EBook Gratuito

APPRENDIMENTO hibernate

Free unaffiliated eBook created from
Stack Overflow contributors.

#hibernate

Sommario

Di

Capitolo 1: Iniziare con l'ibernazione

Osservazioni

Il bean `SessionFactory` è responsabile della creazione, della manutenzione, della chiusura e dello svuotamento di tutte le sessioni del database che `TransactionManager` richiede di creare. Ecco perché autowire `SessionFactory` in DAO e fare eseguire tutte le query attraverso di esso.

Una delle domande più importanti che i nuovi utenti di Hibernate chiedono è "Quando vengono eseguite le mie modifiche?" e la risposta ha senso quando si pensa a come `TransactionManager` funziona con `SesisonFactory` . Le modifiche al database verranno svuotate e impegnate quando si esce dal metodo di servizio annotato con `@Transactional` . La ragione di ciò è che una transazione dovrebbe rappresentare una singola 'unità' di lavoro ininterrotto. Se qualcosa va storto con l'unità, allora si presume che l'unità abbia fallito e tutte le modifiche dovrebbero essere ripristinate. Quindi `SessionFactory` cancellerà e cancellerà la sessione quando esci dal metodo di servizio che hai chiamato in origine.

Questo non vuol dire che non svuoterà e cancellerà la sessione mentre la transazione è in corso. Ad esempio, se chiamo un metodo di servizio per aggiungere una raccolta di 5 oggetti e restituisco il conteggio totale degli oggetti nel database, `SessionFactory` si renderà conto che la query (`SELECT COUNT(*)`) richiede che uno stato aggiornato sia accurato e quindi svuoterebbe l'aggiunta dei 5 oggetti prima di eseguire la query di conteggio. L'esecuzione potrebbe essere simile a questa:

Versioni

Versione	Link alla documentazione	Data di rilascio
4.2.0	http://hibernate.org/orm/documentation/4.2/	2013/03/01
4.3.0	http://hibernate.org/orm/documentation/4.3/	2013/12/01
5.0.0	http://hibernate.org/orm/documentation/5.0/	2015/09/01

Examples

Utilizzo della configurazione XML per configurare Hibernate

Creo un file chiamato `database-servlet.xml` da qualche parte sul classpath.

Inizialmente il tuo file di configurazione sarà simile a questo:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xmlns:jdbc="http://www.springframework.org/schema/jdbc"
xmlns:tx="http://www.springframework.org/schema/tx"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.2.xsd
    http://www.springframework.org/schema/jdbc
http://www.springframework.org/schema/jdbc/spring-jdbc-3.2.xsd
    http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-
tx-3.2.xsd">

</beans>
```

Noterai che ho importato i namespace `tx` e `jdbc` Spring. Questo perché li useremo abbastanza pesantemente in questo file di configurazione.

La prima cosa che vuoi fare è abilitare la gestione delle transazioni basata sull'annotazione (`@Transactional`). La ragione principale per cui le persone usano Hibernate in primavera è perché Spring gestirà tutte le transazioni per te. Aggiungi la seguente riga al tuo file di configurazione:

```
<tx:annotation-driven />
```

Abbiamo bisogno di creare una fonte di dati. L'origine dati è fondamentalmente il database che Hibernate utilizzerà per mantenere i tuoi oggetti. Generalmente un gestore delle transazioni avrà un'origine dati. Se vuoi che Hibernate parli con più fonti di dati, allora hai più gestori di transazioni.

```
<bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="" />
    <property name="url" value="" />
    <property name="username" value="" />
    <property name="password" value="" />
</bean>
```

La classe di questo bean può essere qualsiasi cosa che implementa `javax.sql.DataSource` modo che tu possa scrivere la tua. Questa classe di esempio è fornita da Spring, ma non ha un proprio pool di thread. Un'alternativa popolare è Apache Commons `org.apache.commons.dbcp.BasicDataSource` , ma ce ne sono molti altri. Spiegherò ognuna delle proprietà qui sotto:

- *driverClassName* : il percorso del driver JDBC. Questo è un JAR **specifico del database** che dovrebbe essere disponibile sul classpath. Assicurati di avere la versione più aggiornata. Se si utilizza un database Oracle, è necessario un OracleDriver. Se hai un database MySQL, avrai bisogno di un MySQLDriver. Vedi se riesci a trovare il driver che ti serve qui, ma un google veloce dovrebbe darti il driver corretto.

- *url* : l'URL del tuo database. Di solito questo sarà qualcosa come `jdbc\:oracle\:thin\:\path\to\your\database` o `jdbc:mysql://path/to/your/database` . Se cerchi google in giro per il percorso predefinito del database che stai utilizzando, dovresti essere in grado di scoprire cosa dovrebbe essere. Se ricevi una `HibernateException` con il messaggio `org.hibernate.HibernateException: Connection cannot be null when 'hibernate.dialect' not set` e stai seguendo questa guida, c'è il 90% di possibilità che il tuo URL sia sbagliato, una probabilità del 5% che il tuo database non è stato avviato e un 5% di probabilità che il tuo

nome utente / password sia sbagliato.

- *username* : il nome utente da utilizzare durante l'autenticazione con il database.
- *password* : la password da utilizzare durante l'autenticazione con il database.

La prossima cosa è impostare `SessionFactory` . Questa è la cosa che Hibernate usa per creare e gestire le tue transazioni e in effetti parla con il database. Ha parecchie opzioni di configurazione che cercherò di spiegare sotto.

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="packagesToScan" value="au.com.project />
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.use_sql_comments">true</prop>
            <prop key="hibernate.hbm2ddl.auto">validate</prop>
        </props>
    </property>
</bean>
```

