column="ITEM_ID"/>
    </bag>
</class>

<class name="org.hibernate.auction.Item">
    <id name="id" column="ID"/>
    ...

    <!-- inverse end -->
    <bag name="categories" table="CATEGORY_ITEM" inverse="true" lazy="true">
        <key column="ITEM_ID"/>
        <many-to-many class="org.hibernate.auction.Category" column="CATEGORY_ID"/>
    </bag>
</class>
```

Dei cambiamenti fatto esclusivamente all'estremità "inversa" dell'associazione *non* vengono resi persistenti. Questo significa che Hibernate ha due rappresentazioni in memoria per ogni associazione bidirezionale: un collegamento da A a B e un altro collegamento da B ad A. Questa cosa è più facile da comprendere se pensate al modello ad oggetti Java e come creiamo una relazione molti-a-molti in Java:

```
category.getItems().add(item);          // La categoria ora "sa" della relazione
item.getCategories().add(category);     // L'elemento ora "sa" della relazione

session.update(item);                     // Nessun effetto, niente verrà salvato!
session.update(category);                 // La relazione verrà salvata
```

Il lato non-inverso viene usato per salvare la rappresentazione in-memoria sul database. Otterremmo un INSERT/UPDATE non necessario e probabilmente anche una violazione di chiave esterna se entrambe le estremità scatenassero dei cambiamenti! La stessa cosa vale naturalmente per le associazioni bidirezionali uno-a-molti.

Potete mappare una associazione bidirezionale uno-a-molti mappandola sulle stesse colonne come una associazione molti-a-uno e dichiarando l'estremità "molti" come `inverse="true"`.

```
<class name="eg.Parent">
    <id name="id" column="id"/>
    ....
    <set name="children" inverse="true" lazy="true">
        <key column="parent_id"/>
        <one-to-many class="eg.Child"/>
    </set>
</class>

<class name="eg.Child">
    <id name="id" column="id"/>
    ....
    <many-to-one name="parent" class="eg.Parent" column="parent_id"/>
</class>
```

Mappare una estremità di un'associazione con `inverse="true"` non condiziona il funzionamento delle cascate, si tratta di concetti differenti!

6.9. Associazioni ternarie

Ci sono due approcci possibili per mappare una associazione ternaria. Uno è usare elementi compositi (discusso più avanti). Un altro è di usare una mappa con un'associazione come indice:

```
<map name="contracts" lazy="true">
    <key column="employer_id"/>
    <index-many-to-many column="employee_id" class="Employee"/>
```

```
    <one-to-many class="Contract"/>
</map>
```

```
<map name="connections" lazy="true">
    <key column="node1_id"/>
    <index-many-to-many column="node2_id" class="Node"/>
    <many-to-many column="connection_id" class="Connection"/>
</map>
```

6.10. Associazioni eterogenee

Gli elementi `<many-to-any>` e `<index-many-to-any>` permettono di utilizzare vere e proprie associazioni eterogenee. Questi elementi di mappaggio funzionano nello stesso modo ini cui funziona l'elemento `<any>`, e come questo dovrebbero essere usate raramente, se proprio devono esserlo.

6.11. Esempi di collezioni

Le sezioni precedenti sono abbastanza complesse, quindi vediamo un esempio. La classe seguente:

```
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; }

    ....
    ....
}
```

ha una collezione di istanze di `eg.Child`. Se ogni figlio ha al più un genitore, il mappaggio più naturale è un'associazione uno-a-molti:

```
<hibernate-mapping>

    <class name="eg.Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" lazy="true">
            <key column="parent_id"/>
            <one-to-many class="eg.Child"/>
        </set>
    </class>

    <class name="eg.Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

Questo si mappa sulle seguenti definizioni di tabella:

```
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
```

Se il genitore è *obbligatorio*, usate una associazione bidirezionale uno-a-molti:

```
<hibernate-mapping>

    <class name="eg.Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" inverse="true" lazy="true">
            <key column="parent_id"/>
            <one-to-many class="eg.Child"/>
        </set>
    </class>

    <class name="eg.Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
        <many-to-one name="parent" class="eg.Parent" column="parent_id" not-null="true"/>
    </class>

</hibernate-mapping>
```

Notate il vincolo `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
```

Dall'altro lato, se un figlio potesse avere genitori multipli, sarebbe appropriata una associazione molti-a-molti:

```
<hibernate-mapping>

    <class name="eg.Parent">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <set name="children" lazy="true" table="childset">
            <key column="parent_id"/>
            <many-to-many class="eg.Child" column="child_id"/>
        </set>
    </class>

    <class name="eg.Child">
        <id name="id">
            <generator class="sequence"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

Definizioni delle tabelle:

```
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
```

Capitolo 7. Mappaggio dei componenti

La nozione di un *componente* viene usata in differenti contesti per scopi diversi, in tutto Hibernate.

7.1. Oggetti dipendenti

Un componente è un oggetto contenuto, che viene reso persistente come un tipo di valore ("value type"), non un'entità. Il termine "componente" si riferisce al concetto "orientato agli oggetti" della composizione (non a componenti di livello architetturale). Per esempio, potreste modellare una persona come segue:

```
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;
    }
}
```

Ora `Name` può essere reso persistente come un componente di `Person`. Notate che `Name` definisce metodi "getter" e "setter" per le sue proprietà persistenti, ma non deve dichiarare alcuna interfaccia o proprietà identificatore.

Il nostro mappaggio Hibernate avrebbe questo aspetto:

```
<class name="eg.Person" table="person">
    <id name="Key" column="pid" type="string">
        <generator class="uuid.hex"/>
    </id>
    <property name="birthday" type="date"/>
    <component name="Name" class="eg.Name"> <!-- l'attributo class è opzionale -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </component>
</class>
```

La tabella "person" avrebbe le colonne `pid`, `birthday`, `initial`, `first` e `last`.

Come tutti i tipi di valore, i componenti non supportano riferimenti condivisi. La semantica di valore nullo di un componente è *ad hoc*. Quando si ricarica l'oggetto contenitore, Hibernate supporrà che se tutte le colonne del componente sono nulle, allora l'intero componente è nullo. Questo dovrebbe adattarsi alla maggior parte degli scopi.

Le proprietà di un componente possono essere di un tipo qualunque di Hibernate (collezioni associazioni molti-a-uno, altri componenti, ecc.). Componenti annidati *non* dovrebbero essere considerati un utilizzo esotico. Hibernate è pensato per supportare un modello ad oggetti a grana molto fine.

L'elemento `<component>` consente di usare un sotto-elemento `<parent>` che mappa la proprietà di una classe componente come un riferimento "indietro" all'entità contenitore.

```
<class name="eg.Person" table="person">
    <id name="Key" column="pid" type="string">
        <generator class="uuid.hex"/>
    </id>
    <property name="birthday" type="date"/>
    <component name="Name" class="eg.Name">
        <parent name="namedPerson"/> <!-- retro-riferimento all'oggetto Person -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </component>
</class>
```

7.2. Collezioni di oggetti dipendenti

Le collezioni di componenti sono permesse (ad esempio un array di tipo `Name`). Dichiarate le collezioni di componenti rimpiazzando l'etichetta `<element>` con una `<composite-element>`.

```
<set name="someNames" table="some_names" lazy="true">
    <key column="id"/>
    <composite-element class="eg.Name"> <!-- l'attributo class è obbligatorio -->
        <property name="initial"/>
        <property name="first"/>
        <property name="last"/>
    </composite-element>
</set>
```

Nota: se definite un `Set` di elementi composti, è molto importante definire correttamente `equals()` e `hashCode()` correctly.

Gli elementi composti possono contenere componenti ma non collezioni. Se il vostro elemento composto con-

tiene componenti, usate l'etichetta `<nested-composite-element>`. Si tratta di un caso abbastanza esotico - una collezione di componenti che a loro volta hanno componenti. A questo stadio dovreste chiedervi se una associazione uno-a-molti non sia più appropriata. Provate a rimodellare l'elemento composto come una entità - ma notate che anche se il modello java è lo stesso, il modello relazionale e la semantica di persistenza sono leggermente diversi.

Tenete presente che un mappaggio ad elemento composto non supporta proprietà nulle se state usando un `<set>`. Hibernate deve usare ogni colonna per identificare un record quando cancella oggetti (non c'è una colonna separata di chiave primaria, nella tabella dell'elemento composto), cosa che non è possibile con valori nulli. In un composite-element dovete usare solo proprietà non nulle o scegliere una `<list>`, `<map>`, `<bag>` o `<idbag>`.

Un caso speciale di elemento composto è quello in cui l'elemento stesso ha un altro elemento annidato `<many-to-one>`. Un mappaggio di questo tipo, vi consente di mappare colonne extra di una tabella molti-a-molti sulla classe dell'elemento composto. Qui di seguito mostriamo una associazione molti-a-molti da `Order` a `Item` in cui `purchaseDate`, `price` e `quantity` sono proprietà dell'associazione:

```
<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"/> <!-- l'attributo class è opzionale -->
        </composite-element>
    </set>
</class>
```

Sono possibili anche associazioni ternarie (o quaternarie, ecc):

```
<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>
```

Gli elementi composti possono apparire nelle query usando la stessa sintassi delle associazioni ad altre entità.

7.3. Componenti come indici delle mappe

L'elemento `<composite-index>` vi consente di mappare una classe di componente come chiave di una `Map`. Assicuratevi di implementare `hashCode()` e `equals()` correttamente sulla classe componente, in questo caso.

7.4. Componenti come identificatori composti

Potete usare un componente come un identificatore di una classe di entità. La vostra classe di componente deve soddisfare alcuni requisiti:

- Deve implementare `java.io.Serializable`.
- Deve re-implementare `equals()` and `hashCode()`, consistentemente con la nozione di uguaglianza di chiave

sul database.

Non potete usare un `IdentifierGenerator` per generare chiavi composte. Al contrario, sarà l'applicazione che deve assegnare i propri identificatori.

Poiché un identificatore composto deve venire assegnato all'oggetto prima di salvarlo, non possiamo usare un "valore non salvato" (`unsaved-value`) sull'identificatore per distinguere tra istanze appena istanziate e istanze salvate in una sessione precedente.

Se volete usare `saveOrUpdate()` o save / update in cascata, potete invece implementare `Interceptor.isUnsaved()` . In alternativa, potete anche impostare l'attributo `unsaved-value` su un elemento `<version>` (o `<timestamp>`) per specificare il valore che identifica una nuova istanza transiente. In questo caso, viene usata la versione dell'entità invece dell'identificatore (assegnato), e non dovete essere voi ad implementare `Interceptor.isUnsaved()`.

Per dichiarare un identificatore di classe composta, usate l'elemento `<composite-id>` (con gli stessi attributi ed elementi di `<component>`) al posto di `<id>`:

```
<class name="eg.Foo" table"FOOS">
    <composite-id name="compId" class="eg.FooCompositeID">
        <key-property name="string"/>
        <key-property name="short"/>
        <key-property name="date" column="date_" type="date"/>
    </composite-id>
    <property name="name"/>
    ....
</class>
```

Ora, qualsiasi chiave esterna verso la tabella `FOOS` deve necessariamente essere composta, e dovete dichiararlo nei vostri mappaggi delle altre classi. Una associazione verso `Foo` verrà dichiarata in questo modo:

```
<many-to-one name="foo" class="eg.Foo">
<!-- come sempre l'attributo "class" è opzionale -->
    <column name="foo_string"/>
    <column name="foo_short"/>
    <column name="foo_date"/>
</many-to-one>
```

Questo nuovo elemento `<column>` viene anche usato dai tipi personalizzati multi-colonna. In effetti è ovunque un'alternativa all'attributo `column`. Una collezione con elementi di tipo `Foo` utilizzerebbe:

```
<set name="foos">
    <key column="owner_id"/>
    <many-to-many class="eg.Foo">
        <column name="foo_string"/>
        <column name="foo_short"/>
        <column name="foo_date"/>
    </many-to-many>
</set>
```

Dall'altro lato, `<one-to-many>`, non dichiara colonne, come sempre.

Se lo stesso `Foo` contiene collezioni, anch'esse richiederanno una chiave esterna composta.

```
<class name="eg.Foo">
    ....
    ....
    <set name="dates" lazy="true">
        <key>   <!-- la collezione eredita il tipo della chiave composta -->
            <column name="foo_string"/>
            <column name="foo_short"/>
            <column name="foo_date"/>
```

```
        </key>
        <element column="foo_date" type="date"/>
    </set>
</class>
```

7.5. Componenti dinamici

Potete anche mappare una proprietà di tipo `Map`:

```
<dynamic-component name="userAttributes">
    <property name="foo" column="FOO"/>
    <property name="bar" column="BAR"/>
    <many-to-one name="baz" class="eg.Baz" column="BAZ"/>
</dynamic-component>
```

La semantica di un mappaggio `<dynamic-component>` è identica a `<component>`. Il vantaggio di questo tipo di mappaggio è la capacità di determinare le vere proprietà del bean in fase di messa in esecuzione, semplicemente cambiando il documento di mappaggio. (È anche possibile manipolare in fase di esecuzione il documento di mappaggio usando un parser DOM)

Capitolo 8. Mappaggio di gerarchie di ereditarietà

8.1. Le tre strategie

Hibernate supporta le tre strategie di base per il mappaggio dell'ereditarietà.

- una tabella per un'intera gerarchia di classi
- una tabella per ogni sottoclasse
- una tabella per ogni classe concreta (con qualche limitazione)

È anche possibile utilizzare differenti strategie di mappaggio per rami differenti della stessa gerarchia di ereditarietà, ma questo scenario è soggetto alle stesse limitazioni dei mappaggi "una tabella per ogni classe concreta" (che vedremo nel seguito). Hibernate non supporta la possibilità di mischiare mappaggi `<subclass>` e `<joined-subclass>` nello stesso elemento `<class>`.

Immaginiamo di avere un'interfaccia `Payment`, con le seguenti classi che la implementino: `CreditCardPayment`, `CashPayment`, `ChequePayment`. Il mappaggio a "una tabella per gerarchia" apparirebbe così:

```
<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">
        ...
    </subclass>
    <subclass name="CashPayment" discriminator-value="CASH">
        ...
    </subclass>
    <subclass name="ChequePayment" discriminator-value="CHEQUE">
        ...
    </subclass>
</class>
```

È quindi necessaria esattamente una tabella. C'è una importante limitazione in questa strategia di mappaggio: le colonne dichiarate dalle sottoclassi non possono avere vincoli `NOT NULL`.

Il mappaggio a "una tabella per sottoclasse" apparirebbe così:

```
<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"/>
        ...
    </subclass>
    <joined-subclass name="CashPayment" table="CASH_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
    </subclass>
    <joined-subclass name="ChequePayment" table="CHEQUE_PAYMENT">
        <key column="PAYMENT_ID"/>
        ...
```

```
    </subclass>
</class>
```

Sono quindi richieste quattro tabelle. Le tre tabelle di sottoclasse hanno associazioni di chiave primaria con la tabella di superclasse (cosicché il modello relazionale è in realtà una associazione uno-a-uno).

È importante notare che l'implementazione di Hibernate della strategia "una tabella per sottoclasse" non richiede una colonna discriminatore. Altri sistemi di mappaggio oggetto/relazione usano una implementazione differente di questa strategia, che richiede una colonna di discriminazione del tipo nella tabella della superclasse. L'approccio assunto da Hibernate è molto più difficile da implementare, ma più corretto da un punto di vista relazionale.

Per ognuna di queste strategie di mappaggio, una associazione polimorfica a `Payment` viene mappata usando `<many-to-one>`.

```
<many-to-one name="payment"
    column="PAYMENT"
    class="Payment"/>
```

La strategia "una tabella per classe concreta" è molto differente.

```
<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>
```

Sono state necessarie tre tabelle. Si noti che non menzioniamo esplicitamente da nessuna parte l'interfaccia `Payment`. Invece, usiamo il *polimorfismo implicito* di Hibernate. Notate anche che le proprietà di `Payment` sono state mappate in ognuna delle sue sottoclassi.

In questo caso, una associazione polimorfica a `Payment` viene mappata usando `<any>`.

```
<any name="payment"
        meta-type="class"
        id-type="long">
    <column name="PAYMENT_CLASS"/>
    <column name="PAYMENT_ID"/>
</any>
```

Sarebbe meglio se definissimo uno `UserType` come `meta-tipo`, per gestire il mappaggio dalle stringhe di discriminazione verso la sottoclasse di `Payment`.

```
<any name="payment"
```

```
        meta-type="PaymentMetaType"
        id-type="long">
    <column name="PAYMENT_TYPE"/> <!-- CREDIT, CASH or CHEQUE -->
    <column name="PAYMENT_ID"/>
</any>
```

C'è ancora una cosa da considerare riguardo a questo mappaggio. Poiché le sottoclassi sono mappate ognuna nel proprio elemento `<class>` (e poiché `Payment` è solo un'interfaccia), ognuna delle sottoclassi potrebbe essere parte di un'altra gerarchia di ereditarietà di tipo "tabella per classe" o "tabella per sottoclasse"! (Ed è comunque possibile lanciare interrogazioni polimorfiche sull'interfaccia `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>
```

Anche in questo caso non menzioniamo `Payment` esplicitamente. Se eseguiamo un'interrogazione sull'interfaccia `Payment` - ad esempio, `from Payment` - Hibernate resistuisce automaticamente istanze di `CreditCardPayment` (e delle sue sottoclassi, poiché anch'esse implementano `Payment`), `CashPayment` e `ChequePayment` ma non istanze di `NonelectronicTransaction`.

8.2. Limitazioni

Hibernate assume che un'associazione corrisponda esattamente ad una colonna di chiave esterna. Associazioni multiple per chiave esterna sono tollerate (potete avere bisogno di specificare `inverse="true"` o `insert="false" update="false"`), ma non c'è modo di mappare una associazione a più chiavi esterne. Questo significa che:

- quando un'associazione viene modificata, è sempre la stessa chiave esterna che viene aggiornata
- quando un'associazione è risolta in maniera differita ("fetched lazily"), viene usata una singola interrogazione sulla base di dati
- quando un'associazione è risolta in maniera immediata ("fetched eagerly"), può venire risolta usando una singola join esterna

In particolare, questo implica che le associazioni polimorfiche uno-a-molti verso classi mappate usando la strategia "tabella per classe concreta" *non sono supportate*. (Risolvere queste associazioni implicherebbe effettuare interrogazioni o join multiple.)

La tabella seguente mostra le limitazioni dei mappaggi a "tabella per classe concreta" e del polimorfismo implicito in Hibernate.

Tabella 8.1. Funzionalità dei mappaggi di ereditarietà

Strategia di ereditarietà	**Molti-a-uno polimorfico**	**Uno-a-uno polimorfico**	**Uno-a-molti polimorfico**	**Molti-a-molti polimorfico**	**`load()/get()` polimorfiche**	**Interrogazioni polimorfiche**	**Join polimorfici**
"tabella per gerarchia"	`<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`
"tabella per sottoclasse"	`<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`
"tabella per classe concreta" (polimorfismo implicito)	`<any>`	*non supportata*	*non supportata*	`<many-to-any>`	*usando una query*	`from Payment p`	*non supportata*

Capitolo 9. Lavorare con i dati persistenti

9.1. Creazione di un oggetto persistente

Un oggetto (istanza di entità) è *transiente* o *persistente* rispetto ad una particolare `sessione (Session)`. Oggetti appena istanziati sono naturalmente transienti. La sessione offre servizi per salvare (cioè rendere persistenti, o "persistere") istanze transienti:

```
DomesticCat fritz = new DomesticCat();
fritz.setColor(Color.GINGER);
fritz.setSex('M');
fritz.setName("Fritz");
Long generatedId = (Long) sess.save(fritz);
```

```
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) );
```

Il metodo `save()` con un solo argomento genera e assegna un identificatore unico a `fritz`. La forma con due argomenti tenta di rendere persistente `pk` utilizzando l'identificatore fornito. In generale scoraggiamo l'uso della forma con due argomenti perché può essere usata per creare chiavi primarie con un significato "di business" (ovvero legato in qualche modo al dominio applicativo). È più utile in certe situazioni speciali come quando si usa Hibernate per persistere un bean di entità ("entity bean") di tipo BMP ("bean managed persistence").

Gli oggetti associati possono essere resi persistenti in un ordine qualunque, a meno che abbiate un vincolo `NOT NULL` su una colonna di chiave esterna. Non c'è rischio di violare vincoli di chiave esterna, però potreste violare un vincolo `NOT NULL` se salvate (`save()`) gli oggetti nell'ordine sbagliato.

9.2. Caricamento di un oggetto

I metodi `load()` della `Session` vi danno un modo per recuperare un'istanza persistente se già conoscete il suo identificatore. Una prima versione del metodo riceve un oggetto di tipo "class" e caricherà lo stato in un oggetto nuovo. La seconda versione consente di fornire un'istanza in cui verrà caricato lo stato. La forma che riceve un'istanza è particolarmente utile se progettate di usare Hibernate con bean di entità BMP ed è fornita esattamente per questa ragione. Potete comunque scoprire altri usi (pooling di istanza fatto in casa, ecc.).