- *dataSource* : il tuo bean di origine dati. Se hai cambiato l'Id di dataSource, impostalo qui.
- *packagesToScan* : i pacchetti da scansionare per trovare gli oggetti annotati JPA. Questi sono gli oggetti che la factory di sessione deve gestire, generalmente saranno POJO e annotati con `@Entity` . Per ulteriori informazioni su come impostare le relazioni tra oggetti in Hibernate, vedere qui .
- *annotatedClasses* (non mostrato): puoi anche fornire un elenco di classi per Hibernate da analizzare se non sono tutte nello stesso pacchetto. Dovresti usare i `packagesToScan` `annotatedClasses` o `annotatedClasses` ma non entrambi. La dichiarazione assomiglia a questo:

```
<property name="annotatedClasses">
    <list>
        <value>foo.bar.package.model.Person</value>
        <value>foo.bar.package.model.Thing</value>
    </list>
</property>
```

- *HibernateProperties* : ci sono una miriade di questi documenti amorevolmente documentati qui . I principali che userete sono i seguenti:
- *hibernate.hbm2ddl.auto* : una delle domande più calde di Hibernate descrive questa proprietà. Guardalo per maggiori informazioni Generalmente utilizzo validate e configuro il mio database usando gli script SQL (per una memoria in-house) o creando preventivamente il database (database esistente).
- *hibernate.show_sql* : Boolean flag, se true Hibernate stamperà tutto lo SQL che genera sullo `stdout` . Puoi anche configurare il tuo logger per mostrare i valori che sono associati alle query impostando `log4j.logger.org.hibernate.type=TRACE` `log4j.logger.org.hibernate.SQL=DEBUG` nel tuo log manager (io uso log4j).
- *hibernate.format_sql* : Boolean flag, farà in modo che Hibernate stampa piuttosto il tuo SQL

sullo stdout.
- *hibernate.dialect* (non mostrato, per una buona ragione): un sacco di vecchi tutorial là fuori ti mostrano come impostare il dialetto Hibernate che userà per comunicare con il tuo database. Hibernate **può** rilevare automaticamente quale dialetto utilizzare in base al driver JDBC che si sta utilizzando. Dato che ci sono circa 3 diversi dialetti Oracle e 5 diversi dialetti MySQL, lascerei questa decisione fino a Hibernate. Per un elenco completo dei dialetti, i supporti di Hibernate vedono qui .

Gli ultimi 2 fagioli che devi dichiarare sono:

```
<bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"
    id="PersistenceExceptionTranslator" />

<bean id="transactionManager"
    class="org.springframework.orm.hibernate4.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

`PersistenceExceptionTranslator` traduce le specifiche `HibernateException` o `SQLExceptions` specifiche del database in eccezioni Spring che possono essere comprese dal contesto dell'applicazione.

Il bean `TransactionManager` è ciò che controlla le transazioni e i rollback.

Nota: è necessario automatizzare il bean `SessionFactory` nei DAO.

Configurazione di Hibernate senza XML

Questo esempio è stato preso da qui

```
package com.reborne.SmartHibernateConnector.utils;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class LiveHibernateConnector implements IHibernateConnector {

    private String DB_DRIVER_NAME = "";
    private String DB_URL = "jdbc:h2:~/liveDB;MV_STORE=FALSE;MVCC=FALSE";
    private String DB_USERNAME = "sa";
    private String DB_PASSWORD = "";
    private String DIALECT = "org.hibernate.dialect.H2Dialect";
    private String HBM2DLL = "create";
    private String SHOW_SQL = "true";

    private static Configuration config;
    private static SessionFactory sessionFactory;
    private Session session;

    private boolean CLOSE_AFTER_TRANSACTION = false;

    public LiveHibernateConnector() {

        config = new Configuration();
```

```
        config.setProperty("hibernate.connector.driver_class",         DB_DRIVER_NAME);
        config.setProperty("hibernate.connection.url",                 DB_URL);
        config.setProperty("hibernate.connection.username",         DB_USERNAME);
        config.setProperty("hibernate.connection.password",         DB_PASSWORD);
        config.setProperty("hibernate.dialect",                     DIALECT);
        config.setProperty("hibernate.hbm2dll.auto",                 HBM2DLL);
        config.setProperty("hibernate.show_sql",                    SHOW_SQL);

        /*
         * Config connection pools
         */

        config.setProperty("connection.provider_class",
"org.hibernate.connection.C3P0ConnectionProvider");
        config.setProperty("hibernate.c3p0.min_size", "5");
        config.setProperty("hibernate.c3p0.max_size", "20");
        config.setProperty("hibernate.c3p0.timeout", "300");
        config.setProperty("hibernate.c3p0.max_statements", "50");
        config.setProperty("hibernate.c3p0.idle_test_period", "3000");


        /**
         * Resource mapping
         */

//        config.addAnnotatedClass(User.class);
//        config.addAnnotatedClass(User.class);
//        config.addAnnotatedClass(User.class);

        sessionFactory = config.buildSessionFactory();
    }


    public HibWrapper openSession() throws HibernateException {
        return new HibWrapper(getOrCreateSession(), CLOSE_AFTER_TRANSACTION);
    }


    public Session getOrCreateSession() throws HibernateException {
        if (session == null) {
            session = sessionFactory.openSession();
        }
        return session;
    }

    public void reconnect() throws HibernateException {
        this.sessionFactory = config.buildSessionFactory();
    }


}
```

Si noti che con l'ultima versione di Hibernate questo approccio non funziona bene (la versione di Hibernate 5.2 consente comunque questa configurazione)

Esempio di ibernazione semplice con XML

Per impostare un semplice progetto di ibernazione utilizzando XML per le configurazioni sono

necessari 3 file, hibernate.cfg.xml, un POJO per ogni entità e un EntityName.hbm.xml per ogni entità. Ecco un esempio di ciascun utilizzo di MySQL:

hibernate.cfg.xml

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

<hibernate-configuration>
   <session-factory>
   <property name="hibernate.dialect">
      org.hibernate.dialect.MySQLDialect
   </property>
   <property name="hibernate.connection.driver_class">
      com.mysql.jdbc.Driver
   </property>

   <property name="hibernate.connection.url">
      jdbc:mysql://localhost/DBSchemaName
   </property>
   <property name="hibernate.connection.username">
      testUserName
   </property>
   <property name="hibernate.connection.password">
      testPassword
   </property>

   <!-- List of XML mapping files -->
   <mapping resource="HibernatePractice/Employee.hbm.xml"/>

</session-factory>
</hibernate-configuration>
```

DBSchemaName, testUserName e testPassword saranno tutti sostituiti. Assicurati di utilizzare il nome completo della risorsa se si trova in un pacchetto.

Employee.java

```
package HibernatePractice;

public class Employee {
    private int id;
    private String firstName;
    private String middleName;
    private String lastName;

    public Employee(){

    }
    public int getId(){
        return id;
    }
    public void setId(int id){
        this.id = id;
    }
    public String getFirstName(){
```

```
        return firstName;
    }
    public void setFirstName(String firstName){
        this.firstName = firstName;
    }
    public String getMiddleName(){
        return middleName;
    }
    public void setMiddleName(String middleName){
        this.middleName = middleName;
    }
    public String getLastName(){
        return lastName;
    }
    public void setLastName(String lastName){
        this.lastName = lastName;
    }
}
```

Employee.hbm.xml

```
<hibernate-mapping>
   <class name="HibernatePractice.Employee" table="employee">
      <meta attribute="class-description">
         This class contains employee information.
      </meta>
      <id name="id" type="int" column="empolyee_id">
         <generator class="native"/>
      </id>
      <property name="firstName" column="first_name" type="string"/>
      <property name="middleName" column="middle_name" type="string"/>
      <property name="lastName" column="last_name" type="string"/>
   </class>
</hibernate-mapping>
```

Di nuovo, se la classe è in un pacchetto usa il nome completo della classe nomepacchetto.className.

Dopo aver questi tre file sei pronto per l'uso di ibernazione nel tuo progetto.

Leggi Iniziare con l'ibernazione online: https://riptutorial.com/it/hibernate/topic/907/iniziare-con-l-ibernazione

Capitolo 2: Abilita / Disabilita il log SQL

Osservazioni

La registrazione di queste query è **lenta** , anche più lenta di quanto lo sia Hibernate. Utilizza anche una quantità enorme di spazio di log. Non utilizzare la registrazione in scenari in cui è richiesta una prestazione. Usalo solo quando collaudi le query che Hibernate genera effettivamente.

Examples

Utilizzando un file di configurazione di registrazione

Nel file di configurazione di registrazione di tua scelta imposta la registrazione dei seguenti pacchetti ai livelli mostrati .:

```
# log the sql statement
org.hibernate.SQL=DEBUG
# log the parameters
org.hibernate.type=TRACE
```

Ci saranno probabilmente dei prefissi specifici per i logger che sono richiesti.

Log4j config:

```
log4j.logger.org.hibernate.SQL=DEBUG
log4j.logger.org.hibernate.type=TRACE
```

Spring Boot `application.properties` :

```
logging.level.org.hibernate.SQL=DEBUG
logging.level.org.hibernate.type=TRACE
```

Logback `logback.xml` :

```
<logger name="org.hibernate.SQL" level="DEBUG"/>
<logger name="org.hibernate.type" level="TRACE"/>
```

Usando le proprietà di ibernazione

Questo mostrerà l'SQL generato, ma non mostrerà i valori contenuti nelle query.

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
    <property name="hibernateProperties">
        <props>
            <!-- show the sql without the parameters -->
```

```
            <prop key="hibernate.show_sql">true</prop>
            <!-- format the sql nice -->
            <prop key="hibernate.format_sql">true</prop>
            <!-- show the hql as comment -->
            <prop key="use_sql_comments">true</prop>
        </props>
    </property>
</bean>
```

Abilita / Disabilita il log di debug di SQL

Alcune applicazioni che utilizzano Hibernate generano un'enorme quantità di SQL all'avvio dell'applicazione. A volte è meglio abilitare / disabilitare il log SQL in punti specifici durante il debug.

Per abilitare, basta eseguire questo codice nel tuo IDE quando esegui il debug dell'applicazione:

```
org.apache.log4j.Logger.getLogger("org.hibernate.SQL")
    .setLevel(org.apache.log4j.Level.DEBUG)
```

Disabilitare:

```
org.apache.log4j.Logger.getLogger("org.hibernate.SQL")
    .setLevel(org.apache.log4j.Level.OFF)
```

Leggi Abilita / Disabilita il log SQL online: https://riptutorial.com/it/hibernate/topic/3548/abilita---disabilita-il-log-sql

Capitolo 3: Associazioni di associazioni tra entità

Examples

OneToMany association

Per illustrare la relazione OneToMany abbiamo bisogno di 2 entità, ad esempio Paese e città. Un Paese ha più città.

Nella CountryEntity di seguito definiamo un insieme di città per Paese.