```
Cat fritz = (Cat) sess.load(Cat.class, generatedId);
```

```
// bisogna incapsulare gli identificatori primitivi
long pkId = 1234;
DomesticCat pk = (DomesticCat) sess.load( Cat.class, new Long(pkId) );
```

```
Cat cat = new DomesticCat();
// load pk's state into cat
sess.load( cat, new Long(pkId) );
Set kittens = cat.getKittens();
```

Notate che `load()` lancerà un'eccezione non recuperabile se non c'è una riga di database corrispondente. Se la classe è mappata con un mediatore (proxy), `load()` restituisce un oggetto che è un mediatore non inizializzato e

non tocca realmente il database finché non viene invocato un metodo dell'oggetto. Questo comportamento è molto utile se volete creare un'associazione ad un oggetto senza realmente caricarlo dal database.

Se non siete certi che una riga corrispondente esista, dovreste usare il metodo `get()`, che va direttamente sul database e ritorna null se la riga non esiste.

```
Cat cat = (Cat) sess.get(Cat.class, id);
if (cat==null) {
    cat = new Cat();
    sess.save(cat, id);
}
return cat;
```

Potete anche caricare un oggetto usando una istruzione SQL `SELECT ... FOR UPDATE`. Leggete la prossima sezione per una discussione dei `LockMode` (modalità di locking) di Hibernate.

```
Cat cat = (Cat) sess.get(Cat.class, id, LockMode.UPGRADE);
```

Notate che ogni istanza associata o le collezioni contenute *non* vengono caricate con una select `FOR UPDATE`.

È possibile ricaricare un oggetto e tutte le sue collezioni in qualsiasi momento, usando il metodo `refresh()`. È una cosa particolarmente utile quando dei trigger del database vengono usati per inizializzare alcune proprietà dell'oggetto.

```
sess.save(cat);
sess.flush(); //forza l'INSERT SQL
sess.refresh(cat); //rilegge lo stato (dopo che il trigger si esegue)
```

9.3. Interrogazioni

Se non conoscete l'identificatore (o gli identificatori) deli oggetti che state cercando, usate i metodi `find()` di `Session`. Hibernate supporta un linguaggio di interrogazione orientato agli oggetti semplice ma potente.

```
List cats = sess.find(
    "from Cat as cat where cat.birthdate = ?",
    date,
    Hibernate.DATE
);

List mates = sess.find(
    "select mate from Cat as cat join cat.mate as mate " +
    "where cat.name = ?",
    name,
    Hibernate.STRING
);

List cats = sess.find( "from Cat as cat where cat.mate.bithdate is null" );

List moreCats = sess.find(
    "from Cat as cat where " +
    "cat.name = 'Fritz' or cat.id = ? or cat.id = ?",
    new Object[] { id1, id2 },
    new Type[] { Hibernate.LONG, Hibernate.LONG }
);

List mates = sess.find(
    "from Cat as cat where cat.mate = ?",
    izi,
    Hibernate.entity(Cat.class)
);

List problems = sess.find(
```

```
    "from GoldFish as fish " +
    "where fish.birthday > fish.deceased or fish.birthday is null"
);
```

Il secondo argomento del metodo `find()` riceve un oggetto o un array di oggetti. Il terzo argomento accetta un tipo di Hibernate o un array di tipi. Questi tipi vengono usati per collegare gli oggetti dati ai segnaposto `?` nelle query JDBC (che a loro volta si mappano su parametri IN di un `PreparedStatement`). Così come nel JDBC, dovreste usare questo meccanismo di associazione preferibilmente alla manipolazione delle stringhe.

La classe `Hibernate` definisce un certo numero di metodi statici e costanti che forniscono accesso alla maggior parte dei tipi predefiniti, sotto forma di istanze della classe `net.sf.hibernate.type.Type`.

Se vi aspettate che la vostra query restituisca un numero di oggetti molto largo, ma non vi aspettate di usarli tutti, potreste ottenere performance migliori dai metodi `iterate()`, che restituiscono un `java.util.Iterator`. L'iteratore caricherà oggetti al bisogno, usando gli identificatori restituiti da una query SQL iniziale (facendo n+1 select totali).

```
// caricamento degli id
Iterator iter = sess.iterate("from eg.Qux q order by q.likeliness");
while ( iter.hasNext() ) {
    Qux qux = (Qux) iter.next();  // reperimento dell'oggetto
    // qualcosa che non abbiamo potuto esprimere nella query
    if ( qux.calculateComplicatedAlgorithm() ) {
        // cancellazione dell'istanza corrente
        iter.remove();
        // non abbiamo bisogno di gestire il resto
        break;
    }
}
```

Sfortunatamente, `java.util.Iterator` non dichiara eccezioni, così qualsiasi eccezione SQL o di Hibernate che capiti viene incapsulata in una `LazyInitializationException` (una sottoclasse di `RuntimeException`).

Il metodo `iterate()` ottiene performance migliori anche nel caso in cui ci si aspetti che molti oggetti siano già stati caricati e messi in cache dalla sessione, o se i risultati della query ottengono molte volte gli stessi oggetti. (Quando non ci sono dati in cache o dati ripetuti, `find()` è quasi sempre più veloce.) Ecco un esempio di un'interrogazione che dovrebbe venire chiamata usando `iterate()`:

```

Iterator iter = sess.iterate(
    "select customer, product " +
    "from Customer customer, " +
    "Product product " +
    "join customer.purchases purchase " +
    "where product = purchase.product"
);
```

Se chiamassimo l'interrogazione precedente usando `find()` restituiremmo un `ResultSet` JDBC molto ampio che conterrebbe gli stessi dati molte volte.

Le query di Hibernate a volte restituiscono tuple di oggetti, nel qual caso ogni tupla viene restituita come un array:

```
Iterator foosAndBars = sess.iterate(
    "select foo, bar from Foo foo, Bar bar " +
    "where bar.date = foo.date"
);
while ( foosAndBars.hasNext() ) {
    Object[] tuple = (Object[]) foosAndBars.next();
    Foo foo = tuple[0]; Bar bar = tuple[1];
    ....
```

```
}
```

9.3.1. Interrogazioni scalari

Le interrogazioni possono specificare una proprietà di una classe nella clausola `select`. Possono anche chiamare funzioni aggregate SQL. Le proprietà o gli aggregati sono considerati risultati "scalari".

```
Iterator results = sess.iterate(
        "select cat.color, min(cat.birthdate), count(cat) from Cat cat " +
        "group by cat.color"
);
while ( results.hasNext() ) {
    Object[] row = results.next();
    Color type = (Color) row[0];
    Date oldest = (Date) row[1];
    Integer count = (Integer) row[2];
    .....
}
```

```
Iterator iter = sess.iterate(
    "select cat.type, cat.birthdate, cat.name from DomesticCat cat"
);
```

```
List list = sess.find(
    "select cat, cat.mate.name from DomesticCat cat"
);
```

9.3.2. L'interfaccia Query

Se avete bisogno di specificare limiti per il vostro set di risultati (il numero massimo di righe che volete recuperare o la prima riga che volete caricare) dovreste ottenere un'istanza di `net.sf.hibernate.Query`:

```
Query q = sess.createQuery("from DomesticCat cat");
q.setFirstResult(20);
q.setMaxResults(10);
List cats = q.list();
```

Potete anche definire una query con un nome nel documento di mappaggio. (Ricordatevi di usare una sezione `CDATA` se la vostra query contiene caratteri che potrebbero essere interpretati come caratteri di contrassegno.)

```
<query name="eg.DomesticCat.by.name.and.minimum.weight"><![CDATA[
    from eg.DomesticCat as cat
        where cat.name = ?
        and cat.weight > ?
] ]></query>
```

```
Query q = sess.getNamedQuery("eg.DomesticCat.by.name.and.minimum.weight");
q.setString(0, name);
q.setInt(1, minWeight);
List cats = q.list();
```

L'interfaccia Query supporta l'utilizzo di parametri per nome. I parametri per nome sono identificatori nella forma `:name` nella stringa di interrogazione. Ci sono metodi su `Query` per collegare valori ai parametri per nome o ai parametri `?` nello stile di JDBC. *Contrariamente a JDBC, Hibernate numera i parametri a partire da zero.* I vantaggi dei parametri per nome sono:

- i parametri per nome non hanno dipendenza dall'ordine con cui appaiono nella stringa di interrogazione

- possono apparire più volte nella stessa query
- sono auto-documentanti

```
//parametro per nome (consigliato)
Query q = sess.createQuery("from DomesticCat cat where cat.name = :name");
q.setString("name", "Fritz");
Iterator cats = q.iterate();
```

```
//parametro posizionale
Query q = sess.createQuery("from DomesticCat cat where cat.name = ?");
q.setString(0, "Izi");
Iterator cats = q.iterate();
```

```
//lista di parametri con nome
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();
```

9.3.3. Iterazioni scrollabili

Se il vostro driver JDBC supporta i `ResultSet` scrollabili, l'interfaccia `Query` può essere usata per ottenere un oggetto `ScrollableResults` che consente una navigazione più flessibile dei risultati dell'interrogazione.

```
Query q = sess.createQuery("select cat.name, cat from DomesticCat cat " +
                           "order by cat.name");
ScrollableResults cats = q.scroll();
if ( cats.first() ) {

    // trova il primo nome su ogni pagina di una lista alfabetica di gatti per nome
    firstNamesOfPages = new ArrayList();
    do {
        String name = cats.getString(0);
        firstNamesOfPages.add(name);
    }
    while ( cats.scroll(PAGE_SIZE) );

    // ora recuperiamo la prima pagina di gatti
    pageOfCats = new ArrayList();
    cats.beforeFirst();
    int i=0;
    while( ( PAGE_SIZE > i++ ) && cats.next() ) pageOfCats.add( cats.get(1) );

}
```

Il comportamento del metodo `scroll()` è simile ad `iterate()`, eccettuato il fatto che gli oggetti possono essere inizializzati selettivamente da `get(int)`, invece di essere inizializzati uno ad uno a righe intere.

9.3.4. Filtraggio delle collezioni

Un *filtro* di collezione è un tipo speciale di interrogazione che può essere applicato ad una collezione persistente od un array. La stringa di interrogazione può fare riferimento a `this`, per indicare l'elemento corrente della collezione.

```
Collection blackKittens = session.filter(
    pk.getKittens(), "where this.color = ?", Color.BLACK, Hibernate.enum(Color.class)
);
```

La collezione restituita viene considerata un "sacco" (bag).

Osservate che i filtri non richiedono una clausola `from` (benché possano averne una, se è necessario). I filtri non si limitano a restituire gli elementi stessi delle collezioni.

```
Collection blackKittenMates = session.filter(
    pk.getKittens(), "select this.mate where this.color = eg.Color.BLACK"
);
```

9.3.5. Interrogazioni per criteri

L'HQL è molto potente, ma alcune persone preferiscono costruire dinamicamente le interrogazioni, usando un'API orientata agli oggetti, piuttosto che inserire stringhe nel loro codice Java. Per queste persone, Hibernate fornisce una API di interrogazione intuitiva per criteri (`Criteria`).

```
Criteria crit = session.createCriteria(Cat.class);
crit.add( Expression.eq("color", eg.Color.BLACK) );
crit.setMaxResults(10);
List cats = crit.list();
```

Se non avete familiarità con le sintassi "simil-SQL", questa è forse la maniera più semplice di approcciare Hibernate. Questa API è poi anche più estensibile dell'SQL: le applicazioni potrebbero fornire le loro implementazioni dell'interfaccia `Criterion`.

9.3.6. Interrogazioni in SQL nativo

È possibile esprimere una query in SQL, usando `createSQLQuery()`. Dovete circondare gli alias SQL di parentesi graffe.

```
List cats = session.createSQLQuery(
    "SELECT {cat.*} FROM CAT AS {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 AS {cat} WHERE ROWNUM<10",
    "cat",
    Cat.class
).list()
```

Le interrogazioni SQL possono contenere parametri per nome e posizionali, proprio come quelle di Hibernate.

9.4. Aggiornamento degli oggetti

9.4.1. Aggiornamento nella stessa Session

Le istanze persistenti e transazionali (cioè gli oggetti caricati, salvati, creati o interrogati dalla `Session`) possono venire manipolati dall'applicazione, ed ogni cambiamento allo stato persistente verrà salvato quando la `Session` viene scaricata (*flushed*) (questo concetto è discusso più oltre in questo stesso capitolo). Quindi, la maniera più semplice di aggiornare lo stato di un oggetto è di caricarlo (`load()`), e poi manipolarlo direttamente mentre la `Session` è aperta:

```
DomesticCat cat = (DomesticCat) sess.load( Cat.class, new Long(69) );
```

```
cat.setName("PK");
sess.flush();  // i cambiamenti all'oggetto gatto vengono automaticamente
               // individuati e resi persistenti
```

Alcune volte questo modello di programmazione è inefficiente, poiché richiederebbe sia una SELECT SQL (per caricare un oggetto) sia una UPDATE (per rendere persistente il suo stato aggiornato) nella stessa sessione. Per questo, Hibernate offre un approccio alternativo.

9.4.2. Aggiornamento di oggetti sganciati

Molte applicazioni hanno bisogno di recuperare un oggetto in una transazione, mandarlo allo strato di interfaccia perché venga manipolato, e poi salvarne i cambiamenti in una nuova transazione. (Le applicazioni che usano questo genere di approccio in un ambiente ad alta concorrenza solitamente usano dati versionati per assicurare l'isolamento delle transazioni.) Questo approccio richiede un modello programmativo leggermente differente rispetto a quello descritto nell'ultima sezione. Hibernate supporta questo modello fornendo il metodo `Session.update()`.

```
// nella prima sessione
Cat cat = (Cat) firstSession.load(Cat.class, catId);
Cat potentialMate = new Cat();
firstSession.save(potentialMate);

// in uno strato più elevato dell'applicazione
cat.setMate(potentialMate);

// più tardi, in un'altra sessione
secondSession.update(cat);  // aggiornamento del gatto (cat)
secondSession.update(mate); // aggiornamento dell'amichetto (mate)
```

Se l'oggetto `Cat` con identificatore `catId` fosse già stato caricato da `secondSession` quando l'applicazione tenta di aggiornarlo, verrebbe lanciata un'eccezione.

L'applicazione dovrebbe aggiornare (`update()`) istanze transienti raggiungibili dall'istanza transiente data se e *solo* se vuole che anche il loro stato venga aggiornato. (Eccetto per gli oggetti a ciclo di vita, discussi più avanti).

Gli utenti di Hibernate hanno chiesto un metodo di scopo generale che salvi un'istanza transiente generando un nuovo identificatore o aggiorni lo stato persistente associato con il suo identificatore corrente. Il metodo `saveOrUpdate()` adesso implementa questa funzionalità.

Hibernate distingue istanze "nuove" (non salvate) da istanze "esistenti" (salvate o caricate in una sessione precedente) tramiet il valore della loro proprietà identificatore (o versione, o marca di tempo). L'attributo `unsaved-value` degli elementi `<id>` (o `<version>`, o `<timestamp>`) nel mappaggio specifica quali valori dovrebbero venire interpretati come rappresentanti di una "nuova" istanza.

```
<id name="id" type="long" column="uid" unsaved-value="null">
    <generator class="hilo"/>
</id>
```

I valori consentiti di `unsaved-value` sono:

- `any` - salvare sempre
- `none` - aggiornare sempre
- `null` - salvare quando l'identificatore è nullo (questa è l'opzione predefinita)
- valid identifier value - salvare quando l'identificatore è nullo o il valore dato
- `undefined` - il valore predefinito per `version` o `timestamp`, viene usato un controllo sull'identificatore

```
// nella prima sessione
Cat cat = (Cat) firstSession.load(Cat.class, catID);

// in uno strato più elevato dell'applicazione
Cat mate = new Cat();
cat.setMate(mate);

// più avanti, in un'altra sessione
secondSession.saveOrUpdate(cat);   // aggiorna lo stato esistente (cat ha un id non nullo)
secondSession.saveOrUpdate(mate);  // salva la nuova istanza (mate ha un id nullo)
```

L'utilizzo e la semantica di `saveOrUpdate()` sembra confondere i nuovi utenti. In primo luogo, finché non stiate cercando di usare istanze di una sessione in un'altra sessione, non dovreste aver bisogno di usare `update()` o `saveOrUpdate()`. Alcune applicazioni non avranno mai bisogno di nessuno di questi due metodi.

Di solito `update()` o `saveOrUpdate()` vengono usati nello scenario seguente:

- l'applicazione carica un oggetto nella prima sessione
- l'oggetto viene passato allo strato di interfaccia
- vengono fatte alcune modifiche all'oggetto
- l'oggetto viene ripassato allo strato della logica di business
- l'applicazione rende persistenti queste modifiche chiamando `update()` in una seconda sessione

`saveOrUpdate()` fa quanto segue:

- se l'oggetto è già persistente in questa sessione, non fare nulla
- se l'oggetto non ha proprietà identificatore, lo salva (`save()`)
- se il valore dell'identificatore salva i criteri specificati da `unsaved-value`, lo salva (`save()`)
- se l'oggetto è con versioni (`version` o `timestamp`), allora la versione avrà precedenza sul controllo dell'identificatore, a meno che la versione non sia `unsaved-value="undefined"` (valore di default)
- se un altro oggetto associato con la sessione ha lo stesso identificatore, lancia un'eccezione

9.4.3. Riaggancio di oggetti sganciati

Il metodo `lock()` consente all'applicazione di riassociare un oggetto non modificato con una nuova sessione.

```
//riassocia semplicemente:
sess.lock(fritz, LockMode.NONE);
//fa un controllo di versione, poi riassocia:
sess.lock(izi, LockMode.READ);
//fa un controllo di versione usando SELECT ... FOR UPDATE, quindi riassocia:
sess.lock(pk, LockMode.UPGRADE);
```

9.5. Cancellazione di oggetti persistenti

`Session.delete()` rimuoverà lo stato di un oggetto dal database. Naturalmente, la vostra applicazione potesse ancora mantenere un riferimento ad esso. Per questo, è preferibile pensare a `delete()` come un modo per rendere transiente un'istanza persistente.

```
sess.delete(cat);
```

Potete anche cancellare molti oggetti allo stesso tempo, passando una stringa di interrogazione a `delete()`.

Ora è poi possibile cancellare oggetti in qualsiasi ordine, senza il rischio di violazioni di chiave. Naturalmente, è sempre possibile violare un vincolo `NOT NULL` su una chiave esterna cancellando oggetti nell'ordine sbagliato.

9.6. Scaricamento (flush)

Di tanto in tanto, la `Session` eseguirà le istruzioni SQL necessarie per sincronizzare lo stato della connessione JDBC con lo stato degli oggetti mantenuti in memoria. Questo processo, il *flush*, avviene come comportamento standard nei seguenti punti

- per effetto di alcune invocazioni di `find()` o `iterate()`
- in seguito a `net.sf.hibernate.Transaction.commit()`
- in seguito a `Session.flush()`

Le istruzioni SQL vengono emesse nell'ordine seguente

1. tutti gli inserimenti di entità, nello stesso ordine con cui gli oggetti corrispondenti erano stati salvati usando `Session.save()`
2. tutti gli aggiornamenti di entità
3. tutte le cancellazioni di collezione
4. tutte le cancellazioni, gli aggiornamenti e inserimenti di elementi di collezioni
5. tutti gli inserimenti di collezione
6. tutte le cancellazioni di entità, nello stesso ordine con cui gli oggetti corrispondenti erano stati cancellati usando `Session.delete()`

(Una eccezione è che gli oggetti che usando meccanismi di generazione di identificator `native` vengono inseriti nel momento stesso in cui sono salvati.)

Eccettuato quando chiamate `flush()` esplicitamente, non ci sono assolutamente garanzie riguardo a *quando* la `Session` eseguirà le chiamate JDBC, solo l'*ordine* con cui verranno eseguite. In ogni caso, Hibernate garantisce che i metodi `Session.find(..)` non restituiranno mai dati obsoleti o che restituiranno dati sbagliati.

È possibile cambiare il comportamento predefinito in modo che lo scaricamento avvenga meno frequentemente. La classe `FlushMode` definisce tre modi differenti. Questo non è utile nel caso di transazioni "a sola lettura", in cui potrebbe solo essere usato per ottenere un (molto) leggero incremento di performance.