```
@Entity
@Table(name = "Country")
public class CountryEntity implements Serializable
{
   private static final long serialVersionUID = 1L;

   @Id
   @Column(name = "COUNTRY_ID", unique = true, nullable = false)
   @GeneratedValue(strategy = GenerationType.SEQUENCE)
   private Integer           countryId;

   @Column(name = "COUNTRY_NAME", unique = true, nullable = false, length = 100)
   private String            countryName;

   @OneToMany(mappedBy="country", fetch=FetchType.LAZY)
   private Set<CityEntity> cities = new HashSet<>();

   //Getters and Setters are not shown
}
```

Ora l'entità della città.

```
@Entity
@Table(name = "City")
public class CityEntity implements Serializable
{
   private static final long serialVersionUID = 1L;

   @Id
   @Column(name = "CITY_ID", unique = true, nullable = false)
   @GeneratedValue(strategy = GenerationType.SEQUENCE)
   private Integer           cityId;

   @Column(name = "CITY_NAME", unique = false, nullable = false, length = 100)
   private String            cityName;

   @ManyToOne(optional=false, fetch=FetchType.EAGER)
   @JoinColumn(name="COUNTRY_ID", nullable=false)
   private CountryEntity country;

   //Getters and Setters are not shown
```

```
}
```

Associazione da uno a molti usando XML

Questo è un esempio di come si farebbe una mappatura da uno a molti usando XML. Useremo Autore e Libro come esempio e supponiamo che un autore possa aver scritto molti libri, ma ogni libro avrà un solo autore.

Classe d'autore:

```
public class Author {
    private int id;
    private String firstName;
    private String lastName;

    public Author(){

    }
    public int getId(){
        return id;
    }
    public void setId(int id){
        this.id = id;
    }
    public String getFirstName(){
        return firstName;
    }
    public void setFirstName(String firstName){
        this.firstName = firstName;
    }
    public String getLastName(){
        return lastName;
    }
    public void setLastName(String lastName){
        this.lastName = lastName;
    }
}
```

Classe del libro:

```
public class Book {
    private int id;
    private String isbn;
    private String title;
    private Author author;
    private String publisher;

    public Book() {
        super();
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getIsbn() {
```

```
        return isbn;
    }
    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
    public Author getAuthor() {
        return author;
    }
    public void setAuthor(Author author) {
        this.author = author;
    }
    public String getPublisher() {
        return publisher;
    }
    public void setPublisher(String publisher) {
        this.publisher = publisher;
    }
}
```

Author.hbm.xml:

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

<hibernate-mapping>
   <class name="Author" table="author">
      <meta attribute="class-description">
         This class contains the author's information.
      </meta>
      <id name="id" type="int" column="author_id">
         <generator class="native"/>
      </id>
      <property name="firstName" column="first_name" type="string"/>
      <property name="lastName" column="last_name" type="string"/>
   </class>
</hibernate-mapping>
```

Book.hbm.xml:

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

<hibernate-mapping>
    <class name="Book" table="book_title">
      <meta attribute="class-description">
         This class contains the book information.
      </meta>
        <id name="id" type="int" column="book_id">
            <generator class="native"/>
```

```
        </id>
        <property name="isbn" column="isbn" type="string"/>
        <property name="title" column="title" type="string"/>
         <many-to-one name="author" class="Author" cascade="all">
             <column name="author"></column>
         </many-to-one>
        <property name="publisher" column="publisher" type="string"/>
    </class>
</hibernate-mapping>
```

Ciò che rende la connessione one to many è che la classe Book contiene un Author e che l'xml ha il tag <many-to-one>. L'attributo cascade consente di impostare il modo in cui l'entità figlio verrà salvata / aggiornata.

Leggi Associazioni di associazioni tra entità online:
https://riptutorial.com/it/hibernate/topic/6165/associazioni-di-associazioni-tra-entita

Capitolo 4: Associazioni di mappatura

Examples

Mappatura Hibernate One to One

Ogni Paese ha una Capitale. Ogni capitale ha un Paese.

Country.java

```
package com.entity;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToOne;
import javax.persistence.Table;

@Entity
@Table(name = "countries")
public class Country {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    @Column(name = "name")
    private String name;

    @Column(name = "national_language")
    private String nationalLanguage;

    @OneToOne(mappedBy = "country")
    private Capital capital;

    //Constructor

    //getters and setters

   }
```

Capital.java

```
package com.entity;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.OneToOne;
import javax.persistence.Table;

@Entity
```

```
@Table(name = "capitals")
public class Capital {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private int id;

    private String name;

    private long population;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "country_id")
    private Country country;

    //Constructor

    //getters and setters

}
```

HibernateDemo.java

```
package com.entity;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class HibernateDemo {

public static void main(String ar[]) {
    SessionFactory sessionFactory = new Configuration().configure().buildSessionFactory();
    Session session = sessionFactory.openSession();
    Country india = new Country();
    Capital delhi = new Capital();
    delhi.setName("Delhi");
    delhi.setPopulation(357828394);
    india.setName("India");
    india.setNationalLanguage("Hindi");
    delhi.setCountry(india);
    session.save(delhi);
    session.close();
  }

}
```

Leggi Associazioni di mappatura online: https://riptutorial.com/it/hibernate/topic/6478/associazioni-di-mappatura

Capitolo 5: caching

Examples

Abilitazione della cache di ibernazione in WildFly

Per abilitare la cache di secondo livello per Hibernate in WildFly, aggiungi questa proprietà al tuo file `persistence.xml` :

```
<property name="hibernate.cache.use_second_level_cache" value="true"/>
```

È inoltre possibile abilitare la cache delle query con questa proprietà:

```
<property name="hibernate.cache.use_query_cache" value="true"/>
```

WildFly non richiede di definire un provider di cache quando si abilita la cache di secondo livello di Hibernate, poiché per impostazione predefinita viene utilizzato Infinispan. Se si desidera utilizzare un fornitore di cache alternativo, tuttavia, è possibile farlo con la proprietà `hibernate.cache.provider_class` .

Leggi caching online: https://riptutorial.com/it/hibernate/topic/3462/caching

Capitolo 6: Caricamento pigro vs caricamento Eager

Examples

Caricamento pigro vs caricamento Eager

Il recupero o il caricamento dei dati può essere principalmente classificato in due tipi: desideroso e pigro.

Per utilizzare Hibernate, assicurati di aver aggiunto l'ultima versione di esso alla sezione delle dipendenze del tuo file pom.xml:

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.2.1.Final</version>
</dependency>
```

1. Caricamento Eager E Caricamento Lazy

La prima cosa di cui dovremmo discutere qui è quale carico pigro e caricamento avido sono:

Eager Loading è un modello di progettazione in cui l'inizializzazione dei dati avviene sul posto. Significa che le raccolte vengono recuperate completamente nel momento in cui viene prelevato il loro genitore (recupera immediatamente)

Lazy Loading è un modello di progettazione che viene utilizzato per rinviare l'inizializzazione di un oggetto fino al punto in cui è necessario. Questo può effettivamente contribuire alle prestazioni dell'applicazione.

2. Utilizzo dei diversi tipi di caricamento

Il caricamento lento può essere abilitato utilizzando il seguente parametro XML:

```
lazy="true"
```

Analizziamo l'esempio. Per prima cosa abbiamo una classe User:

```
public class User implements Serializable {

    private Long userId;
    private String userName;
    private String firstName;
    private String lastName;
    private Set<OrderDetail> orderDetail = new HashSet<>();

    //setters and getters
```

```
    //equals and hashcode
    }
```

Guarda il Set di orderDetail che abbiamo. Ora diamo un'occhiata alla **classe OrderDetail** :

```
public class OrderDetail implements Serializable {

    private Long orderId;
    private Date orderDate;
    private String orderDesc;
    private User user;

    //setters and getters
    //equals and hashcode
}
```

La parte importante che è coinvolta nell'impostazione del caricamento lazy in `UserLazy.hbm.xml` :

```
<set name="orderDetail" table="USER_ORDER" inverse="true" lazy="true" fetch="select">
    <key>
        <column name="USER_ID" not-null="true" />
    </key>
   <one-to-many class="com.baeldung.hibernate.fetching.model.OrderDetail" />
</set>
```

Questo è il modo in cui il caricamento lazy è abilitato. Per disabilitare il caricamento lazy possiamo semplicemente usare: `lazy = "false"` e questo a sua volta abiliterà il caricamento ansioso. Di seguito è riportato l'esempio dell'impostazione di caricamento ingerente in un altro file User.hbm.xml:

```
<set name="orderDetail" table="USER_ORDER" inverse="true" lazy="false" fetch="select">
    <key>
        <column name="USER_ID" not-null="true" />
    </key>
   <one-to-many class="com.baeldung.hibernate.fetching.model.OrderDetail" />
</set>
```

Scopo

Per coloro che non hanno giocato con questi due progetti, lo scopo di pigro e desideroso è all'interno di una **sessione** specifica di *SessionFactory* . *Eager* carica tutto all'istante, significa che non è necessario chiamare nulla per recuperarlo. Ma il recupero pigro di solito richiede qualche azione per recuperare raccolta / oggetto mappato. Questo a volte è problematico ottenere il recupero pigro al di fuori della *sessione* . Ad esempio, hai una vista che mostra i dettagli di alcuni POJO mappati.

```
@Entity
public class User {
    private int userId;
    private String username;
    @OneToMany
    private Set<Page> likedPage;
```

```
    // getters and setters here
}

@Entity
public class Page{
    private int pageId;
    private String pageURL;

    // getters and setters here
}

public class LazzyTest{
    public static void main(String...s){
        SessionFactory sessionFactory = new SessionFactory();
        Session session = sessionFactory.openSession();
        Transaction transaction = session.beginTransaction();

        User user = session.get(User.class, 1);
        transaction.commit();
        session.close();

        // here comes the lazy fetch issue
        user.getLikedPage();
    }
}
```

Quando *proverai* a farti **prendere pigro** fuori dalla *sessione* , otterrai *lazyinitializeException* . Questo perché per impostazione predefinita la strategia di recupero per tutti oneToMany o qualsiasi altra relazione è *lazy* (chiamata su DB su richiesta) e quando si è chiusa la sessione, non si ha il potere di comunicare con il database. quindi il nostro codice cerca di recuperare la raccolta di *LikePage* e genera un'eccezione perché non esiste una sessione associata per il rendering del DB.