```
sess = sf.openSession();
Transaction tx = sess.beginTransaction();
//consente alle interrogazioni di restituire stato obsoleto
sess.setFlushMode(FlushMode.COMMIT);
Cat izi = (Cat) sess.load(Cat.class, id);
izi.setName(iznizi);
// esegue alcune interrogazioni....
sess.find("from Cat as cat left outer join cat.kittens kitten");
//il cambiamento su izi non viene scaricato!
...
tx.commit(); //avviene lo scaricamento
```

9.7. Fine di una sessione

La conclusione di una sessione implica quattro fasi distinte:

- scaricamento della sessione
- commit della transazione
- chiusura della sessione
- gestione delle eccezioni

9.7.1. Scaricamento della sessione

Se state usando l'API `Transaction`, non avete bisogno di preoccuparvi di questo passo. Verrà effettuato implicitamente quando la transazione verrà committata. In caso contrario dovreste chiamare `Session.flush()` per assicurarvi che tutti i cambiamenti siano sincronizzati con il database.

9.7.2. Commit della transazione sul database

Se state usando l'API `Transaction` di Hibernate, questo apparirà come:

```
tx.commit(); // scaricamento della Session e commit della transazione
```

Se state gestendo autonomamente le transazioni JDBC, dovreste chiamare manualmente `commit()` sulla connessione JDBC.

```
sess.flush();
sess.connection().commit();  // non necessario per un datasource JTA
```

Se decidete di *non* fare il commit dei cambiamenti:

```
tx.rollback();  // rollback della transazione
```

or:

```
// non necessario per un datasource JTA, importante altrimenti
sess.connection().rollback();
```

Se fate il rollback della transazione dovreste immediatamente chiudere e scartare la sessione corrente, per assicurarvi che lo stato interno di Hibernate sia coerente.

9.7.3. Chiusura della sessione

Una chiamata a `Session.close()` segna la fine di una sessione. L'implicazione principale di `close()` è che la connessione JDBC verrà liberata dalla session.

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

```
sess.flush();
sess.connection().commit();  // non necessario per un datasource JTA
sess.close();
```

Se avete fornito le vostre connessioni, `close()` restituisce un riferimento ad esse, in modo tale che possiate chiuderle o restituirle al lotto (pool) manualmente. In caso contrario `close()` le restituisce al lotto.

9.7.4. Gestione delle eccezioni

Se la `Session` lancia un'eccezione (compresa una qualsiasi `SQLException`), dovreste immediatamente fare il rollback della transazione, chiamare `Session.close()` e scartare l'istanza di `Session`. Alcuni metodi di `Session` *non* lasceranno la sessione in uno stato coerente.

Raccomandiamo il seguente idioma di gestione delle eccezioni:

```
Session sess = factory.openSession();
Transaction tx = null;
try {
```

```
    tx = sess.beginTransaction();
    // fa del lavoro
    ...
    tx.commit();
}
catch (Exception e) {
    if (tx!=null) tx.rollback();
    throw e;
}
finally {
    sess.close();
}
```

O, quando si gestiscono manualmente le transazioni JDBC:

```
Session sess = factory.openSession();
try {
    // fa del lavoro
    ...
    sess.flush();
    sess.connection().commit();
}
catch (Exception e) {
    sess.connection().rollback();
    throw e;
}
finally {
    sess.close();
}
```

O, quando si usa un datasource iscritto con il JTA:

```
UserTransaction ut = .... ;
Session sess = factory.openSession();
try {
    // fa del lavoro
    ...
    sess.flush();
}
catch (Exception e) {
    ut.setRollbackOnly();
    throw e;
}
finally {
    sess.close();
}
```

9.8. Cicli di vita e grafi di oggetti

Per salvare o aggiornare tutti gli oggetti in un grafo di oggetti associati dovete

- chiamare `save()`, `saveOrUpdate()` o `update()` su ogni oggetto individuale O
- mappare gli oggetti associati con `cascade="all"` o `cascade="save-update"`.

Nello stesso modo, per cancellare tutti gli oggetti un grafo

- chiamate `delete()` su ogni oggetto individuale O
- mappate gli oggetti associati usando `cascade="all"`, `cascade="all-delete-orphan"` o `cascade="delete"`.

Raccomandazione:

- Se il periodo di esistenza dell'oggetto figlio è incluso in quello del genitore fate in modo che diventi un oggetto del ciclo di vita *(lifecycle object)*, specificando `cascade="all"`.
- In caso contrario, chiamate esplicitamente `save()` e `delete()` su di esso dal codice applicativo. Se volete davvero risparmiarvi della digitazione extra, usate `cascade="save-update"` e chiamate esplicitamente `delete()`.

Il mappaggio di un'associazione (molti-a-uno, o collezione) con `cascade="all"` marchia l'associazione come una relazione di stile *genitore/figlio* in cui il salvataggio/aggiornamento/cancellazione del genitore risulta in operazioni analoghe dei figli. Inoltre, il semplice fatto di avere un riferimento ad un figlio in un genitore persistente risulterà in salvataggio / aggiornamento del figlio. La metafora è tuttavia incompleta. Un figlio che diventi non più referenziato dal padre *non* viene automaticamente cancellato, eccetto nel caso di una associazione `<one-to-many>` mappata con `cascade="all-delete-orphan"`. La semantica precisa delle operazioni di cascata sono come segue:

- Se un parente viene salvato, tutti i figli vengono passati a `saveOrUpdate()`
- Se un parente viene passato a `update()` o `saveOrUpdate()`, tutti i figli vengono passati a `saveOrUpdate()`
- Se un figlio transiente diventa referenziato da un padre persistente, viene passato a `saveOrUpdate()`
- Se un parente viene cancellato, tutti i figli vengono passati a `delete()`
- Se un figlio transiente viene de-referenziato da un padre persistente, *non succede niente di speciale* (l'applicazione dovrebbe cancellare esplicitamente il figlio se necessario) a meno che la relazione non sia mappata come come `cascade="all-delete-orphan"`, nel qual caso il figlio "orfano" viene cancellato.

Hibernate non implementa completamente il concetto di "persistenza per raggiungibilità", che implicherebbe una (inefficiente) raccolta dei rifiuti (garbage collection) persistenti. In ogni caso, in seguito alle richieste di molta gente, Hibernate supporta la nozione di entità che diventano persistenti quando vengono referenziate da un altro oggetto persistente. Le associazioni marchiate `cascade="save-update"` si comportano così. Se volete usare questo approccio in tutta l'applicazione, è più comodo specificare l'attributo `default-cascade` nell'elemento `<hibernate-mapping>`.

9.9. Intercettatori (interceptors)

L'interfaccia `Interceptor` fornisce dei punti di richiamo (callback) dalla sessione verso l'applicazione, consentendole di ispezionare e/o manipolare proprietà di un oggetto persistente prima che venga salvato, aggiornato, cancellato o caricato. Un utilizzo possibile per questo è di tracciare delle informazioni di auditing. Per esempio, l'`Interceptor` seguente imposta automaticamente il `createTimestamp` quando viene creato un oggetto `Auditable` e aggiorna la proprietà `lastUpdateTimestamp` quando un `Auditable` viene aggiornato.

```
package net.sf.hibernate.test;

import java.io.Serializable;
import java.util.Date;
import java.util.Iterator;

import net.sf.hibernate.Interceptor;
import net.sf.hibernate.type.Type;

public class AuditInterceptor implements Interceptor, Serializable {

    private int updates;
    private int creates;

    public void onDelete(Object entity,
                         Serializable id,
                         Object[] state,
                         String[] propertyNames,
                         Type[] types) {
        // non fare nulla
    }
```

```
    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) {
        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("Creazioni: " + creates + ", Aggiornamenti: " + updates);
    }

    public void preFlush(Iterator entities) {
        updates=0;
        creates=0;
    }

    ......
    ......

}
```

L'interceptor dovrebbe essere specificato alla creazione di una sessione.

```
Session session = sf.openSession( new AuditInterceptor() );
```

9.10. API dei metadati

Hibernate richiede un modello di meta-livello molto ricco di tutte le entità e dei tipi di valori. Di tanto in tanto, questo modello è molto utile alla stessa applicazione. Ad esempio, l'applicazione potrebbe usare i metadati di Hibernate per implementare un algoritmo "intelligente" di copia "profonda" che capisca quali oggetti dovrebbero venire copiati (ad esempio i tipi di valore mutabili) e quali non dovrebbero (ad esempio i tipi di valore immutabili e, magari, le entità associate).

Hibernate espone metadati tramite le interfacce `ClassMetadata` e `CollectionMetadata` e la gerarchia `Type`. Le istanze delle interfacce dei metadati possono venire ottenute dalla `SessionFactory`.

```
Cat fritz = ......;
Long id = (Long) catMeta.getIdentifier(fritz);
ClassMetadata catMeta = sessionfactory.getClassMetadata(Cat.class);
Object[] propertyValues = catMeta.getPropertyValues(fritz);
String[] propertyNames = catMeta.getPropertyNames();
Type[] propertyTypes = catMeta.getPropertyTypes();
// ottiene una mappa di tutte le proprietà che non sono collezioni o associazioni
// TODO: e i componenti?
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] );
    }
}
```

Capitolo 10. Transazioni e concorrenza

Hibernate non è in se stesso un database. È uno strumento leggero di mappaggio oggetto-relazione. La gestione delle transazioni viene delegata alla sottostante connessione con il database. Se la connessione è iscritta con il JTA, le operazioni effettuate dalla `Session` sono atomicamente parte della transazione JTA più esterna. Hibernate può essere considerato un sottile strato di adattamento sul JDBC che aggiunge la semantica orientata agli oggetti.

10.1. Configurazioni, sessioni e "factory"

Una `SessionFactory` è un oggetto che supporta l'utilizzo concorrente (threadsafe), costoso da creare, che è pensato per essere condiviso da tutti i thread dell'applicazione. Una `Session` è invece un oggetto non costoso da crearsi, non utilizzabile in maniera concorrente, che dovrebbe essere usato una volta sola per un singolo processo di business e poi scartato. Ad esempio, quando si usa Hibernate in un applicazione basata sui servlet, i servlet possono ottenere una `SessionFactory` usando

```
SessionFactory sf = (SessionFactory)getServletContext().getAttribute("my.session.factory");
```

Ogni chiamata ad un metodo di servizio potrebbe creare una nuova `Session`, farci sopra il `flush()` (scaricamento su db) , mandare un `commit()` sulla sua connesione, chiuderla (`close()`) ed infine eliminarla. (La `SessionFactory` può anche essere memorizzata nel JNDI o in una variabile di utilità *Singleton*.)

In un "session bean" senza stato si può usare un approccio simile. Il bean dovrebbe ottenere una `SessionFactory` con il metodo `setSessionContext()`. A questo punto, ogni metodo di business dovrebbe creare una `Session`, fare il `flush()` e chiuderla (`close()`). Naturalmente, l'applicazione non dovrebbe chiamare `commit()` sulla connessione. (Va lasciata al JTA, perché la connessione al database partecipa automaticamente nelle transazioni gestite dal contenitore.)

Usiamo l'API `Transaction` di Hibernate come discusso in precedenza, una singola `commit()` di una `Transaction` di Hibernate scarica lo stato e fa il commit di ogni connessione di database sottostante (con una gesione particolare delle transazioni JTA).

Assicuratevi di capire la semantica del `flush()`. Lo scaricamento (flushing) sincronizza il contenitore persistente con i cambiameti in memoria, ma *non* vice-versa. Notate che per tutte le connessioni/transazioni JDBC di Hibernate, il livello di isolamento transazionale si applica a tutte le operazioni che vengono eseguite da Hibernate stesso!

Le prossime sezioni discuteranno gli approcci alternativi che usano il versionamento per assicurare l'atomicità delle transazioni. Sono approcci che vengono considerati "avanzati" e vanno usati con attenzione.

10.2. Thread e connessioni

Dovreste osservare le indicazioni seguenti quando create Session di Hibernate:

- Non creare più di una istanza di `Session` o `Transaction` concorrenti per connessione di database.
- Siate estremamente attenti quando create più di una `Session` per database per transazione. La `Session` mantiene traccia di aggiornamenti fatti agli oggetti caricati, e quindi una `Session` differente potrebbe vedere dati non più validi.
- La `Session` *non* è threadsafe (non consente più utilizzi concorrenti)! Non accedete alla stessa `Session` in due thread di esecuzione concorrenti. Una `Session` di solito è una singola unità di lavoro!

10.3. Considerazioni sull'identità degli oggetti

L'applicazione può accedere concorrentemente allo stesso stato persistente in due differenti unità di lavoro. Però, un'istanza di una classe persistente non viene mai condivisa tra due istanze di `Session`. Da qui, discendono due differenti nozioni di identità:

Identità per il database
```
foo.getId().equals( bar.getId() )
```

Identità per la JVM (java virtual machine)
```
foo==bar
```

Per due oggetti appartenenti ad una *particulare* `Session`, le due nozioni sono equivalenti. Però, mentre l'applicazione potrebbe accedere in maniera concorrente lo "stesso" (secondo l'identità persistente) oggetto di business in due sessioni differenti, le due istanze sono in realtà "differenti" (secondo l'identità della virtual machine).

Questo approccio fa sì che siano Hibernate e il database, a preoccuparsi della concorrenza. L'applicazione non deve mai sincronizzare l'accesso ad un oggetto di business, finché rispetta il fatto che l'accesso alla `Session` venga fatto da un singolo thread o le regole sull'identità degli oggetti (all'interno di una `Session` l'applicazione può tranquillamente utilizzare == per confrontare gli oggetti).

10.4. Controllo di concorrenza ottimistico

Molti processi di business richiedono una serie di interazioni con l'utente inframmezzate da accessi al database. Nelle applicazioni web e aziendali non è accettabile che una transazione sul database si estenda lungo una serie di interazioni con l'utente.

Mantenere l'isolamento dei processi di business in questi casi diventa una responsabilità parziale dello strato applicativo, ed in questo caso si dice che questo processo è una *transazione applicativa* di lunga durata. Una singola transazione applicativa di solito si estende su diverse transazioni sul database: essa sarà atomica se una sola di queste transazioni sul database (l'ultima) memorizza i dati aggiornati, e le altre semplicemente li leggono.

L'unico approccio che è consistente con alta concorrenza e alta scalabilità è il controllo di concorrenza ottimistico con versionamento. Hibernate fornisce tre possibili approcci alla produzione di codice applicativo che utilizzi la concorrenza ottimistica.

10.4.1. Sessione lunga con versionamento automatico

Una singola istanza di `Session` e gli oggetti persistenti che gestisce sono utilizzate per tutta la transazione applicativa.

La `Session` utilizza il locking ottimistico con versionamento per assicurarsi che molte transazioni sul database appaiano all'applicazione come una singola transazione applicativa logica. La `Session` è disconnessa dalla connessione JDBC mentre aspetta l'interazione con l'utente. Questo approccio è il più efficiente in termini di accesso al database. L'applicazione non deve preoccuparsi con il controllo delle versioni o con il riaggancio alla sessione delle istanze sganciate.

```
// foo è un'istanza caricata precedentemente dalla Session
session.reconnect();
```

```
foo.setProperty("bar");
session.flush();
session.connection().commit();
session.disconnect();
```

L'oggetto `foo` sa ancora da quale `Session` era stata caricato. Non appena la `Session` avrà una connessione JDBC verrà fatto il commit dei cambiamenti sull'oggetto.

Questo pattern è problematico se la `Session` è troppo grande per essere memorizzata durante il tempo di ragionamento dell'utente, ad esempio una `HttpSession` dovrebbe essere mantenuta il più ridotta possibile. Poiché la `Session` è anche la cache di primo livello (obbligatoria) e contiene tutti gli oggetti che ha caricato, possiamo probabilmente utilizzare questa strategia solo per pochi cicli di richiesta e risposta. Questo è in realtà raccomandato anche perché la `Session` avrebbe presto dati scaduti, in caso contrario.

10.4.2. Sessioni multiple con versionamento automatico

Ogni interazione con il contenitore persistente dei dati avviene in una nuova `Session`. Però, le stesse istanze persistenti vengono riutilizzate per ogni interazione con il database. L'applicazione manipola lo stato delle istanze sganciate originariamente caricate in un'altra `Session` e quindi le "riassocia" usando `Session.update()` o `Session.saveOrUpdate()`.

```
// foo è una istanza caricata da una Session precedente
foo.setProperty("bar");
session = factory.openSession();
session.saveOrUpdate(foo);
session.flush();
session.connection().commit();
session.close();
```

È anche possibile chiamare `lock()` invece di `update()` e usare `LockMode.READ` (effettua un controllo di versione e aggira tutte le cache) se si è sicuri che l'oggetto non è stato modificato.

10.4.3. Controllo delle versioni da parte dell'applicazione

Ogni interazione con il database avviene in una nuova `Session` che ricarica tutte le istanze persistenti prima di manipolarle. Questo approccio obbliga l'applicazione a gestire in proprio il controllo delle versioni per assicurarsi che le transazioni applicative siano isolate. (Naturalmente Hibernate *aggiornerà* ancora i numeri di versione per voi). Questo approccio è il meno efficiente in termini di accesso al database, ed è il più simile a quello degli EJB di entità.

```
// foo è un'istanza caricata da una Session precedente
session = factory.openSession();
int oldVersion = foo.getVersion();
session.load( foo, foo.getKey() );
if ( oldVersion!=foo.getVersion ) throw new StaleObjectStateException();
foo.setProperty("bar");
session.flush();
session.connection().commit();
session.close();
```

Naturalmente, se state lavorando in un ambiente a bassa concorrenza dei dati e non avete bisogno di controllo delle versioni, potete adottare questo approccio e semplicemente evitare il controllo di versione.

10.5. Disconnessione della sessione

Il primo approccio descritto sopra è di mantenere una singola `Session` che si estende per un intero processo di business durante il periodo di ragionamento dell'utente. (Ad esempio, un servlet potrebbe mantenere una `Session` nella `HttpSession` dell'utente.) Per ragioni di performance si dovrebbe

1. fare il commit della `Transaction` (o della connessione JDBC) e poi
2. sconnettere la `Session` dalla connessione JDBC

prima di aspettare un'azione da parte dell'utente. Il metodo `Session.disconnect()` sconnetterà la sessione dalla connessione JDBC e restituirà la connessione al lotto di connessioni disponibili per l'uso (a meno che non siate stati voi a fornirla direttamente).

`Session.reconnect()` ottiene una nuova connessione (o potete fornirne una voi) e fa ripartire la sessione. Dopo la riconnessione, è possibile chiamare `Session.lock()` per forzare un controllo di versione sui dati che non sono stati modificati ma che potrebbero essere stati aggiornati da un'altra transazione. Non avete bisogno di porre dei "lock" su dati che *state* modificando.

Ecco un esempio:

```
SessionFactory sessions;
List fooList;
Bar bar;
....
Session s = sessions.openSession();

Transaction tx = null;
try {
    tx = s.beginTransaction();

    fooList = s.find(
            "select foo from eg.Foo foo where foo.Date = current date"
        // uses db2 date function
    );
    bar = (Bar) s.create(Bar.class);

    tx.commit();
}
catch (Exception e) {
    if (tx!=null) tx.rollback();
    s.close();
    throw e;
}
s.disconnect();
```

In seguito:

```
s.reconnect();

try {
    tx = s.beginTransaction();

    bar.setFooTable( new HashMap() );
    Iterator iter = fooList.iterator();
    while ( iter.hasNext() ) {
        Foo foo = (Foo) iter.next();
        s.lock(foo, LockMode.READ);    //controlliamo che foo non sia scaduto
        bar.getFooTable().put( foo.getName(), foo );
    }

    tx.commit();
}
catch (Exception e) {
    if (tx!=null) tx.rollback();
    throw e;
}
finally {
```

```
    s.close();
}
```

Potete vedere da quanto precede che la relazione tra `Transactioni` e `Sessioni` è molti-a-uno. Una `Session` rappresenta una conversazione tra l'applicazione e il database. La `Transaction` spezza quella conversazione in unità di lavoro atomiche al livello del database.

10.6. Locking Pessimistico

Gli utenti non devono spendere molto tempo preoccupandosi delle strategie di locking. Solitamente è sufficiente specificare un livello di isolamento per le connessioni JDBC e poi semplicemente fare in modo che il database faccia tutto il lavoro. Però, gli utenti avanzati possono desiderare a volte di ottenere lock pessimistici esclusivi, o riottenere dei lock all'inizio di una nuova transazione.

Hibernate userà sempre il meccanismo di lock del database, e non porrà mai dei lock sugli oggetti in memoria!

La classe `LockMode` definisce i differenti livelli di lock che possono essere acquisiti da Hibernate. Un lock si può ottenere con i meccanismi seguenti:

- `LockMode.WRITE` viene assunto automaticamente quando Hibernate modifica o inserisce una riga.
- `LockMode.UPGRADE` può essere acquisito in seguito ad una richiesta esplicita dell'utente utilizzando `SELECT ... FOR UPDATE` su dei database che supportino questa sintassi.
- `LockMode.UPGRADE_NOWAIT` può essere acquisito in seguito ad una richiesta esplicita dell'utente usando `SELECT ... FOR UPDATE NOWAIT` in Oracle.
- `LockMode.READ` viene acquisito automaticamente quando Hibernate legge dati a livello di isolamento pari a "Repeatable Read" (letture ripetibili) o "Serializable". Può essere acquisito anche per esplicita richiesta dell'utente.
- `LockMode.NONE` rappresenta una situazione di assenza di lock. Tutti gli oggetti si portano in questa modalità di lock alla fine di una `Transaction`. Gli oggetti associati con la session tramite una chiamata a `update()` o `saveOrUpdate()` vengono avviati in questa modalità.

La "richiesta esplicita dell'utente" viene espressa in una delle modalità seguenti:

- Una chiamata a `Session.load()`, specificando un `LockMode`.
- Una chiamata a `Session.lock()`.
- Una chiamata a `Query.setLockMode()`.

Se si chiama `Session.load()` con `UPGRADE` o `UPGRADE_NOWAIT`, e l'oggetto richiesto non era ancora stato caricato dalla sessione, l'oggetto viene caricato usando `SELECT ... FOR UPDATE`. Se si chiama `load()` per un oggetto che è già stato caricato con un lock meno restrittivo di quello che è stato richiesto, Hibernate chiama `lock()` per quell'oggetto.

`Session.lock()` effettua un controllo del numero di versione se la modalità di lock è `READ`, `UPGRADE` o `UPGRADE_NOWAIT`. (Nel caso di `UPGRADE` o `UPGRADE_NOWAIT`, viene usato `SELECT ... FOR UPDATE`.)

Se il database non supporta la modalità di lock richiesta, Hibernate userà la modalità alternativa più appropriata (invece di lanciare un'eccezione). Questo fa sì che le applicazioni risultino portabili.

Capitolo 11. HQL: Il linguaggio di interrogazione di Hibernate (Hibernate Query Language)

Hibernate è dotato di un linguaggio di interrogazione estremamente potente che (del tutto intenzionalmente) assomiglia molto all'SQL. Ma la sintassi non deve ingannare: l'HQL è pienamente orientato agli oggetti, e comprende nozioni come l'ereditarietà, il polimorfismo e l'associazione.

11.1. Dipendenza da maiuscole e minuscole

Le interrogazioni non distinguono tra maiuscole e minuscole, eccetto per i nomi delle classi java e delle proprietà. Quindi `SeLeCT` è la stessa cosa di `sELEct` che è la stessa cosa di `SELECT` ma `net.sf.hibernate.eg.FOO` non è `net.sf.hibernate.eg.Foo` e `foo.barSet` non è `foo.BARSET`.

Questo manuale fa uso di parole chiave HQL in lettere minuscole. Alcuni utenti trovano che le interrogazioni con parole chiave in maiuscolo siano più leggibili, ma troviamo che questa convenzione sia brutta, quando utilizzata in interrogazioni annidate in codice java.

11.2. La clausola from

L'interrogazione più semplice possibile in Hibernate ha la forma:

```
from eg.Cat
```

che restituisce semplicemente tutte le istanze della classe `eg.Cat`.

La maggior parte delle volte, avrete bisogno di assegnare un *sinonimo*, poiché vorrete fare riferimento al `Cat` in altre parti dell'interrogazione.

```
from eg.Cat as cat
```

Questa query assegna il sinonimo `cat` alle istanze di `Cat`, in modo da poter usare quel sinonimo più avanti nell'interrogazione. La parola chiave `as` è opzionale, potremmo anche scrivere:

```
from eg.Cat cat
```

Possono apparire anche classi multiple, il che risulta in un prodotto cartesiano o join "incrociato".

```
from Formula, Parameter
```

```
from Formula as form, Parameter as param
```

Viene considerata una buona abitudine dare ai sinonimi delle interrogazioni nomi che comincino con lettere minuscole, in maniera coerente con gli standard di denominazione di Java per le variabili locali (ad esempio `domesticCat`).

11.3. Associazioni e join

Possiamo anche assegnare sinonimi ad entità associate, o anche ad elementi di una collezione di valori, usando

un `join`.

```
from eg.Cat as cat
    inner join cat.mate as mate
    left outer join cat.kittens as kitten

from eg.Cat as cat left join cat.mate.kittens as kittens

from Formula form full join form.parameter param
```

I tipi di join supportati sono presi in prestito dall'SQL ANSI

- `inner join`
- `left outer join`
- `right outer join`
- `full join` (di solito inutile)

I costrutti `inner join`, `left outer join` e `right outer join` possono venire abbreviati.

```
from eg.Cat as cat
    join cat.mate as mate
    left join cat.kittens as kitten
```

In aggiunta, un join di tipo "fetch" (raccolta) consente di inizializzare le associazioni o le collezioni insieme agli oggetti genitori, usando una singola select. Questo è particolarmente utile nel caso di una collezione. Sovrascrive in maniera efficace le dichiarazioni dei join esterni (outer join) e della raccolta differita (lazy) del file di mappaggio per le associazioni e le collezioni.

```
from eg.Cat as cat
    inner join fetch cat.mate
    left join fetch cat.kittens
```

Un "fetch join" (join con raccolta) non ha solitamente bisogno di assegnare un sinonimo, perché gli oggetti associati non dovrebbero venire usati nella clausola `where` (né in un'altra clausola qualsiasi). Nello stesso modo, gli oggetti associati non vengono restituiti direttamente nei risultati della query. Possono, invece, essere raggiunti tramite l'oggetto genitore

Notate che, nell'implementazione corrente, solo un ruolo di collezione può essere concretizzato ("fetched") in una interrogazione (qualsiasi altra cosa non sarebbe performante). Notate anche che il costrutto `fetch` non può essere usato in interrogazioni chiamate usando `scroll()` o `iterate()`. Notate infine che `full join fetch` e `right join fetch` non hanno senso.

11.4. La clausola select

La clausola `select` sceglie quali oggetti e proprietà vanno restituiti nel set dei risultati della query. Considerate che:

```
select mate
from eg.Cat as cat
    inner join cat.mate as mate
```

La query selezionerà gli amici (`mates`) dei gatti (`Cats`). In realtà è possibile esprimere la stessa interrogazione in maniera più compatta come:

```
select cat.mate from eg.Cat cat
```

Potete anche selezionare elementi di una collezione, usando la funzione speciale `elements`. L'interrogazione seguente restituisce tutti i gattini (kittens) di ogni gatto (cat).

```
select elements(cat.kittens) from eg.Cat cat
```

Le interrogazioni possono restituire proprietà di qualsiasi tipo di valore, comprese le proprietà di tipo componente:

```
select cat.name from eg.DomesticCat cat
where cat.name like 'fri%'

select cust.name.firstName from Customer as cust
```

Le interrogazioni possono restituire oggetti multipli e/o proprietà come un array di tipo `Object[]`

```
select mother, offspr, mate.name
from eg.DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

o come un oggetto java tipizzato

```
select new Family(mother, mate, offspr)
from eg.DomesticCat as mother
    join mother.mate as mate
    left join mother.kittens as offspr
```

purché ovviamente `Family` abbia un costruttore appropriato.

11.5. Funzioni aggregate

Le query HQL possono anche restituire i risultati di funzioni aggregate sulle proprietà:

```
select avg(cat.weight), sum(cat.weight), max(cat.weight), count(cat)
from eg.Cat cat
```

Le collezioni possono anche apparire all'interno di funzioni aggregate nella clausola `select`.

```
select cat, count( elements(cat.kittens) )
from eg.Cat cat group by cat
```

Le funzioni aggregate supportate sono

- `avg(...), sum(...), min(...), max(...)`
- `count(*)`
- `count(...), count(distinct ...), count(all...)`

Le parole chiave `distinct` e `all` possono essere usate con la stessa semantica dell'SQL.

```
select distinct cat.name from eg.Cat cat

select count(distinct cat.name), count(cat) from eg.Cat cat
```

11.6. Interrogazioni polimorfiche

Una interrogazione come:

```
from eg.Cat as cat
```

non restituisce solo istanze di `Cat`, ma anche delle sottoclassi come `DomesticCat`. Le interrogazioni di Hibernate possono indicare *qualsiasi* classe o interfaccia Java nella clausola `from`. L'interrogazione restituirà istanze di tutte le classi persistenti che estendono quella classe o implementano l'interfaccia. La prossima interrogazione restituisce tutti gli oggetti persistenti:

```
from java.lang.Object o
```

L'interfaccia `Named` potrebbe essere implementata da diverse classi persistenti:

```
from eg.Named n, eg.Named m where n.name = m.name
```

Notate che queste ultime due interrogazioni richiederanno più di una `SELECT` SQL. Questo significa che la clausola `order by` non ordinerà correttamente l'intero insieme dei risultati. (e significa anche che non potete chiamare le query usando `Query.scroll()`.)

11.7. La clausola where

La clausola `where` consente di limitare la lista di istanze rese da una interrogazione.

```
from eg.Cat as cat where cat.name='Fritz'
```

restituisce le istanze di `Cat` il cui nome (name) è 'Fritz'.

```
select foo
from eg.Foo foo, eg.Bar bar
where foo.startDate = bar.date
```

restituirà tutte le istanze di `Foo` per le quali esiste una istanza di `bar` con una proprietà `date` uguale alla proprietà `startDate` del `Foo`. Le espressioni a percorso composto fanno sì che la clausola `where` sia estremamente potente. Considerate:

```
from eg.Cat cat where cat.mate.name is not null
```

Questa interrogazione si traduce in una query SQL con un join di tabella (interno) Se doveste scrivere una cosa come

```
from eg.Foo foo
where foo.bar.baz.customer.address.city is not null
```

otterreste una query che richiederebbe quattro join di tabella in SQL.

L'operatore = può essere usato per confrontare non solo proprietà, ma anche istanze:

```
from eg.Cat cat, eg.Cat rival where cat.mate = rival.mate

select cat, mate
from eg.Cat cat, eg.Cat mate
where cat.mate = mate
```

La proprietà speciale (in minuscolo) `id` può essere usata per fare riferimento all'identificatore univoco di un og-

getto. (potete anche usare il suo nome di proprietà)

```
from eg.Cat as cat where cat.id = 123

from eg.Cat as cat where cat.mate.id = 69
```

La seconda query è efficiente. Non è richiesto un join di tabella!

Possono anche essere usate le proprietà di identificatori compositi. Supponete che `Person` abbia un identificatore composto che consiste in `country` e `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
```

Ancora una volta, la seconda interrogazione non richiede join di tabella.

Nello stesso modo, la proprietà speciale `class` accede al valore del discriminatore di una istanza nel caso della persistenza polimorfica. Un nome di classe java annidato nella clausola where verrà tradotto nel suo valore di discriminazione.

```
from eg.Cat cat where cat.class = eg.DomesticCat
```

Potete anche specificare proprietà o componenti o tipi utente compositi (e di componenti di componenti, ecc.). Non tentate di utilizzare una espressione di percorso che finisca in una proprietà di tipo di componente (in opposizione ad una proprietà di un componente). Ad esempio, se `store.owner` è una entità con un componente indirizzo `(address)`

```
store.owner.address.city    // okay
store.owner.address         // error!
```

Un tipo "any" ha le proprietà speciali `id` e `class`, che consentono di esprimere un join nel modo seguente (in cui `AuditLog.item` è una proprietà mappata con `<any>`).

```
from eg.AuditLog log, eg.Payment payment
where log.item.class = 'eg.Payment' and log.item.id = payment.id
```

Notate che `log.item.class` e `payment.class` possono fare riferimento ai valori di colonne di database completamente diverse nella query precedente.

11.8. Espressioni

Le espressioni consentite nella clausola `where` includono la maggior parte delle cose che si scriverebbero in SQL:

- operatori matematici `+, -, *, /`
- operatori di confronto binario `=, >=, <=, <>, !=, like`
- operazioni logiche `and, or, not`
- concatenamento di stringhe ||
- funzioni scalari SQL come `upper()` e `lower()`
- le parentesi `( )` indicano i raggruppamenti
- `in`, `between`, `is null`

- parametri di ingresso JDBC `?`
- parametri con nome `:name`, `:start_date`, `:x1`
- letterali SQL `'foo'`, `69`, `'1970-01-01 10:00:01.0'`
- costanti Java `public static final` come `eg.Color.TABBY`

`in` e `between` possono essere usati così:

```
from eg.DomesticCat cat where cat.name between 'A' and 'B'

from eg.DomesticCat cat where cat.name in ( 'Foo', 'Bar', 'Baz' )
```

e le corrispondenti forme negative possono essere scritte

```
from eg.DomesticCat cat where cat.name not between 'A' and 'B'

from eg.DomesticCat cat where cat.name not in ( 'Foo', 'Bar', 'Baz' )
```

Nello stesso modo, `is null` e `is not null` possono essere usati per testare i valori null.

I booleani possono essere utilizzati facilmente nelle espressioni dichiarando delle sostituzioni HQL nella configurazione di hibernate:

```
<property name="hibernate.query.substitutions">true 1, false 0</property>
```

Questo sostituirà le parole chiave `true` e `false` con i letterali `1` and `0` nell'SQL tradotto da questo HQL:

```
from eg.Cat cat where cat.alive = true
```

Potete controllare la dimensione di una collezione con la proprietà speciale `size`, o la funzione speciale `size()`.

```
from eg.Cat cat where cat.kittens.size > 0

from eg.Cat cat where size(cat.kittens) > 0
```

Per le collezioni indicizzate, potete fare riferimento agli indici minimo e massimo usando `minIndex` e `maxIndex`. Nello stesso modo, potete fare riferimento agli elementi minimo e massimo di una collezione di un tipo di base usando `minElement` e `maxElement`.

```
from Calendar cal where cal.holidays.maxElement > current date
```

Ci sono anche le forme funzionali (le quali, a differenza dei costrutti qui sopra, non sono sensibili a maiuscole e minuscole):

```
from Order order where maxindex(order.items) > 100

from Order order where minelement(order.items) > 10000
```

Le funzioni SQL `any, some, all, exists, in` sono supportate quando viene loro passato l'insieme degli elementi o degli indici di una collezione (con le funzioni `elements` e `indices`) o il risultato di una sotto-interrogazione (vedete oltre).

```
select mother from eg.Cat as mother, eg.Cat as kit
where kit in elements(foo.kittens)

select p from eg.NameList list, eg.Person p
where p.name = some elements(list.names)

from eg.Cat cat where exists elements(cat.kittens)
```

```
from eg.Player p where 3 > all elements(p.scores)

from eg.Show show where 'fizard' in indices(show.acts)
```

Notate che questi costrutti - `size`, `elements`, `indices`, `minIndex`, `maxIndex`, `minElement`, `maxElement` - hanno alcune restrizioni di utilizzo:

- in una clausola `where`: solo per database con subselect
- in una clausola `select`: solo `elements` e `indices` hanno senso

Gli elementi delle collezioni indicizzate (array, liste, mappe) possono essere reperiti tramite il loro indice (solo in una clausola where):

```
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
```

Le espressioni all'interno di `[]` possono anche essere espressioni matematiche.

```
select item from Item item, Order order
where order.items[ size(order.items) - 1 ] = item
```

L'HQL fornisce anche la funzione predefinita `index()`, per gli elementi di una associazione uno-a-molti o una collezione di valori.

```
select item, index(item) from Order order
    join order.items item
where index(item) < 5
```

Possono essere usate le funzioni scalari SQL supportate dal database sottostante

```
from eg.DomesticCat cat where upper(cat.name) like 'FRI%'
```

Se non siete ancora convinti da tutto questo, pensate a quanto più lunga e meno leggibile sarebbe la query seguente se dovesse essere espressa in SQL:

```
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)
```

Suggerimento: qualcosa come

```
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
    )
```

11.9. La clausola order by

La lista restituita da una query può essere ordinata secondo una qualsiasi proprietà di una delle classi restituite o dei componenti:

```
from eg.DomesticCat cat
order by cat.name asc, cat.weight desc, cat.birthdate
```

Gli elementi opzionali `asc` o `desc` indicano rispettivamente ordine ascendente o discendente.

11.10. La clausola group by

Una interrogazione che renda valori aggregati può essere raggruppata in base a una proprietà qualunque di una delle classi rese o dei componenti:

```
select cat.color, sum(cat.weight), count(cat)
from eg.Cat cat
group by cat.color

select foo.id, avg( elements(foo.names) ), max( indices(foo.names) )
from eg.Foo foo
group by foo.id
```

Nota: potete usare i costrutti `elements` e `indices` in una clausola select, anche su database senza sub-select.

È consentita anche la clausola `having`.

```
select cat.color, sum(cat.weight), count(cat)
from eg.Cat cat
group by cat.color
having cat.color in (eg.Color.TABBY, eg.Color.BLACK)
```

Le funzioni SQL e le funzioni aggregate sono consentite nelle clausole `having` e `order by`, se supportate dal database sottostante (ad esempio non in MySQL).

```
select cat
from eg.Cat cat
    join cat.kittens kitten
group by cat
having avg(kitten.weight) > 100
order by count(kitten) asc, sum(kitten.weight) desc
```

Notate che né la clausola `group by` né la `order by` possono contenere espressioni aritmetiche.

11.11. Sottointerrogazioni

Per i database che supportano i sub-select, Hibernate supporta le sottointerrogazioni all'interno delle interrogazioni. Una sottointerrogazione deve essere circondata da parentesi (spesso da una chiamata di funzione aggregata SQL). Sono permesse anche le sottointerrogazioni correlate (ovvero quelle che fanno riferimento ad un sinonimo nella interrogazione esterna).

```
from eg.Cat as fatcat
where fatcat.weight > (
    select avg(cat.weight) from eg.DomesticCat cat
)

from eg.DomesticCat as cat
where cat.name = some (
    select name.nickName from eg.Name as name
)

from eg.Cat as cat
where not exists (
    from eg.Cat as mate where mate.mate = cat
)

from eg.DomesticCat as cat
where cat.name not in (
    select name.nickName from eg.Name as name
)
```

11.12. Esempi HQL

Le interrogazioni di Hibernate possono essere abbastanza potenti e complesse. In effetti, il potere del linguaggio di interrogazione è uno dei principali punti di forza di Hibernate. Qui presentiamo alcuni esempi di interrogazioni molto simili a query che sono state usate in un recente procetto. Notate che molte delle interrogazioni che scriverete sono molto più semplici di queste!

La prossima interrogazione restituisce l'id dell'ordine, il numero di oggetti e il valore totale dell'ordine per tutti gli ordini non pagati per un cliente particolare e un valore totale minimo, ordinando i risultati per valore totale. Nella determinazione dei prezzi, utilizza il catalogo corrente. La query SQL risultante, ha quattro join interni e una subselect non correlata che insistono sulle tabelle `ORDER`, `ORDER_LINE`, `PRODUCT`, `CATALOG` e `PRICE`.