La soluzione per questo è usare:

1. Apri Sessione in vista - In cui si mantiene aperta la sessione anche nella vista di rendering.
2. `Hibernate.initialize(user.getLikedPage())` prima della sessione di chiusura - Questo indica a `Hibernate.initialize(user.getLikedPage())` di inizializzare gli elementi della raccolta

Leggi Caricamento pigro vs caricamento Eager online:
https://riptutorial.com/it/hibernate/topic/7249/caricamento-pigro-vs-caricamento-eager

Capitolo 7: Criteri e proiezioni

Examples

Elenco utilizzando restrizioni

Supponendo di avere una tabella TravelReview con i nomi di città come colonna "titolo"

```
 Criteria criteria =
    session.createCriteria(TravelReview.class);
  List review =
    criteria.add(Restrictions.eq("title", "Mumbai")).list();
  System.out.println("Using equals: " + review);
```

Possiamo aggiungere restrizioni ai criteri concatenandoli come segue:

```
List reviews = session.createCriteria(TravelReview.class)
   .add(Restrictions.eq("author", "John Jones"))
   .add(Restrictions.between("date",fromDate,toDate))
   .add(Restrictions.ne("title","New York")).list();
```

Utilizzando le proiezioni

Se vogliamo recuperare solo poche colonne, possiamo usare la classe Proiezioni per farlo. Ad esempio, il codice seguente recupera la colonna del titolo

```
 // Selecting all title columns
  List review = session.createCriteria(TravelReview.class)
        .setProjection(Projections.property("title"))
    .list();
  // Getting row count
  review = session.createCriteria(TravelReview.class)
    .setProjection(Projections.rowCount())
    .list();
  // Fetching number of titles
  review = session.createCriteria(TravelReview.class)
    .setProjection(Projections.count("title"))
    .list();
```

Usa filtri

`@Filter` è usato come campo `WHERE` , qui alcuni esempi

Entità studentesca

```
@Entity
@Table(name = "Student")
public class Student
{
    /*...*/
```

```
    @OneToMany
    @Filter(name = "active", condition = "EXISTS(SELECT * FROM Study s WHERE state = true and
s.id = study_id)")
    Set<StudentStudy> studies;

    /* getters and setters methods */
}
```

Entità di studio

```
@Entity
@Table(name = "Study")
@FilterDef(name = "active")
@Filter(name = "active", condition="state = true")
public class Study
{
    /*...*/

    @OneToMany
    Set<StudentStudy> students;

    @Field
    boolean state;

    /* getters and setters methods */
}
```

StudentStudy Entity

```
@Entity
@Table(name = "StudentStudy")
@Filter(name = "active", condition = "EXISTS(SELECT * FROM Study s WHERE state = true and s.id
= study_id)")
public class StudentStudy
{
    /*...*/

    @ManytoOne
    Student student;

    @ManytoOne
    Study study;

    /* getters and setters methods */
}
```

In questo modo, ogni volta che il filtro "attivo" è abilitato,

-Ogni interrogazione che facciamo sull'entità studentesca restituirà **TUTTI gli** Studenti con **SOLO il** loro `state = true` studi `state = true`

-Ogni interrogazione che facciamo sull'entità dello studio restituirà **TUTTI** `state = true` studi sullo `state = true`

-Ogni interrogazione che facciamo sul StudentStudy entiy restituirà **SOLO** quelli con uno `state =`

`true` rapporto di studio

Si noti che study_id è il nome del campo sulla tabella sql StudentStudy

Leggi Criteri e proiezioni online: https://riptutorial.com/it/hibernate/topic/3939/criteri-e-proiezioni

Capitolo 8: Hibernate Entity Relationships usando le annotazioni

Parametri

Annotazione	Dettagli
`@OneToOne`	Specifica una relazione uno a uno con un oggetto corrispondente.
`@OneToMany`	Specifica un singolo oggetto che si associa a molti oggetti.
`@ManyToOne`	Specifica una raccolta di oggetti che si associano a un singolo oggetto.
`@Entity`	Specifica un oggetto che si associa a una tabella di database.
`@Table`	Specifica quale tabella di database viene mappata anche da questo oggetto.
`@JoinColumn`	Specifica in quale colonna è memorizzata una chiave di foregin.
`@JoinTable`	Specifica una tabella intermedia che memorizza le chiavi esterne.

Examples

Multi-direzionale Molti a molti utilizzando l'oggetto tabella di join gestito dall'utente

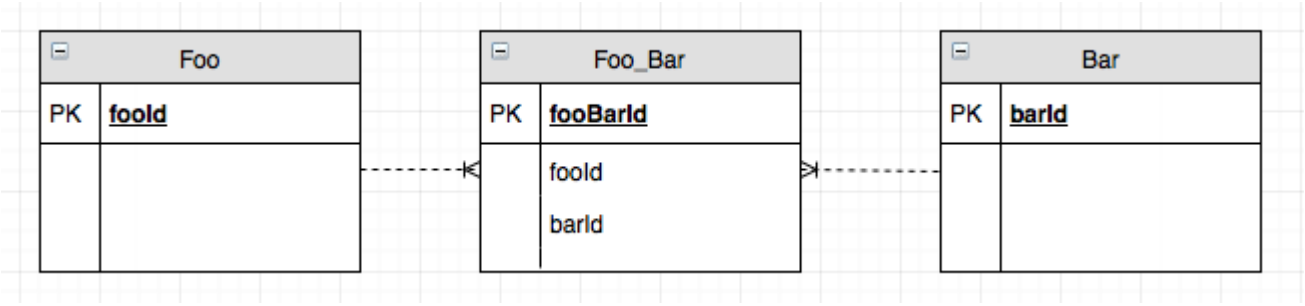
```
@Entity
@Table(name="FOO")
public class Foo {
    private UUID fooId;

    @OneToMany(mappedBy = "bar")
    private List<FooBar> bars;
}

@Entity
@Table(name="BAR")
public class Bar {
    private UUID barId;

    @OneToMany(mappedBy = "foo")
    private List<FooBar> foos;
}

@Entity
@Table(name="FOO_BAR")
public class FooBar {
    private UUID fooBarId;
```

```
    @ManyToOne
    @JoinColumn(name = "fooId")
    private Foo foo;

    @ManyToOne
    @JoinColumn(name = "barId")
    private Bar bar;

    //You can store other objects/fields on this table here.
}
```

Specifica una relazione a due vie tra molti oggetti `Foo` su molti oggetti `Bar` usando una tabella di join intermedia che l'utente gestisce.