```
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
```

Che mostro! A dire il vero, nella vita reale non ho molta passione per le sottointerrogazioni, quindi la mia era più come la seguente:

```
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 prossima interrogazione conta il numero di pagamenti in ogni stato, escludendo tutti i pagamenti nello stato `AWAITING_APPROVAL` quando il cambiamento di stato più recente era stato fatto dall'utente corrente. Si traduce in una query SLQ con due join interni e una subselect correlata sulle tabelle `PAYMENT`, `PAYMENT_STATUS` e `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
```

Se avessi mappato la collezione `statusChanges` come una lista invece di un set, l'interrogazione sarebbe stata molto più semplice da scrivere.

```
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 prossima interrogazione usa la funzione `isNull()` di MS SQL Server per restituire tutti i conti e i pagamenti non effettuati per l'organizzazione a cui l'utente corrente appartiene. Si traduce in una query SQL con tre join interni, un join esterno e una subselect sulle tabelle `ACCOUNT`, `PAYMENT`, `PAYMENT_STATUS`, `ACCOUNT_TYPE`, `ORGANIZATION` e `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
```

Per alcuni database, avremmo bisogno di fare a meno della subselect correlata.

```
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
```

11.13. Suggerimenti

Potete contare il numero dei risultati di una interrogazione senza restituirli veramente:

```
( (Integer) session.iterate("select count(*) from ....").next() ).intValue()
```

Per ordinare un risultato per dimensione di una collezione, usate l'interrogazione seguente:

```
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)
```

Se il vostro database supporta le sottointerrogazioni, potete mettere una condizione sulla dimensione della selezione nella clausola where della vostra query:

```
from User usr where size(usr.messages) >= 1
```

Mentre se il database non supporta i subselect potete usare la query seguente:

```
select usr.id, usr.name
from User usr.name
    join usr.messages msg
group by usr.id, usr.name
having count(msg) >= 1
```

Poiché questa soluzione non può restituire uno `User` con zero messaggi a causa del join interno, è anche utile la forma seguente:

```
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
```

Le proprietà di un javabean possono essere assegnate a parametri della query con nome:

```
Query q = s.createQuery("from foo in class Foo where foo.name=:name and foo.size=:size");
q.setProperties(fooBean); // fooBean has getName() and getSize()
List foos = q.list();
```

Le collezioni sono paginabili usando l'interfaccia `Query` con un filtro:

```
Query q = s.createFilter( collection, "" ); // the trivial filter
q.setMaxResults(PAGE_SIZE);
q.setFirstResult(PAGE_SIZE * pageNumber);
List page = q.list();
```

Gli elementi delle collezioni possono essere ordinati o raggruppati usando un filtro di interrogazione:

```
Collection orderedCollection = s.filter( collection, "order by this.amount" );
Collection counts = s.filter( collection, "select this.type, count(this) group by this.type" );
```

Potete individuare la dimensione di una collezione senza inizializzarla:

```
( (Integer) session.iterate("select count(*) from ....").next() ).intValue();
```

Capitolo 12. Interrogazioni per criteri

Hibernate ora offre una API di interrogazione per criteri intuitiva ed estensibile. Per ora quest'API è meno potente delle più mature funzionalità di interrogazione HQL. In particolare, le interrogazioni per criteri non supportano la proiezione o l'aggregazione.

12.1. Creazione di un'istanza di `Criteria`

L'interfaccia `net.sf.hibernate.Criteria` rappresenta un'interrogazione nei confronti di una particolare classe persistente. La `Session` è un produttore di istanze di `Criteria`.

```
Criteria crit = sess.createCriteria(Cat.class);
crit.setMaxResults(50);
List cats = crit.list();
```

12.2. Riduzione dell'insieme dei risultati

Un criterio individuale di interrogazione è un'istanza dell'interfaccia `net.sf.hibernate.expression.Criterion`. La classe `net.sf.hibernate.expression.Expression` definisce metodi "factory" (produttori) per otteere alcuni tipi predefiniti di `Criterion`.

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "Fritz%") )
    .add( Expression.between("weight", minWeight, maxWeight) )
    .list();
```

Le espressioni possono essere raggruppate logicamente.

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "Fritz%") )
    .add( Expression.or(
        Expression.eq( "age", new Integer(0) ),
        Expression.isNull("age")
    ) )
    .list();
```

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.in( "name", new String[] { "Fritz", "Izi", "Pk" } ) )
    .add( Expression.disjunction()
        .add( Expression.isNull("age") )
            .add( Expression.eq("age", new Integer(0) ) )
            .add( Expression.eq("age", new Integer(1) ) )
            .add( Expression.eq("age", new Integer(2) ) )
    ) )
    .list();
```

C'è un certo numero di tipi di criterio predefiniti (sottoclassi di `Expression`), ma uno di essi è particolarmente utile, perché consente di specificare direttamente dell'SQL.

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.sql("lower({alias}.name) like lower(?)", "Fritz%", Hibernate.STRING) )
    .list();
```

il segnaposto `{alias}` verrà sostituito dall'alias di riga della entità su cui si sta facendo l'interrogazione.

12.3. Ordinamento dei risultati

È possibile ordinare i risultati utilizzando `net.sf.hibernate.expression.Order`.

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "F%")
    .addOrder( Order.asc("name") )
    .addOrder( Order.desc("age") )
    .setMaxResults(50)
    .list();
```

12.4. Associazioni

Potete specificare semplicemente vincoli su entità correlate navigando le associazioni utilizzando `createCriteria()`.

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "F%")
    .createCriteria("kittens")
        .add( Expression.like("name", "F%")
    .list();
```

notate che il secondo `createCriteria()` restituisce una nuova istanza di `Criteria` che si riferisce agli elementi della collezione `kittens`.

La forma alternativa seguente è utile in alcune circostanze.

```
List cats = sess.createCriteria(Cat.class)
    .createAlias("kittens", "kt")
    .createAlias("mate", "mt")
    .add( Expression.eqProperty("kt.name", "mt.name") )
    .list();
```

(`createAlias()` non crea una nuova istanza di `Criteria`.)

Notate che le collezioni kittens che appartengono alle istanze di `Cat` restituita dalle due interrogazioni precedenti *non sono* pre-filtrate dal criterio! Se volete recuperare solo i gattini che corrispondano al criterio, dovete utilizzare `returnMaps()`.

```
List cats = sess.createCriteria(Cat.class)
    .createCriteria("kittens", "kt")
        .add( Expression.eq("name", "F%") )
    .returnMaps()
    .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");
}
```

12.5. Caricamento dinamico delle associazioni

Usando il metodo `setFetchMode()` è possibile specificare la semantica di caricamento delle associazioni in fase di esecuzione.

```
List cats = sess.createCriteria(Cat.class)
    .add( Expression.like("name", "Fritz%") )
    .setFetchMode("mate", FetchMode.EAGER)
    .setFetchMode("kittens", FetchMode.EAGER)
    .list();
```

Questa interrogazione caricher sia `mate` sia `kittens` tramite join esterni.

12.6. Interrogazioni per esempi

La classe `net.sf.hibernate.expression.Example` permette di costruire un criterio di interrogazione a partire da una data istanza.

```
Cat cat = new Cat();
cat.setSex('F');
cat.setColor(Color.BLACK);
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .list();
```

Le proprietà di versione, gli identificatori e le associazioni vengono ignorati. Il comportamento predefinito è di escludere le proprietà di valore null.

Potete impostare come l'`Example` venga applicato.

```
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();
```

Potete anche usare gli esempi per impostare criteri sugli oggetti associati.

```
List results = session.createCriteria(Cat.class)
    .add( Example.create(cat) )
    .createCriteria("mate")
        .add( Example.create( cat.getMate() ) )
    .list();
```

Capitolo 13. Interrogazioni SQL native

Potete anche esprimere interrogazioni nel dialetto SQL nativo del vostro database. È una cosa particolarmente utile se volete utilizzare funzionalità specifiche del database come la parola chiave CONNECT in Oracle. Ciò consente anche di seguire un percorso di migrazione più pulito da una applicazione direttamente basata su SQL/JDBC ad una che si appoggi ad Hibernate.

13.1. Creazione di una `Query` basaa su SQL

Le interrogazioni SQL sono esposte tramite l'interfaccia `Query`, proprio come le normali interrogazioni HQL. La sola differenza è nell'uso del metodo `Session.createSQLQuery()`.

```
Query sqlQuery = sess.createSQLQuery("select {cat.*} from cats {cat}", "cat", Cat.class);
sqlQuery.setMaxResults(50);
List cats = sqlQuery.list();
```

I tre parametri forniti al metodo `createSQLQuery()` sono:

- la stringa con l'interrogazione SQL
- il nome di un alias di tabella
- la classe persistente restituita dall'interrogazione

Il nome dell'alias viene usato nella stringa sql per riferirsi alle proprietà della classe mappata (in questo caso `Cat`). Potete recuperare oggetti multipli per riga fornendo un array di `String` con i nomi degli alias e un array di `Class` per le classi corrispondenti.

13.2. Alias e riferimenti alle proprietà

La notazione `{cat.*}` usata sopra è un'abbreviazione per "tutte le proprietà". Potete anche elencare esplicitamente le proprietà, ma dovete lasciare che Hibernate fornisca alias di colonna per ogni proprietà. Le etichette per gli alias di queste colonne sono il nome della proprietà preceduto dall'alias di tabella. Nell'esempio seguente, recuperiamo oggetti `Cat` da una tabella diversa (`cat_log`) rispetto a quella dichiarata nei metadati di mappaggio. Notate che possiamo anche usare gli alias di proprietà nelle clausole "where".

```
String sql = "select cat.originalId as {cat.id}, "
    + "  cat.mateid as {cat.mate}, cat.sex as {cat.sex}, "
    + "  cat.weight*10 as {cat.weight}, cat.name as {cat.name}"
    + "     from cat_log cat where {cat.mate} = :catId"
List loggedCats = sess.createSQLQuery(sql, "cat", Cat.class)
    .setLong("catId", catId)
    .list();
```

Nota: se elencate esplicitamente ogni proprietà, dovete includere tutte le proprietà della classe *e delle sue sottoclassi*!

13.3. Interrogazioni SQL con nome

Le interrogazioni SQL con nome possono venire definite nel documento di mappaggio e chiamate esattamente

nello stesso modo in cui viene chiamata una interrogazione HQL con nome.

```
List people = sess.getNamedQuery("mySqlQuery")
    .setMaxResults(50)
    .list();
```

```
<sql-query name="mySqlQuery">
    <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 'Hiber%'
</sql-query>
```

Capitolo 14. Ottimizzare le prestazioni di Hibernate

14.1. Capire gli aspetti legati alle prestazioni delle collezioni

Abbiamo già parlato delle collezioni da un punto di vista funzionale. In questa sezione mettiamo in evidenza alcune questioni legate a come le collezioni si comportano durante l'esecuzione

14.1.1. Tassonomia

Hibernate definisce tre tipi fondamentali di collezioni:

- collezioni di valori
- associazioni uno-a-molti
- associazioni molti-a-molti

Questa classificazione distingue le varie relazioni tra tabelle e chiavi esterne, ma non ci dice abbastanza di quello che ci interessa sul modello relazionale. Per capire completamente la struttura relazionale e le caratteristiche di performance, dobbiamo anche prendere in cosiderazione la struttura della chiave primaria che viene usata da Hibernate per modificare o cancellare le righe corrispondenti alla collezione. Questo suggerisce la classificazione seguente:

- collezioni con indice (indexed collection)
- insiemi (set)
- "sacchi" (bags)

Tutte le collezioni indicizzate (mappe, liste, array) hanno una chiave primaria che consiste nelle colonne `<key>` (chiave) e `<index>` (indice). Solitamente in questi casi gli aggiornamenti delle collezioni sono molto performanti, poiché la chiave primaria può essere indicizzata in modo efficiente e una riga particolare può quindi essere localizzata rapidamente quando Hibernate cerca di modificarla o cancellarla.

Gli insiemi hanno una chiave primaria che consiste delle colonne `<key>` ed `<element>`. Questo può essere meno efficiente per alcuni tipi di elemento della collezione, in particolare per elementi composti o campi molto lunghi di testo o dati binari; il database può non essere in grado di indicizzare una chiave primaria complessa in maniera altrettanto efficiente che nel caso precedente. Da un altro punto di vista, per associazioni uno-a-molti o molti-a-molti, in particolare nel caso di identificatori sintetici, è probabile che sia efficiente nello stesso modo. (annotazione: se volete che `SchemaExport` crei davvero la chiave primaria di un `<set>` per voi, dovete dichiarare tutte le colonne come `not-null="true"`.)

I "sacchi" (bags) sono il caso peggiore. Poiché un bag consente elementi duplicati e non ha una colonna indice, non può essere definita una chiave primaria. Hibernate non ha modo di distinguere tra righe duplicate, e quindi risolve il problema rimuovendo completamente (con una singola `DELETE`) e ricreando la collezione ogni volta che cambia. Questo tuttavia può essere molto inefficiente.

Notate che per una collezione uno-a-molti, la "chiave primaria" può non essere la chiave primaria fisica della tabella del database - ma anche in questo caso la classificazione qui sopra è comunque utile, poiché riflette come Hibernate recupera righe specifiche della collezione.

14.1.2. Liste, mappe e insiemi sono le collezioni più efficienti da modificare

Dalla discussione di cui sopra, dovrebbe essere chiaro che le collezioni indicizzate e (di solito) gli insiemi consentono le operazioni più efficienti in termini di aggiunta, rimozione e modifica di elementi.

C'è un vantaggio ulteriore che le collezioni indicizzate hanno rispetto agli insiemi per le associazioni molti-a-molti o le collezioni di valori. Per come è fatta la struttura di un `Set`, Hibernate non aggiorna neppure (`UPDATE`) una riga, quando un elemento è "cambiato". I cambiamenti ad un `Set` funzionano semper via `INSERT` e `DELETE` (di righe individuali). Ancora una volta, ripetiamo che questa considerazione non si applica alle associazioni uno-a-molti.

Poiché ricordiamo che gli array non possono essere caricati a richiesta (lazy), concludiamo quindi che le liste, le mappe e gli insiemi sono i tipi di collezione più performanti. (Con l'avvertimento, ancora una volta, che un set può essere meno efficiente per alcune collezioni di valori)

Gli insiemi sono probabilmente il genere di collezione più comune nelle applicazioni basate su Hibernate.

C'è una funzionalità non documentata in questa versione di Hibernate. Il mappaggio `<idbag>` implementa una semantica a "bag" per una collezione di valori o una associazione molti-a-molti ed è più efficiente di qualsiasi altro stile di collezione, in questo caso!

14.1.3. I bag e le liste sono le collezioni inverse più efficienti

Prima che buttiate via i "bag" per sempre, c'è un caso particolare in cui essi (e le liste) sono molto più performanti degli insiemi. Per una collezione con `inverse="true"` (l'idioma standard per una relazione uno-a-molti, ad esempio) possiamo aggiungere elementi ad un bag o una lista senza bisogno di inizializzare (fetch) gli elementi del bag stesso! Questo perché `Collection.add()` o `Collection.addAll()` devono sempre ritornare "true" per un bag o una `List` (a differenza di un `Set`). Questo può rendere il codice seguente molto più veloce:

```
Parent p = (Parent) sess.load(Parent.class, id);
    Child c = new Child();
    c.setParent(p);
    p.getChildren().add(c);  //no need to fetch the collection!
    sess.flush();
```

14.1.4. Cancellazione in un colpo solo

Di tanto in tanto, cancellare elementi di una collezione ad uno ad uno può essere estremamente inefficiente. Hibernate non è completamente stupido, per cui sa che non deve farlo nel caso in cui una collezione sia stata appena svuotata (tramite `list.clear()`, ad esempio). In questo caso, Hibernate utilizzerà una singola `DELETE` ed è tutto!

Supponiamo di aggiungere un elemento singolo ad una collezione di dimensione venti, e poi rimuovere due elementi. Hibernate lancerà una `INSERT` e due `DELETE` (a meno che la collezione sia un bag). Questo è certamente auspicabile.

Però, supponiamo di rimuovere diciotto elementi, lasciandone due, e poi di aggiungere tre elementi nuovi. Ci sono due modi possibili di procedere.

- cancellare le diciotto righe una ad una e poi inserire le tre
- rimuovere tutta la collezione in un solo comando `DELETE` e inserire tutti i cinque elementi rimanenti uno ad uno

Hibernate non è abbastanza furbo da sapere che la seconda opzione è probabilmente più veloce, in questo caso. (e probabilmente non sarebbe auspicabile che Hibernate lo fosse, perché un comportamento del genere può confondere dei trigger, ecc.)

Fortunatamente, potete imporre questo comportamento (cioè la seconda strategia) in ogni momento scartando (cioè dereferenziando) la collezione originale ed impostando una nuova collezione con tutti gli elementi che devono rimanere. Questo può essere molto utile e potente, in certi casi.

Abbiamo già mostrato come si può usare l'inizializzazione a richiesta (lazy) per le collezioni persistenti nel capitolo sui mappaggi delle collezioni. Un effetto simile si può ottenere per i riferimenti agli oggetti comuni, usando i mediatori (proxy) CGLIB. Abbiamo anche detto che Hibernate fa il caching degli oggetti persistenti al livello della `Session`. È comunque possibile impostare strategie di caching più aggressive per classi specifiche.

Nella prossima sezione, vi mostriamo come usare queste funzionalità, e quindi raggiungere prestazioni più elevate quando necessario.

14.2. Mediatori (proxy) per l'inizializzazione a richiesta (lazy)

Hibernate implementa un sistema per l'inizializzazione ritardata (lazy) degli oggetti persistenti tramite dei mediatori (proxy) creati in fase di esecuzione tramite una tecnica di arricchimento del codice binario (byte-code) che sfrutta le funzionalità fornite dall'eccellente libreria CGLIB.

Il file di mappaggio dichiara una classe o un'interfaccia che va usata come interfaccia del proxy per quella classe. L'approccio raccomandato è specificare la classe stessa:

```
<class name="eg.Order" proxy="eg.Order">
```

Il tipo dei proxy in fase di esecuzione sarà una sottoclasse di `Order`. Notate che la classe "mediata" (proxied) deve implementare un costruttore di default per lo meno con visibilità a livello di package.

Ci sono alcune peculiarità di cui essere a conoscenza, quando si estende questo approccio alle classi polimorfiche, ad esempio:

```
<class name="eg.Cat" proxy="eg.Cat">
    ......
    <subclass name="eg.DomesticCat" proxy="eg.DomesticCat">
        .....
    </subclass>
</class>
```

Prima di tutto, le istanze di `Cat` non potranno essere oggetto di "cast" a `DomesticCat`, anche se l'istanza sottostante è effettivamente un `DomesticCat`.

```
Cat cat = (Cat) session.load(Cat.class, id);  // instantiate a proxy (does not hit the db)
if ( cat.isDomesticCat() ) {                  // hit the db to initialize the proxy
    DomesticCat dc = (DomesticCat) cat;       // Error!
    ....
}
```

In secondo luogo, è possibile che la semantica di == non valga per il proxy.

```
Cat cat = (Cat) session.load(Cat.class, id);            // instantiate a Cat proxy
DomesticCat dc =
    (DomesticCat) session.load(DomesticCat.class, id);  // required new DomesticCat proxy!
System.out.println(cat==dc);                            // false
```

Comunque, queste situazioni non sono poi così male come sembra. Anche se ora abbiamo due riferimenti diversi ad oggetti proxy, l'istanza sottostante è comunque la stessa:

```
cat.setWeight(11.0);  // hit the db to initialize the proxy
System.out.println( dc.getWeight() );  // 11.0
```

Terzo, non è possibile usare un mediatore CGLIB per una classe `final` o per una classe con metodi `final`.

Infine, se il vostro oggetto persistente acquisisce delle risorse in fase di istanziazione (ad esempio negli inizializzatori o nel costruttore di default), quelle risorse saranno acquisite anche dal proxy, poiché la classe del proxy è effettivamente una sottoclasse della classe persistente.

Questi problemi sono tutti derivanti da limitazioni di base nel modello a ereditarietà singola di Java. Se volete evitarli, le vostre classi persistenti devono implementare un'interfaccia che dichiari i loro metodi di business. Dovete poi specificare queste interfacce nel file di mapping, ad esempio così:

```
<class name="eg.Cat" proxy="eg.ICat">
    ......
    <subclass name="eg.DomesticCat" proxy="eg.IDomesticCat">
        .....
    </subclass>
</class>
```

laddove `Cat` implementa l'interfaccia `ICat` e `DomesticCat` implementa l'interfaccia `IDomesticCat`. A questo punto, `load()` o `iterate()` possono restituire direttamente istanze di `Cat` e `DomesticCat` . (Notate che `find()` non restituisce mediatori.)

```
ICat cat = (ICat) session.load(Cat.class, catid);
Iterator iter = session.iterate("from cat in class eg.Cat where cat.name='fritz'");
ICat fritz = (ICat) iter.next();
```

Anche le relazioni sono inizializzate in maniera ritardata. Questo significa che dovete dichiarare le proprietà di tipo `ICat`, e non `Cat`.

Alcune operazioni *non* richiedono inizializzazione del proxy

- `equals()`, se la classe persistente non sovrascrive `equals()`
- `hashCode()`, se la classe persistente non sovrascrive `hashCode()`
- Il metodo "getter" per l'identificatore.

Hibernate individuerà le classi persistenti che sovrascrivono `equals()` o `hashCode()`.

Le eccezioni che capitano quando si inizializza un proxy vengono racchiuse in una `LazyInitializationException`.

In alcuni casi, dobbiamo assicuarci che un mediatore o una collezione vengano inizializzati prima di chiudere la `Session`. Naturalmente, possiamo sempre forzare l'inizializzazione chiamando `cat.getSex()` o `cat.getKittens().size()`, ad esempio. Ma questo confonde chi legge il codice e non è pratico per del codice generico. I metodi statici `Hibernate.initialize()` e `Hibernate.isInitialized()` forniscono all'applicazione un modo comodo per lavorare con collezioni inizializzate a richiesta o con i mediatori. `Hibernate.initialize(cat)` imporrà l'inizializzazione di un mediatore `cat`, a condizione che la sua `Session` sia ancora aperta. `Hibernate.initialize( cat.getKittens() )` ha un effetto simile per la collezione dei gattini (kitten ;)).

14.3. La cache di secondo livello

Una `Session` di Hibernate è una cache di dati persistenti durante la transazione. È possibile configurare una cache a livello di cluster o a livello di macchina virtuale (JVM-level o `SessionFactory`-level) per classi o collezioni specifiche. È anche possibile agganciare (plug-in) una cache in cluster. Fate attenzione, tuttavia: le cache non sono mai coscienti di cambiamenti fatti ai dati sul contentitore fisico da un'altra applicazione (benché possano essere configurate in modo tale da fare scadere i dati conservati in memoria).

L'impostazione predefinita di Hibernate è di usare la libreria EHCache per il caching a livello di JVM (Il supporto di JCS è deprecato e verrà rimosso in una versione futura di Hibernate). È possibile scegliere una implementazione diversa speficicando il nome di una classe che implementi `net.sf.hibernate.cache.CacheProvider` usando la proprietà `hibernate.cache.provider_class`.

Tabella 14.1. Fornitori di cache

Cache	Classe fornitore	Tipo	Funziona in cluster	Supporta interrogazione della cache
Hashtable (non adatta per un uso in produzione)	`net.sf.hibernate.cache.HashtableCacheProvider`	memoria		sì
EHCache	`net.sf.ehcache.hibernate.Provider`	memoria, disco		sì
OSCache	`net.sf.hibernate.cache.OSCacheProvider`	memoria, disco		sì
SwarmCache	`net.sf.hibernate.cache.SwarmCacheProvider`	cluster (via ip multicast)	sì (invalidazione sul cluster)	
JBoss TreeCache	`net.sf.hibernate.cache.TreeCacheProvider`	cluster (via ip multicast), transazionale	sì (replicazione)	

14.3.1. Mappaggi e cache

L'elemento `<cache>` per il mappaggio di una classe o di una collezione ha la forma seguente:

```
<cache
    usage="transactional|read-write|nonstrict-read-write|read-only"  (1)
/>
```

(1) `usage` specifica la strategia di caching: `transactional`, `read-write`, `nonstrict-read-write` or `read-only`

In alternativa (preferibilmente), si possono specificare gli elementi `<class-cache>` e `<collection-cache>` in `hibernate.cfg.xml`.

L'attributo `usage` speficica una *strategia di concorrenza per la cache*.

14.3.2. Strategia: sola lettura

Se la vostra applicazione ha bisogno di leggere ma non modifica mai istanze di una classe persistente, si può usare una cache `read-only` (sola lettura). Si tratta della strategia più semplice e più performante. Funziona anche perfettamente in un cluster.

```
<class name="eg.Immutable" mutable="false">
    <cache usage="read-only"/>
    ....
</class>
```

14.3.3. Strategia: lettura/scrittura

Se l'applicazione deve modificare i dati, una cache `read-write` (lettura/scrittura) potrebbe essere appropriata. Questa strategia di caching non dovrebbe essere mai usata se è richiesto un livello di isolamento serializzabile delle transazioni. Se la cache è usata in un ambiente JTA, dovete specificare la proprietà `hibernate.transaction.manager_lookup_class`, indicando una strategia per ottenere il `TransactionManager` JTA. In altri ambienti, dovete assicurarvi che la transazione venga completata quando vengono chiamati `Session.close()` o `Session.disconnect()`. Se volete usare questa strategia in un cluster, dovete assicurarvi che l'implementazione della cache sottostante supporti il locking. La cache fornita con Hibernate *non* lo fa.

```
<class name="eg.Cat" .... >
    <cache usage="read-write"/>
    ....
    <set name="kittens" ... >
        <cache usage="read-write"/>
        ....
    </set>
</class>
```

14.3.4. Strategia: lettura/scrittura non stretta

Se l'applicazione ha bisogno di modificare dati solo occasionalmente (cioè se è molto improbabile che due transazioni tentino di modificare lo stesso oggetto simultaneamente) e l'isolamento stretto delle transazioni non è richiesto, potrebbe essere appropriata una cache `nonstrict-read-write` (lettura/scrittura non stretta). Se la cache è usata in un ambiente JTA, dovete specificare `hibernate.transaction.manager_lookup_class`. In altri ambienti, dovete assicurare che la transazione sia completa quando vengono chiamati `Session.close()` o `Session.disconnect()`.

14.3.5. Strategia: transazionale

La strategia di caching `transazionale` fornisce supporto per cache completamente transazionali come la JBoss TreeCache. Una cache di questo tipo può essere usata solo in un contesto JTA e dovete specificare la proprietà `hibernate.transaction.manager_lookup_class`.

Nessuno dei fornitori di cache supporta tutte le strategie di concorrenza. La tabella seguente mostra quali fornitori sono compatibili con quali strategie di concorrenza.

Tabella 14.2. Supporto alle strategie di concorrenza delle cache

Cache	read-only	nonstrict-read-write	read-write	transactional
Hashtable (non adatta per un uso in	Sì	Sì	Sì	

Cache	read-only	nonstrict-read-write	read-write	transactional
produzione)				
EHCache	Sì	Sì	Sì	
OSCache	Sì	Sì	Sì	
SwarmCache	Sì	Sì		
JBoss TreeCache	Sì			Sì

14.4. Gestione della cache di `Session`

Ogni volta che passate un oggetto ai metodi`save()`, `update()` o `saveOrUpdate()` e ogni volta che ne recuperate uno usando `load()`, `find()`, `iterate()`, o `filter()`, quell'oggetto viene aggiunto alla cache interna della `Session`. Quando poi viene chiamato `flush()`, lo stato di quell'oggetto sarà sincronizzato con il database. Se non volete che questa sincronizzazione avvenga, o se state elaborando un grande numero di oggetti e volete gestire la memoria efficentemente, potete usare il metodo `evict()` per rimuovere l'oggetto e le sue collezioni dalla cache.

```
Iterator cats = sess.iterate("from eg.Cat as cat"); //a huge result set
while ( cats.hasNext() ) {
    Cat cat = (Cat) iter.next();
    doSomethingWithACat(cat);
    sess.evict(cat);
}
```

La `Session` fornisce anche un metodo `contains()` per determinare se un'istanza appartiene alla cache di sessione.

Per rimuovere completamente tutti gli oggetti dalla cache si sessione, esiste il metodo `Session.clear()`

Per la cache di secfondo livello, ci sono dei metodi definiti su `SessionFactory` e che hanno lo scopo di rimuovere lo stato di un'istanza dalla cache, una intera classe, una istanza di collezione o un intero ruolo di collezione.

14.5. La cache delle query

Gli insiemi di risultati (result set) delle query possono anche venire messi in cache. Questo è utile solo per quelle query che vengono lanciate frequentemente con gli stessi parametri. Per usare la cache delle query dovete prima attivarla settando la proprietà `hibernate.cache.use_query_cache=true`. Questo causa la creazione di due regioni nella cache, una che mantiene i set di risultati delle query, (`net.sf.hibernate.cache.QueryCache`), l'altra che mantiene le etichette di tempo (timestamp) degli aggiornamenti più recenti alle tabelle interrogate. (`net.sf.hibernate.cache.UpdateTimestampsCache`). Notate che la cache delle query non memorizza lo stato delle entità nel result set; quello che mette in cache sono solo i valori dei risultati e i valori dei tipi. Per questo, la cache delle query viene solitamente usata insieme alla cache di secondo livello.

La maggior parte delle interrogazione non traggono particolari benefici dal caching, per questo l'impostazione predefinita non lo prevede. Per attivarlo, chiamate `Query.setCacheable(true)`. Questo metodo consente alla query di cercare risultati nella cache o di aggiungere i suoi risultati quando viene eseguita.

Se avete bisogno di controllo più raffinato sulle politiche di scadenza delle cache, potete specificare una regione della cache per nome e per una particolare interrogazione chiamando il metodo `Query.setCacheRegion()`.

```
List blogs = sess.createQuery("from Blog blog where blog.blogger = :blogger")
    .setEntity("blogger", blogger)
    .setMaxResults(15)
    .setCacheable(true)
    .setCacheRegion("frontpages")
    .list();
```

Capitolo 15. Guida degli strumenti

La cosiddetta "ingegnerizzazione circolare" ("roundtrip engineering") con Hibernate è possibile o utilizzando un insieme di strumenti a linea di comando manutenuti come parte del progetto Hibernate stesso, o sfruttando il supporto ad Hibernate fornito da progetti come XDoclet, Middlegen e AndroMDA.

La distribuzione principale di Hibernate include lo strumento più importante (che può essere usato anche direttamente dall'interno di Hibernate):

- Generazione di uno schema DDL da un file di mappaggio (cioè `SchemaExport`, `hbm2ddl`)

Altri strumenti forniti direttamente dal progetto Hibernate vengono rilasciati in un pacchetto separato, detto delle *Hibernate Extensions*. Il pacchetto include strumenti per i compiti seguenti:

- Generazione di sorgenti Java da un file di mappaggio (cioè `CodeGenerator`, `hbm2java`)
- generazione di file di mappaggio da classi Java compilate o da sorgenti Java con indicazioni di contrassegno ("markup") di XDoclet markup (ovvero `MapGenerator`, `class2hbm`)

In realtà c'è un altro programma di utilità che sopravvive tra le estensioni di Hibernate: `ddl2hbm`. Viene considerato deprecato e non viene più manutenuto: Middlegen svolge lo stesso compito facendo un lavoro migliore.

Strumenti di terze parti con supporto per Hibernate sono:

- Middlegen (generazione di file di mappaggio da uno schema di database esistente)
- AndroMDA (MDA (Model-Driven Architecture o architettura guidata dal modello) è un approccio alla generazione di codice per classi persistenti a partire da diagrammi UML e dalla loro rappresentazione XML/XMI

Questi strumenti di terze parti non sono documentati in questo manuale. Fate riferimento al sito di hibernate per informazioni aggiornate al riguardo (una istantanea del sito è disponibile nel pacchetto di distribuzione).

15.1. Generazione dello schema

Il DDL può venire generato dai file di mappaggio tramite una utilità a riga di comando. Un file di comandi ("batch") si trova nella cartella `hibernate-x.x.x/bin` dell'archivio principale di Hibernate.

Lo schema generato include vincoli di integrità referenziale (chiavi primarie ed esterne) per le entità e le tabelle di collezione. Vengono anche create le tabelle e le sequenze per i generatori di identificatori mappati nei file hbm.

Dovete specificare un `dialetto` SQL tramite la proprietà `hibernate.dialect` quando si usa questo strumento.

15.1.1. Personalizzazione dello schema

Molti elementi di mappaggio in Hibernate definiscono un attributo opzionale che si chiama `length`. Con esso potete impostare la lunghezza di una colonna.

Alcuni tag accettano anche un attributo `not-null` (che genera un vincolo `NOT NULL` sulle colonne della tabella)

e un attributo `unique` (per generare vincoli `UNIQUE`).

Alcuni tag accettano un attributo `index` per specificare il nome di un indice per la colonna. Un attributo `unique-key` può essere usato per raggruppare colonne in un vincolo di chiave a singola unità. Attualmente il valore specificato dell'attributo `unique-key` *non* viene usato per denominare il vincolo, ma solo per raggruppare le colonne nel file di mappaggio.

Esempi:

```
<property name="foo" type="string" length="64" not-null="true"/>

<many-to-one name="bar" foreign-key="fk_foo_bar" not-null="true"/>

<element column="serial_number" type="long" not-null="true" unique="true"/>
```

In alternativa, questi elementi accettano anche un elemento figlio `<column>`, che è particolarmente utile per i tipi multi-colonna:

```
<property name="foo" type="string">
    <column name="foo" length="64" not-null="true" sql-type="text"/>
</property>

<property name="bar" type="my.customtypes.MultiColumnType"/>
    <column name="fee" not-null="true" index="bar_idx"/>
    <column name="fi" not-null="true" index="bar_idx"/>
    <column name="fo" not-null="true" index="bar_idx"/>
</property>
```

L'attributo `sql-type` consente all'utente di sovrascrivere il mappaggio predefinito dal tipo di Hibernate al tipo di dati SQL.

L'attributo `check` vi consente di specificare un vincolo di controllo.

```
<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>
```

Tabella 15.1. Summary

Attributi	Valori
`length`	`true|false`
`not-null`	`true|false`
`unique`	`true|false`
`index`	`nome_indice`
`unique-key`	`nome_chiave_univoca`
`foreign-key`	`nome_chiave_esterna`
`sql-type`	`tipo_colonna`
`check`	espressione SQL

15.1.2. Esecuzione del programma

Lo strumento `SchemaExport` scrive uno script DDL sull'uscita standard (stdout) e/o esegue le istruzioni DDL.

`java -cp` *classpath_di_hibernate* `net.sf.hibernate.tool.hbm2ddl.SchemaExport` *opzioni file_di_mappaggio*

Tabella 15.2. Opzioni della linea di comando di `SchemaExport`

Opzione	Descrizione
`--quiet`	non scrive lo script sull'uscita standard (stdout)
`--drop`	elimina solo le tabelle
`--text`	non esporta sul database
`--output=my_schema.ddl`	emette lo script ddl su un file
`--config=hibernate.cfg.xml`	legge la configurazione di Hibernate da un file XML particolare
`--properties=hibernate.properties`	legge le proprietà del database da un file
`--format`	nello script l'SQL generato viene formattato in una maniera "carina"
`--delimiter=x`	imposta un delimitatore di fine linea per lo script

Potete anche annidare `SchemaExport` nella vostra applicazione:

```
Configuration cfg = ....;
new SchemaExport(cfg).create(false, true);
```

15.1.3. Proprietà

Le proprietà del database possono essere specificate

- come proprietà di sistema con `-D`<*property*>
- in un file `hibernate.properties`
- in un file di proprietà con un nome diverso con `--properties`

Le proprietà richieste sono:

Tabella 15.3. Proprietà di connessione di SchemaExport

Nome proprietà	Descrizione
`hibernate.connection.driver_class`	classe del driver jdbc
`hibernate.connection.url`	url jdbc
`hibernate.connection.username`	nome utente database
`hibernate.connection.password`	parola chiave database
`hibernate.dialect`	dialetto

15.1.4. Utilizzo di Ant

È possibile chiamare lo `SchemaExport` dal vostro script di Ant:

```
<target name="schemaexport">
    <taskdef name="schemaexport"
        classname="net.sf.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>
```

15.1.5. Aggiornamenti incrementali dello schema

Lo strumento `SchemaUpdate` è in grado di aggiornare uno schema esistente con cambiamenti "incrementali". Notate che `SchemaUpdate` dipende in maniera massiccia dall'API dei metadati JDBC, e per questo non funziona con tutti i driver JDBC.

`java -cp` *classpath_di_hibernate* `net.sf.hibernate.tool.hbm2ddl.SchemaUpdate` *opzioni file_di_mappaggio*

Tabella 15.4. Opzioni da linea di comando per `SchemaUpdate`

Opzione	Descrizione
`--quiet`	non scrive lo script su stdout
`--properties=hibernate.properties`	legge le proprietà del database da un file

Potete annidare `SchemaUpdate` nella vostra applicazione:

```
Configuration cfg = ....;
new SchemaUpdate(cfg).execute(false);
```

15.1.6. Utilizzo di Ant per gli aggiornamenti incrementali dello schema

Potete chiamare `SchemaUpdate` da uno script di Ant:

```
<target name="schemaupdate">
    <taskdef name="schemaupdate"
        classname="net.sf.hibernate.tool.hbm2ddl.SchemaUpdateTask"
        classpathref="class.path"/>

    <schemaupdate
        properties="hibernate.properties"
        quiet="no">
        <fileset dir="src">
            <include name="**/*.hbm.xml"/>
        </fileset>
```

```
    </schemaupdate>
</target>
```

15.2. Generazione di codice

Il generatore di codice di Hibernate può essere usato per generare l'implementazione della struttura delle classi java da un file di mappaggio di Hibernate. Lo strumento è incluso nel pacchetto delle estensioni di Hibernate (Hibernate Extensions), scaricabile separatamente dal pacchetto principale.

`hbm2java` interpreta i file di mappaggio e a partire da questi genera classi java complete. In questo modo, usando `hbm2java` è possibile "solo" fornire i file `.hbm` e non preoccuparsi della produzione manuale delle classi Java.

`java -cp` *classpath_di_hibernate* `net.sf.hibernate.tool.hbm2java.CodeGenerator` *opzioni file_di_mappaggio*

Tabella 15.5. Opzioni da linea di comando del generatore di codice

Opzione	Descrizione
`--output=`*cartella_di_output*	cartella radice per il codice generato
`--config=`*file_di_configurazione*	file opzionale per configurare hbm2java

15.2.1. Il file di configurazione (opzionale)

Il file di configurazione fornisce una maniera per specificare dei "produttori" multipli per il codice sorgente e per dichiarare attributi `<meta>` che sono "globali" per visibilità. Leggete di più al riguardo nella sezione sull'attributo `<meta>`.