Gli oggetti `Foo` sono memorizzati come righe in una tabella chiamata `FOO` . Gli oggetti `Bar` vengono memorizzati come righe in una tabella chiamata `BAR` . Le relazioni tra gli oggetti `Foo` e `Bar` sono memorizzate in una tabella chiamata `FOO_BAR` . C'è un oggetto `FooBar` come parte dell'applicazione.

Comunemente utilizzato quando si desidera memorizzare informazioni aggiuntive sull'oggetto join, ad esempio la data di creazione della relazione.

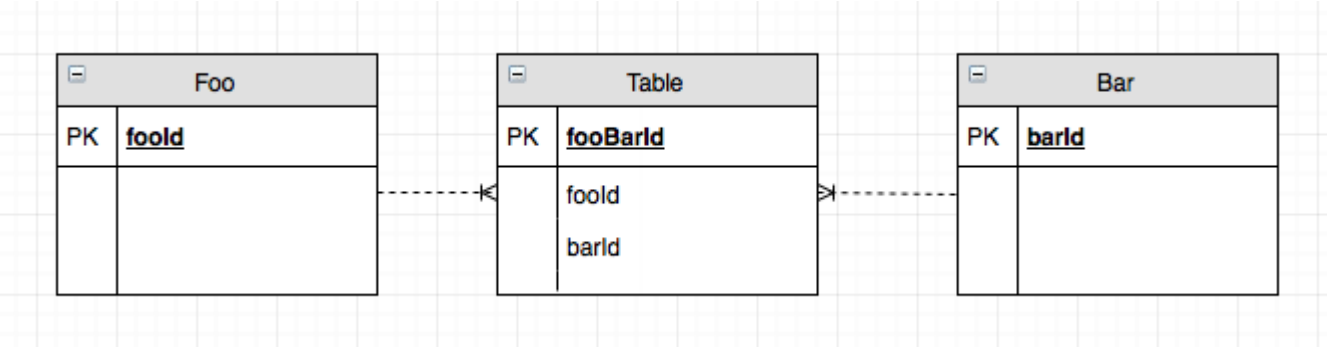


Multi-direzionale Molti a molti utilizzando la tabella di join gestita da Hibernate

```
@Entity
@Table(name="FOO")
public class Foo {
    private UUID fooId;

    @OneToMany
    @JoinTable(name="FOO_BAR",
        joinColumns = @JoinColumn(name="fooId"),
        inverseJoinColumns = @JoinColumn(name="barId"))
    private List<Bar> bars;
}

@Entity
@Table(name="BAR")
public class Bar {
    private UUID barId;

    @OneToMany
    @JoinTable(name="FOO_BAR",
        joinColumns = @JoinColumn(name="barId"),
        inverseJoinColumns = @JoinColumn(name="fooId"))
    private List<Foo> foos;
}
```

Specifica una relazione tra molti oggetti `Foo` su molti oggetti `Bar` utilizzando una tabella di join intermedia gestita da Hibernate.

Gli oggetti `Foo` sono memorizzati come righe in una tabella chiamata `FOO` . Gli oggetti `Bar` vengono memorizzati come righe in una tabella chiamata `BAR` . Le relazioni tra gli oggetti `Foo` e `Bar` sono memorizzate in una tabella chiamata `FOO_BAR` . Tuttavia ciò implica che non vi è alcun oggetto `FooBar` come parte dell'applicazione.

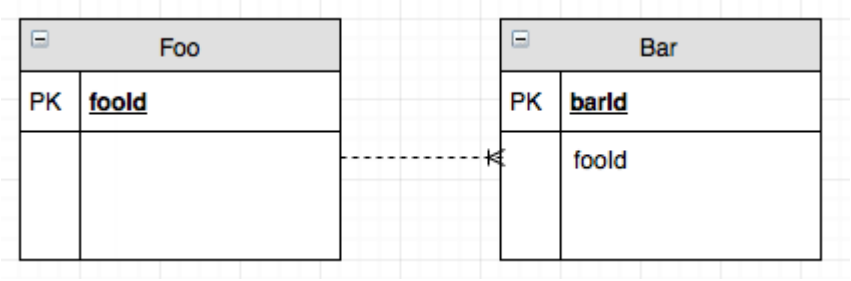


Relazione bi-direzionale da uno a molti utilizzando la mappatura delle chiavi esterne

```
@Entity
@Table(name="FOO")
public class Foo {
    private UUID fooId;

    @OneToMany(mappedBy = "bar")
    private List<Bar> bars;
}

@Entity
@Table(name="BAR")
public class Bar {
    private UUID barId;

    @ManyToOne
    @JoinColumn(name = "fooId")
    private Foo foo;
}
```

Specifica una relazione bidirezionale tra un oggetto `Foo` e molti oggetti `Bar` utilizzando una chiave esterna.