```
<codegen>
    <meta attribute="implements">codegen.test.IAuditable</meta>
    <generate renderer="net.sf.hibernate.tool.hbm2java.BasicRenderer"/>
    <generate
        package="autofinders.only"
        suffix="Finder"
        renderer="net.sf.hibernate.tool.hbm2java.FinderRenderer"/>
</codegen>
```

Questo file di configurazione dichiara un attributo meta globale "implements" e specifica due produttori (renderers), quello predefinito (BasicRenderer) e un produttore che genera dei "Finder" (vedete anche in "generazione basica dei finder" più sotto).

Il secondo produttore viene fornito con attributi "package" e "suffix".

L'attributo "package" specifica che i file di codice sorgente generati da questo renderer dovrebbero essere posti in questo package invece che in quello specificato nei file `.hbm`.

L'attributo "suffix" specifica il suffisso per i file generati. Nel caso dell'esempio, un file chiamato `Foo.java` diventerebbe invece `FooFinder.java`.

15.2.2. L'attributo `meta`

L'etichetta `<meta>` è una maniera semplice di annotare il file `hbm.xml`, e dare agli strumenti un posto naturale per memorizzare o leggere informazioni che non siano direttamente correlate con il nucleo di Hibernate.

Potete usare l'etichetta `<meta>` per indicare ad `hbm2java` di generare solo metodi "setter" protetti, fare in modo tale che le classi implementino sempre un certo insieme di interfacce, fare in modo tale che estendano una certa classe di base, o altro.

L'esempio seguente:

```
<class name="Person">
    <meta attribute="class-description">
        Javadoc per la classe Person
        @author Frodo
    </meta>
    <meta attribute="implements">IAuditable</meta>
    <id name="id" type="long">
        <meta attribute="scope-set">protected</meta>
        <generator class="increment"/>
    </id>
    <property name="name" type="string">
        <meta attribute="field-description">Il nome della persona</meta>
    </property>
</class>
```

produrrà qualcosa come ciò che segue (il codice è stato accorciato per renderlo più comprensibile). Notate il commento Javadoc e il metodo set protetto:

```
// package di default

import java.io.Serializable;
import org.apache.commons.lang.builder.EqualsBuilder;
import org.apache.commons.lang.builder.HashCodeBuilder;
import org.apache.commons.lang.builder.ToStringBuilder;

/**
 *         Javadoc per la classe Person
 *         @author Frodo
 *
 */
public class Person implements Serializable, IAuditable {

    /** identifier field */
    public Long id;

    /** nullable persistent field */
    public String name;

    /** full constructor */
    public Person(java.lang.String name) {
        this.name = name;
    }

    /** default constructor */
    public Person() {
    }

    public java.lang.Long getId() {
        return this.id;
    }

    protected void setId(java.lang.Long id) {
        this.id = id;
    }

    /**
     * Il nome della persona
     */
```

```
    public java.lang.String getName() {
        return this.name;
    }

    public void setName(java.lang.String name) {
        this.name = name;
    }

}
```

Tabella 15.6. Meta tag supportati

Attributo	Descrizione
`class-description`	inserito nel javadoc per le classi
`field-description`	inserito nel javadoc per i campi/proprietà
`interface`	Se è vero viene generata un'interfaccia invece di una classe.
`implements`	l'interfaccia che la classe deve implementare
`extends`	classe che dovrebbe essere estesa da questa classe (ignorata per le sottoclassi)
`generated-class`	sovrascrive il nome della vera classe generata
`scope-class`	visibilità per la classe
`scope-set`	visibilità per un metodo setter
`scope-get`	visibilità per un metodo getter
`scope-field`	visibilità per il campo vero e proprio
`use-in-tostring`	include la proprietà nel `toString()`
`implement-equals`	include un metodo `equals()` e un `hashCode()` in questa classe.
`use-in-equals`	include la proprietà nei metodi `equals()` e `hashCode()`.
`bound`	aggiunge il supporto di un propertyChangeListener per la proprietà
`constrained`	come bound + il supporto di un vetoChangeListener per una proprietà
`gen-property`	la proprietà non verrà generata se è falsa (usare con cautela)
`property-type`	Sovrascrive il tipo di default della proprietà. Da usare con l'etichetta "any" per specificare il tipo concreto invece di avere solo Object.
`class-code`	Codice extra che verrà inserito alla fine della classe
`extra-import`	Clausola di importazione extra che verrà inserita alla fine di tutte le altre
`finder-method`	vedere "generatore elementare di metodi individuatori" più sotto
`session-method`	vedere "generatore elementare di metodi individuatori" più

Attributo	Descrizione
	sotto

All'interno di un file `hbm.xml`, gli attributi dichiarati tramite l'elemento `<meta>` come comportamento predefinito vengono "ereditati".

Cosa significa? Significa che se ad esempio volete fare sì che tutte le vostre classi implementino l'interfaccia `IAuditable` dovete solo aggiungere un `<meta attribute="implements">IAuditable</meta>` all'inizio del file `hbm.xml`, proprio dopo `<hibernate-mapping>`. Ora tutte le classi definite in quel file `hbm.xml` implementeranno `IAuditable`! (Eccetto se una classe ha anche un attributo meta "implements", perché le etichette meta specificate localmente sovrascrivono/rimpiazzano sempre quelle ereditate).

Nota: questo si applica a *tutti* i `<meta>`-tag. Così può anche essere usato ad esempio per specificare che tutti i campi dovrebbero essere dichiarati protetti, invece che privati come è il comportamento predefinito. Questo si imposta aggiungendo `<meta attribute="scope-field">protected</meta>` ad esempio proprio sotto l'elemento `<class>`, e tutti i campi della classe saranno generati come protetti.

Per evitare che un `<meta>`-tag venga ereditato potete semplicemente specificare `inherit="false"` per l'attributo, ad esempio `<meta attribute="scope-class" inherit="false">public abstract</meta>` restringerà la visibilità di classe alla classe corrente, e non alle sottoclassi.

15.2.3. Generatore elementare di metodi individuatori ("finder")

Ora è possibile fare in modo tale che `hbm2java` generi dei metodi individuatori elementari per le proprietà di Hibernate. Questo richiede che due cose vengano impostate nel file `hbm.xml`.

La prima è l'indicazione di quale sia il campo per cui si vogliono generare gli individuatori. Si indica con un blocco meta all'interno di un elemento "property", come in:

```
<property name="name" column="name" type="string">
     <meta attribute="finder-method">findByName</meta>
</property>
```

Il nome del metodo individuatore sarà il testo racchiuso nelle etichette meta.

Il secondo è la creazione di un file di configurazione per hbm2java nella forma:

```
<codegen>
    <generate renderer="net.sf.hibernate.tool.hbm2java.BasicRenderer"/>
    <generate suffix="Finder" renderer="net.sf.hibernate.tool.hbm2java.FinderRenderer"/>
</codegen>
```

A questo punto si deve usare il parametro per `hbm2java --config=xxx.xml` laddove `xxx.xml` è il file di configurazione che è appena stato creato.

Un parametro opzionale è l'etichetta meta al livello di classe nel formato:

```
<meta attribute="session-method">
    com.whatever.SessionTable.getSessionTable().getSession();
</meta>
```

Che sarebbe il modo in cui si reperiscono le sessioni se usate il pattern *Thread Local Session* (documentato nell'area "Design Patterns" del sito web di Hibernate).

15.2.4. Generatore basato su Velocity

Ora è possibile usare Velocity come strumento alternativo di resa/generazione. Il seguente file config.xml mostra come configurare hbm2java per usare il generatore basato su velocity.

```
<codegen>
 <generate renderer="net.sf.hibernate.tool.hbm2java.VelocityRenderer">
  <param name="template">pojo.vm</param>
 </generate>
</codegen>
```

Il parametro `template` è un percorso di risorsa che punta al file delle macro velocity macro che volete usare. Il file deve essere raggiungibile sul classpath di hbm2java: per questo ricordate di aggiungere al vostro task di ant o script di shell la directory in cui si trova pojo.vm (la posizione predefinita è `./tools/src/velocity`)

Ricordatevi che la versione attuale di `pojo.vm` genera solo le parti più elementari dei bean java. Non è tanto completa e ricca di funzionalità quanto il generatore nativo di Hibernate - in particolare non sono supportati la maggior parte dei tag `meta`.

15.3. Generazione dei file di mappaggio

È possibile generare uno scheletro di file di mappaggio a partire da classi persistenti compilate usando una utilità a linea di comando chiamata `MapGenerator`, parte del pacchetto delle estensioni di Hibernate (Hibernate Extensions).

Il generatore di mappaggio fornisce un meccanismo per produrre mappaggi dalle classi compilate. Usa il meccanismo della "reflection" java per trovare le *proprietà* e usa dei metodi euristici per indovinare un mappaggio appropriato per il tipo di proprietà. Il mappaggio generato è inteso solo come un punto di partenza: non c'è modo di produrre un mappaggio completo senza informazioni extra fornite dall'utente. In ogni modo, questo strumento libera da una parte del lavoro ripetitivo e bruto coinvolto nella produzione di un mappaggio.

Le classi vengono aggiunte al mappaggio una alla volta. Lo strumento rigetterà le classi che a suo giudizio non siano *persistibili tramite Hibernate*.

Per essere *persistibile tramite Hibernate* una classe

- non deve essere un tipo primitivo
- non dev'essere un array
- non deve essere un'interfaccia
- non deve essere una classe annidata
- deve avere un costruttore di default (senza argomenti).

Notate che le interfacce e le classi annidate in realtà sono persistibili da Hibernate, ma questo non è solitamente ciò che l'utente vuole.

`MapGenerator` risalirà la catena delle superclassi di tutte le classi aggiunte tentando di aggiungere quante più superclassi possibile (persistibili da Hibernate) alla stessa tabella di database. La ricerca si ferma non appena viene trovata una proprietà che ha un nome che appare in una lista di *nomi candidati come UID*.

La lista predefinita di nomi di proprietà candidati come UID è: `uid`, `UID`, `id`, `ID`, `key`, `KEY`, `pk`, `PK`.

Le proprietà vengono reperite quando ci sono due metodi nella classe, un "setter" (impostatore) e un "getter" (recuperatore), laddove il tipo del singolo argomento dell'impostatore è lo stesso del tipo di ritorno del recupe-

ratore (che non deve avere argomenti), mentre l'impostatore restituisce `void`. Inoltre, il nome dell'impostatore deve cominciare con la stringa `set` e deve essere vero o che il nome del recuperatore comincia con `get` o che comincia con `is` e il tipo della proprietà è boolean. In entrambi i casi, il resto dei nomi deve conocordare. Questa porzione corrispondente è il nome della proprietà, eccettuato il fatto che il carattere iniziale del nome della proprietà è reso minuscolo se la seconda lettera è minuscola.

Le regole per determinare il tipo (sul database) di ogni proprietà sono:

1. Se il tipo java è `Hibernate.basic()`, la proprietà è una colonna di quel tipo.
2. Per i tipi personalizzati `hibernate.type.Type` e `PersistentEnum` viene usata una colonna semplice, nello stesso modo.
3. Se il tipo della proprietà è un array, viene usato un array di Hibernate, e `MapGenerator` tenta di riflettere (ovvero agire via "reflection") sul tipo di elemento dell'array.
4. Se la proprietà ha tipo `java.util.List`, `java.util.Map`, o `java.util.Set`, vengono usati i corrispondenti tipi di Hibernate, ma `MapGenerator` non può procedere oltre nel lavorare sull'interno di questi tipi.
5. Se il tipo della proprietà è qualsiasi altra classe, `MapGenerator` rimanda la decisione sulla rappresentazione sul database finché tutte le classi non sono state processate. A questo punto, se la classe era stata reperita tramite la ricerca per superclassi descritta sopra, allora la proprietà è una associazione `many-to-one`. Se la classe ha delle proprietà, allora è un `component`. In caso contrario è serializzabile, o non persistibile.

15.3.1. Esecuzione dello strumento

Lo strumento scrive mappaggi XML sull'uscita standard e/o su un file.

Quando invocate lo strumento, dovete mettere sul classpath le vostre classi compilate.

`java -cp` *classpath_di_hibernate_e_delle_vostre_classi* `net.sf.hibernate.tool.class2hbm.MapGenerator` *opzioni e nomi delle classi*

Ci sono due modi di funzionamento: a linea di comando o interattivo.

La modalità interattiva viene selezionata fornendo sulla linea di comando il parametro `--interact`. Questa modalità fornisce una console con un "prompt" (cursore di inserimento comandi). Usandola potete settare il nome della proprietà UID per ogni classe usando il comando `uid=XXX` in cui `XXX` è il nome della proprietà UID. Altre alternative di comandi sono semplicemente un nome di classe completamente qualificato (cioè con la parte relativa ai package, come in java.lang.String), o il comando "done" che emette l'XML è termina.

In modalità a linea di comando i parametri sono le opzioni che seguono, inframmezzate dai nomi completamente qualificati delle classi che vanno processate. La maggior parte delle opzioni sono intese come utilizzabili più volte; ogni uso coinvolge le classi che vengono aggiunte conseguentemente.

Tabella 15.7. Opzioni da linea di comando del MapGenerator

Opzione	Descrizione
`--quiet`	non scrive il mappaggio O-R sullo standard output
`--setUID=uid`	imposta la lista di UID candidati all'uid singolo
`--addUID=uid`	aggiunge uid in cima alla lista di UID candidati
`--select=`*modalità*	usa la modalità di selezione *modalità*(e.g., *distinct* o *all*) per le classi aggiunte in seguito
-	limita la profondità della ricorsione dei dati dei componenti per le classi

Opzione	Descrizione
`-dep-th=<piccolo-valore-intero>`	aggiunte in seguito
`--output=mio_file.xml`	scrive il mappaggio OR su un file
nome.completo.di.Classe	aggiunge la classe al mappaggio
`--abstract=`*nome.completo.di.Classe*	vedi sotto

Il parametro "abstract" istruisce lo strumento di generazione del mappaggio in modo tale da ignorare superclassi specifiche in modo che classi con ereditarietà comune non vengano mappate su una sola grande tabella. Ad esempio, considerate queste gerarchie di classe:

```
Animale-->Mammifero-->Umano

Animale-->Mammifero-->Marsupiale-->Canguro
```

Se il parametro `--abstract` *non* viene usato, tutte le classi verranno mappate come sottoclassi di `Animale`, il che risulterà in una grande tabella che contiene tutte le proprietà di tutte le classi più una colonna discriminatore che indicherà quale sottoclasse è realmente memorizzata in una riga. Se `Mammifero` è marcata come `abstract`, `Umano` e `Marsupiale` verranno mappate in dichiarazioni `<class>` separate e memorizzate in tabelle separate. `Canguro` sarà ancora una sottoclasse di `Marsupiale` a meno che anche `Marsupiale` sia marchiata come `abstract`.

Capitolo 16. Esempio: Genitore/Figlio (Parent/Child)

Una delle primissime cose che i nuovi utenti tentano di fare con Hibernate è modellare una relazione di tipo genitore / figlio. Ci sono due approcci differenti, per farlo. Per varie ragioni, l'approccio più conveniente, soprattutto per i neofiti, è modellare sia `Parent` sia `Child` come classi di entità con una associazione `<one-to-many>` da `Parent` a `Child`. (L'approccio alternativo è dichiarare il `Child` come un `<composite-element>`.) Ora, la semantica predefinita di una associazione uno-a-molti in Hibernate è molto meno affine alla semantica usuale di una relazione genitore - figlio di quanto non lo sia quella di un mappaggio ad elemento composito. Mostreremo ora come usare una *associazione uno a molti bidirezionale e con cascate* per modellare una relazione genitore / figlio in maniera efficiente. Non è per niente difficile!

16.1. Una nota sulle collezioni

Le collezioni di Hibernate vengono considerate logicamente parte della entità che le possiede, e mai delle entità contenute. Questa è una precisazione cruciale, che ha le seguenti conseguenze:

- Quando rimuoviamo / aggiungiamo un oggetto da / a una collezione, il numero di versione del proprietario viene incrementato.
- Se un oggetto che è stato rimosso da una collezione è un'istanza di un tipo di valore ("value type"), cioè un elemento composito, quell'oggetto cesserà di essere persistente e il suo stato verrà completamente rimosso dal database. Nello stesso modo, aggiungendo una istanza di un tipo di valore alla collezione causerà il fatto che il suo stato sarà reso persistente.
- Dall'altro lato, se un'entità viene rimossa da una collezione (che sia associata uno-a-molti o molti-a-molti), non verrà cancellata, come funzionamento predefinito. Questo comportamento è del tutto coerente - un cambiamento allo stato interno di un'altra entità non dovrebbe causare il fatto che l'entità associata svanisca! Nello stesso modo, l'aggiunta di un'entità a una collezione non causa il fatto che quell'entità venga automaticamente resa persistente (nel comportamento predefinito).

Invece, il comportamento standard prevede che aggiungere un'entità a una collezione si limiti a creare un collegamento tra le due entità, così come rimuoverla determinerà la rimozione di quel collegamento. Questo funzionamento è il più appropriato per moltissimi casi, mentre non è appropriato per nulla nel caso di una relazione genitore / figlio in cui la vita del figlio sia legata al ciclo di vita del genitore.

16.2. Uno-a-molti bidirezionale

Supponete che cominciamo con una semplice associazione `<one-to-many>` da `Parent` a `Child`.

```
<set name="children">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

Se dovessimo eseguire il codice seguente

```
Parent p = .....;
Child c = new Child();
p.getChildren().add(c);
session.save(c);
session.flush();
```

Hibernate produrrebbe le due istruzioni SQL che seguono:

- una `INSERT` per creare il record per `c`
- una `UPDATE` per creare il collegamento da `p` a `c`

Questo non solo è inefficiente, ma viola anche i vincoli `NOT NULL` sulla colonna `parent_id`.

La causa soggiacente è che il collegamento (la chiave esterna `parent_id`) da `p` a `c` non viene considerata parte dello stato dell'oggetto `Child` è quindi non viene creata nell'istruzione `INSERT`. La soluzione è quindi fare in modo che il collegamento sia parte del mappaggio di `Child`.

```
<many-to-one name="parent" column="parent_id" not-null="true"/>
```

(Abbiamo anche bisogno di aggiungere la proprietà `parent` sulla classe `Child`.)

Ora che l'entità `Child` gestisce lo stato del collegamento, diciamo alla collezione di non aggiornarlo. Usiamo quindi l'attributo `inverse`.

```
<set name="children" inverse="true">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

Per aggiungere un nuovo `Child` verrebbe allora usato il codice seguente:

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
c.setParent(p);
p.getChildren().add(c);
session.save(c);
session.flush();
```

E ora verrà generata una sola `INSERT` SQL!

Per facilitare un po' le cose, possiamo creare un metodo `addChild()` al `Parent`.

```
public void addChild(Child c) {
    c.setParent(this);
    children.add(c);
}
```

A questo punto il codice per aggiungere un `Child` appare così:

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
session.save(c);
session.flush();
```

16.3. Ciclo di vita con cascate

La chiamata esplicita a `save()` ci infastidisce ancora. Abbiamo quindi bisogno di gestire la situazione usando le cascate.

```
<set name="children" inverse="true" cascade="all">
    <key column="parent_id"/>
```

```
    <one-to-many class="Child"/>
</set>
```

Questo semplifica il codice seguente in questo modo:

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = new Child();
p.addChild(c);
session.flush();
```

In maniera similare, non abbiamo bisogno di iterare sui figli per salvare o cancellare un `Parent`. Quanto segue rimuove `p` e tutti i suoi figli dal database.

```
Parent p = (Parent) session.load(Parent.class, pid);
session.delete(p);
session.flush();
```

Però il codice seguente

```
Parent p = (Parent) session.load(Parent.class, pid);
Child c = (Child) p.getChildren().iterator().next();
p.getChildren().remove(c);
c.setParent(null);
session.flush();
```

non rimuoverà ancora `c` from the database; rimuoverà solo il link verso `p` (e causerà la violazione di vincolo `NOT NULL`, in questo caso). C'è bisogno di cancellare esplicitamente (`delete()`) il `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();
```

Ora, nel nostro caso un `Child` non può esistere senza il suo genitore. Quindi se rimuoviamo un `Child` dalla collezione, vogliamo che venga cancellato davvero. Per questo, dobbiamo usare `cascade="all-delete-orphan"`.

```
<set name="children" inverse="true" cascade="all-delete-orphan">
    <key column="parent_id"/>
    <one-to-many class="Child"/>
</set>
```

Nota: anche se il mappaggio della collezione specifica `inverse="true"`, le cascate sono comunque gestite iterando sugli elementi della collezione. Quindi, se avete bisogno che un oggetto venga salvato, cancellato, o aggiornato per cascata, dovete aggiungerlo alla collezione. Non è sufficiente chiamare solo `setParent()`.

16.4. Come utilizzare `update()` in cascata

Immaginate che carichiamo un `Parent` in una `Session`, facciamo qualche cambiamento ad una azione di interfaccia e vogliamo rendere persistenti questi cambiamenti in una nuova Session (chiamando `update()`). Il `Parent` conterrà una collezione di figli e, poiché è abilitato l'aggiornamento in cascata, Hibernate ha bisogno di sapere quali figli siano appena stati istanziati, e quali invece rappresentino righe già esistenti nel database. Assumiamo che sia il `Parent` sia il `Child` abbiano proprietà di identificazione (sintetiche) di tipo `java.lang.Long`. Hibernate userà il valore della proprietà identificatore per determinare quali dei figli sono nuovi. (Potete anche usare le proprietà versione o marca di tempo (timestamp), vedete Sezione 9.4.2, “Aggiornamento di oggetti sganciati”.)

L'attributo `unsaved-value` viene usato per specificae il valore di identificatore di una istanza appena creata. Se non specificato, `unsaved-value` vale "null", il che è perfetto, per un identificatore di tipo `Long`. Se avessimo usato una proprietà di identificazione di un tipo primitivo, dovremmo specificare

```
<id name="id" type="long" unsaved-value="0">
```

per il mappaggio del `Child`. (C'è anche un attributo `unsaved-value` per i mappaggi di proprietà di versione e timestamp.)

Il codice seguente aggiornerà il `parent` e `child` e inserirà `newChild`.

```
//parent e child sono già stati caricati in una sessione precedente
parent.addChild(child);
Child newChild = new Child();
parent.addChild(newChild);
session.update(parent);
session.flush();
```

Bene, questo è perfetto per il caso in cui si abbia un identificatore generato automaticamente, ma cosa succede quando si hanno identificatori assegnati manualmente e identificatori composti? In questo caso è più difficile, perché `unsaved-value` non può distinguere tra un oggetto appena istanziato (con identificatore assegnato dall'utente) e un oggetto caricato in una sessione precedente). In questi casi, avrete probabilmente bisogno di dare una mano ad Hibernate, o

- definendo `unsaved-value="null"` o `unsaved-value="negative"` su una proprietà `<version>` o `<timestamp>` per la classe.
- impostare `unsaved-value="none"` e salvare esplicitamente (con `save()`) i figli appena istanziati prima di chiamare `update(parent)`
- impostare `unsaved-value="any"` ed aggiornare esplicitamente (con `update()`) i figli precedentemente resi persistenti prima di chiamare `update(parent)`

`none` è il valore `unsaved-value` predefinito per gli identificatori assegnati e composti.

C'è una possibilità ulteriore. C'è un nuovo metodo sulla classe `Interceptor` che si chiama `isUnsaved()` che consente all'applicazione di implementare la propria strategia per distinguere gli oggetti appena istanziati. Ad esempio, potreste definire una classe di base per le vostre classi persistenti.

```
public class Persistent {
    private boolean _saved = false;
    public void onSave() {
        _saved=true;
    }
    public void onLoad() {
        _saved=true;
    }
    ......
    public boolean isSaved() {
        return _saved;
    }
}
```

(La proprietà `saved` è non-persistente.) Ora implementate `isUnsaved()` insieme a `onLoad()` e `onSave()` come segue:

```
public Boolean isUnsaved(Object entity) {
    if (entity instanceof Persistent) {
        return new Boolean( !( (Persistent) entity ).isSaved() );
```

```
    }
    else {
        return null;
    }
}

public boolean onLoad(Object entity,
    Serializable id,
    Object[] state,
    String[] propertyNames,
    Type[] types) {

    if (entity instanceof Persistent) ( (Persistent) entity ).onLoad();
    return false;
}

public boolean onSave(Object entity,
    Serializable id,
    Object[] state,
    String[] propertyNames,
    Type[] types) {

    if (entity instanceof Persistent) ( (Persistent) entity ).onSave();
    return false;
}
```

16.5. Conclusione

Ci sono vari concetti da digerire, qui, e potrebbe sembrare confuso, in un primo momento. Comunque, nella pratica funziona tutto molto bene. La maggior parte delle applicazioni basate su Hibernate usando il pattern genitore / figlio in vari posti.

Abbiamo menzionato un'alternativa nel primo paragrafo. Nessuna delle questioni precedenti esiste nel caso di mappaggi con `<composite-element>`, che hanno esattamente la semantica di una relazione padre / figlio. Sfortunatamente ci sono due grosse limitazioni per gli elementi composti: non possono avere collezioni, e non dovrebbero essere figli di un'entità diversa dal loro genitore unico. (Per quanto, *possano* avere una chiave primaria surrogata usando il mappaggio `<idbag>`.)

Capitolo 17. Esempio: una applicazione che realizza un weblog

17.1. Classi persistenti

Le classi persistenti rappresentano un weblog, e un articolo pubblicato su di esso. Devono venire modellate come una relazione standard genitore/figlio, ma useremo un "bag" ordinato invece di un insieme.

```
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;
    }
}
```

17.2. Mappaggi di hibernate

I mappaggi XML sono abbastanza semplici.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
        "-//Hibernate/Hibernate Mapping DTD 2.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping package="eg">

    <class
        name="Blog"
        table="BLOGS"
        lazy="true">

        <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"
            lazy="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 2.0//EN"
```

```
        "http://hibernate.sourceforge.net/hibernate-mapping-2.0.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>
```

17.3. Codice di Hibernate

La classe seguente mostra il genere di operazioni che possiamo effettuare su queste classi tramite Hibernate.

```
package eg;

import java.util.ArrayList;
import java.util.Calendar;
import java.util.Iterator;
import java.util.List;

import net.sf.hibernate.HibernateException;
import net.sf.hibernate.Query;
import net.sf.hibernate.Session;
import net.sf.hibernate.SessionFactory;
import net.sf.hibernate.Transaction;
import net.sf.hibernate.cfg.Configuration;
import net.sf.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.save(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.list().get(0);
            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;
```

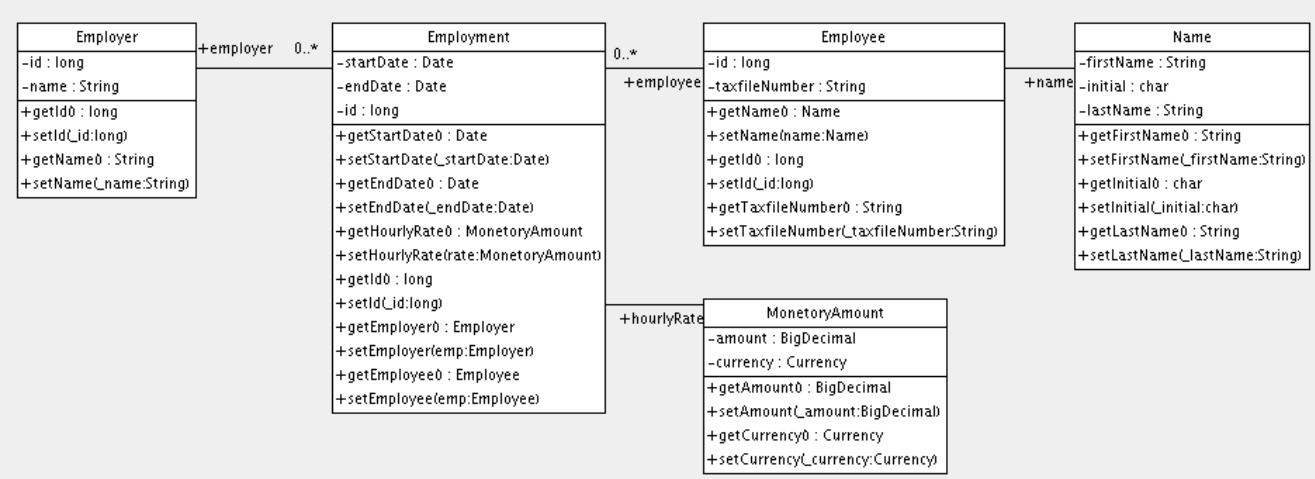
```
    }
}
```

Capitolo 18. Alcuni mappaggi di esempio

Queste sezioni vi mostrano alcuni mappaggi di associazioni complesse.

18.1. Employer/Employee (Datore di lavoro / impiegato)

Il modello seguente della relazione tra `Employer` e `Employee` usa una vera classe di entità (`Employment`) per rappresentare l'associazione. Facciamo in questo modo perché potrebbe esserci più di un periodo di impiego che lega gli stessi due dipendenti. Per modellizzare i valori monetari e i nomi degli impiegati vengono usati dei componenti.



Ecco un possibile documento di mappaggio:

```
<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="MonetoryAmount">
            <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">
```

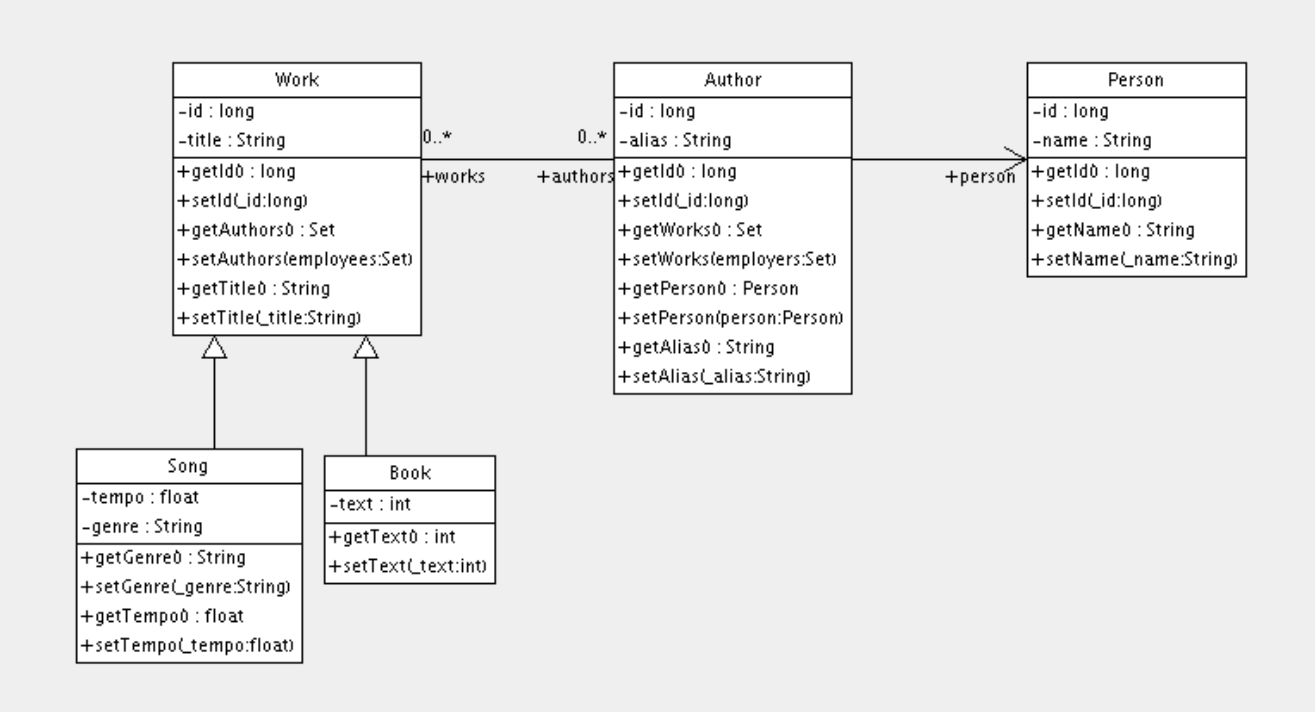
```
        <id name="id">
            <generator class="sequence">
                <param name="sequence">employee_id_seq</param>
            </generator>
        </id>
        <property name="taxfileNumber"/>
        <component name="name" class="Name">
            <property name="firstName"/>
            <property name="initial"/>
            <property name="lastName"/>
        </component>
    </class>

</hibernate-mapping>
```

Ed ecco lo schema delle tabelle generato automaticamente da `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
```

18.2. Autore/Opera (Author/Work)

Considerate il seguente modello per le relazioni tra `Work`, `Author` e `Person`. Rappresentiamo la relazione tra `Work` e `Author` (tra un'opera e il suo autore) come una associazione molti-a-molti. Abbiamo invece scelto di rappresentare la relazione tra `Author` e `Person` come una associazione uno-a-uno. Un'altra possibilità sarebbe che `Author` estendesse `Person`.



Il seguente documento di mappaggio rappresenta correttamente queste relazioni:

```
<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" lazy="true">
            <key>
                <column name="work_id" not-null="true"/>
            </key>
            <many-to-many class="Author">
                <column name="author_id" not-null="true"/>
            </many-to-many>
        </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">
            <!-- L'autore deve avere lo stesso identificatore della persona (Person) -->
            <generator class="assigned"/>
        </id>

        <property name="alias"/>
        <one-to-one name="person" constrained="true"/>

        <set name="works" table="author_work" inverse="true" lazy="true">
            <key column="author_id"/>
            <many-to-many class="Work" column="work_id"/>
```

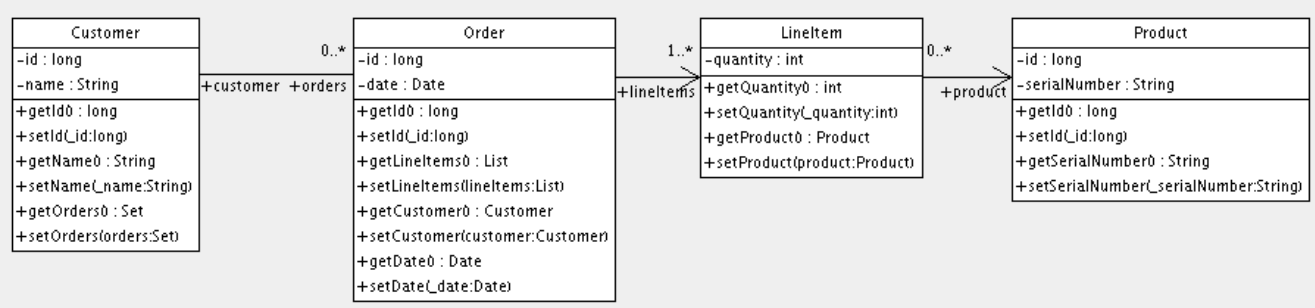
```
        </set>

    </class>

    <class name="Person" table="persons">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
    </class>

</hibernate-mapping>
```

In questo file di mappaggio ci sono quattro tabelle: `works`, `authors` e `persons` contengano le opere, i dati degli autori e i dati delle persone. `author_work` è una tabella di associazione che collega gli autori alle opere. Ecco lo schema delle tabelle così come generato da `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
```

18.3. Cliente/Ordine/Prodotto (Customer/Order/Product)

Ora consideriamo un modello per le relazioni tra `Customer` (cliente), `Order` (ordine), `LineItem` (linea d'ordine) e `Product` (prodotto). C'è una associazione uno-a-molti tra `Customer` e `Order`, ma come potremmo rappresentare la relazione `Order` / `LineItem` / `Product`? Abbiamo scelto di mappare `LineItem` come una classe di associazione che rappresenti la relazione molti-a-molti tra gli `Order` e i `Product`. In Hibernate, questo si chiama un elemento composito.



Il documento di mappaggio è:

```
<hibernate-mapping>

    <class name="Customer" table="customers">
        <id name="id">
            <generator class="native"/>
        </id>
        <property name="name"/>
        <set name="orders" inverse="true" lazy="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" lazy="true">
            <key column="order_id"/>
            <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>
```

Le tabelle`customers`, `orders`, `line_items` e `products` contengono rispettivamente dati dei clienti, degli ordini, delle linee d'ordine e dei prodotti. `line_items` svolge anche il ruolo di tabella di associazione tra gli ordini e i prodotti.

```
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
```

Capitolo 19. Buone abitudini (best practices)

Scrivete classi a granularità fine, e mappatele usando `<component>`.

Usate una classe `Address` (indirizzo) per incapsulare `street` (via), `suburb` (comune), `state` (stato), `postcode` (codice postale). Questa pratica facilita il riuso del codice e la sua ristrutturazione (refactoring).

Dichiarate proprietà identificatrici sulle classi persistenti.

In Hibernate le proprietà di identificazione sono opzionali, tuttavia ci sono molte buone ragioni per cui è preferibile utilizzarle. Raccomandiamo che gli identificatori siano 'sintetici' (ovvero generati, senza altro significato applicativo), e di un tipo non-primitivo. Per la massima flessibilità usate `java.lang.Long` o `java.lang.String`.

Mettete ogni mappaggio di classe in un file separato.

Non usate un unico documento di mappaggio monolitico. Mappate `com.eg.Foo` nel file `com/eg/Foo.hbm.xml`. Questo è particolarmente utile nel lavoro di gruppo.

Caricate i mappaggi come risorse.

Distribuite i mappaggi insieme alle classi che mappano.

Prendete in considerazione l'esternalizzazione rispetto al codice delle stringhe di interrogazione.

Questa è una buona pratica in particolare se le vostre interrogazioni chiamano delle funzioni SQL non ansi-standard. Esternalizzare le stringhe nei file di mappaggio renderà l'applicazione più portabile.

Usate variabili di sostituzione.

Come in JDBC, sostituite sempre i valori non constanti con "?" nelle query.Non usate mai manipolazione di stringhe per sostituire un valore non costante in una interrogazione! È anche meglio prendere in considerazione l'uso di parametri con nome, nelle interrogazioni.

Non gestite le connessioni JDBC per conto vostro.

Hibernate permette all'applicazione di gestire le connessioni JDBC. Questo approccio dovrebbe essere considerato un'ultima spiaggia. Se non potete usare i fornitori di connessione predefiniti, considerate la possibilità di implementare voi stessi l'interfaccia `net.sf.hibernate.connection.ConnectionProvider`.

Valutate l'uso di un tipo proprietario("custom type").

Immaginate di avere un tipo di oggetto java, ad esempio proveniente da una libreria, che abbia bisogno di essere reso persistente, ma non fornisca i metodi di accesso necessari per mapparlo come un componente. Dovreste valutare la possibilità di implementare `net.sf.hibernate.UserType`. Questo approccio libera il codice applicativo dalla necessità di implementare trasformazioni da/a un tipo di Hibernate.

Usate codice JDBC scritto a mano nei colli di bottiglia.

Nelle aree critiche rispetto alle performance del sistema, alcune operazioni (ad esempio cancellazioni o aggiornamenti massicci) potrebbero beneficiare da un'implementazione diretta in JDBC. Ma vi preghiamo di attendere fino a che non *sappiate* con certezza che qualcosa è un collo di bottiglia. Non assumete inoltre, che il JDBC diretto sia necessariamente più veloce: se avete bisogno di usarlo, potrebbe essere sensato aprire una `Session` di Hibernate e usare la sottostante connessione SQL. In questo modo potete comunque usare la stessa strategia transazionale e il fornitore di connessioni sottostante.

Comprendete i meccanismi di scaricamento (flushing) della `Session`.

Di tanto in tanto la Session sincronizza il suo stato persistente con il database. Le performance saranno coinvolte se questo processo capita troppo spesso. Potete a volte minimizzare la quantità di scaricamenti non necessari disabilitando i meccanismi automatici, o anche cambiando l'ordine delle interrogazioni e delle altre operazioni all'interno di una particolare transazione.

In un'architettura a tre livelli, valutate l'uso di `saveOrUpdate()`.

Quando usate una architettura basata su servlet / session bean, potreste passare gli oggetti persistenti caricati nel session bean da e al servlet o allo strato delle jep. Usate una nuova sessione per gestire ogni richiesta. Usate poi `Session.update()` o `Session.saveOrUpdate()` per aggiornare lo stato persistente di un oggetto.

In un'architettura a due livelli, valutate l'use della disconnessione delle sessioni.

Quando usate solo un servlet, potete riutilizzare la stessa sessione per richieste multiple dei client. Semplicemente ricordate di sconnettere la sessione prima di restituire il controllo al client.

Non trattate le eccezioni come se fossero recuperabili.

Questa, più che una migliore pratica, è una pratica necessaria. Quando capita un'eccezione, fate il rollback della `Transaction` e chiudete la `Session`. Se non lo fate, Hibernate non può garantire che lo stato in memoria rappresenti accuratamente lo stato persistente. Come caso particolare, non usate `Session.load()` per determinare se un'istanza con quel particolare identificatore esista sul database; usate `find()`, invece.

Preferite il caricamento differito (lazy) per le associazioni.

Usate con moderazione il caricamento diretto (via outer-join). Usate i mediatori (proxy) e/o le collezioni a caricamento differito per la maggior parte delle associazioni con classi che non siano messe in cache a livello della JVM. Per le associazioni con le classi in cache, dove ci sia un'alta possibilità di avere gli oggetti in cache disabilitate esplicitamente il caricamento diretto usando `outer-join="false"`. Se in un particolare caso dovesse essere appropriato un caricamento diretto con outer-join, potete usare una interrogazione con un `left join`.

Valutate la possibilità di separare la logica di business da Hibernate.

Mascherate il codice di accesso ai dati (via hibernate) dietro un'interfaccia. Combinate i pattern *DAO* e *Thread Local Session*. Potete anche avere alcune classi rese persistenti da codice JDBC manuale, associate ad Hibernate tramite uno `UserType`. (Questo consiglio ha senso per applicazioni "sufficientemente grandi", non è appropriato per un'applicazione con poche tabelle!)