Gli oggetti `Foo` sono memorizzati come righe in una tabella chiamata `FOO` . Gli oggetti `Bar` vengono memorizzati come righe in una tabella chiamata `BAR` . La chiave esterna è memorizzata nella tabella `BAR` in una colonna chiamata `fooId` .



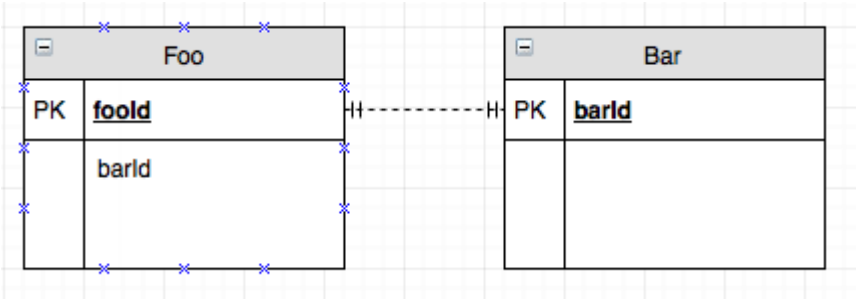
Relazione bi-direzionale uno-a-uno gestita da Foo.class

```
@Entity
@Table(name="FOO")
public class Foo {
    private UUID fooId;

    @OneToOne(cascade = CascadeType.ALL)
    @JoinColumn(name = "barId")
    private Bar bar;
}

@Entity
@Table(name="BAR")
public class Bar {
    private UUID barId;

    @OneToOne(mappedBy = "bar")
    private Foo foo;
}
```

Specifica una relazione bidirezionale tra un oggetto `Foo` e un oggetto `Bar` utilizzando una chiave esterna.

Gli oggetti `Foo` sono memorizzati come righe in una tabella chiamata `FOO` . Gli oggetti `Bar` vengono memorizzati come righe in una tabella chiamata `BAR` . La chiave esterna è memorizzata nella tabella `FOO` in una colonna denominata `barId` .

Si noti che il valore `mappedBy` è il nome del campo sull'oggetto, non il nome della colonna.



Relazione unidirezionale unidirezionale con la tabella di join gestita dall'utente

```
@Entity
@Table(name="FOO")
public class Foo {
    private UUID fooId;
```

```
    @OneToMany
    @JoinTable(name="FOO_BAR",
        joinColumns = @JoinColumn(name="fooId"),
        inverseJoinColumns = @JoinColumn(name="barId", unique=true))
    private List<Bar> bars;
}

@Entity
@Table(name="BAR")
public class Bar {
    private UUID barId;

    //No Mapping specified here.
}

@Entity
@Table(name="FOO_BAR")
public class FooBar {
    private UUID fooBarId;

    @ManyToOne
    @JoinColumn(name = "fooId")
    private Foo foo;

    @ManyToOne
    @JoinColumn(name = "barId", unique = true)
    private Bar bar;

    //You can store other objects/fields on this table here.
}
```

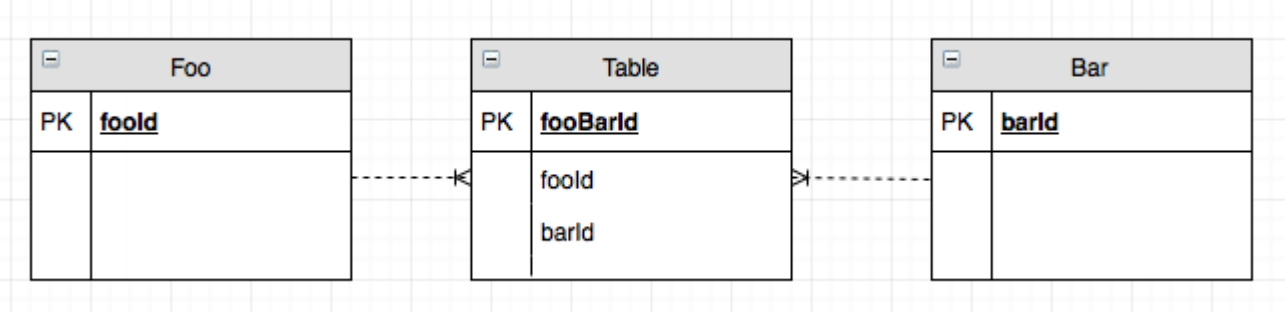
Specifica una relazione unidirezionale tra un oggetto `Foo` e molti oggetti `Bar` utilizzando una tabella di join intermedia che l'utente gestisce.

Questo è simile a una relazione `ManyToMany` , ma se si aggiunge un vincolo `unique` alla chiave esterna di destinazione è possibile far rispettare che è `OneToMany` .

Gli oggetti `Foo` sono memorizzati come righe in una tabella chiamata `FOO` . Gli oggetti `Bar` vengono memorizzati come righe in una tabella chiamata `BAR` . Le relazioni tra gli oggetti `Foo` e `Bar` sono memorizzate in una tabella chiamata `FOO_BAR` . C'è un oggetto `FooBar` come parte dell'applicazione.

Si noti che non vi è alcuna mappatura degli oggetti `Bar` indietro agli oggetti `Foo` . `Bar` oggetti della `Bar` possono essere manipolati liberamente senza influenzare gli oggetti `Foo` .

Molto comunemente usato con Spring Security quando si imposta un oggetto `User` che ha un elenco di `Role` che possono eseguire. È possibile aggiungere e rimuovere i ruoli ad un utente, senza doversi preoccupare di cascate eliminazione `Role` s'.



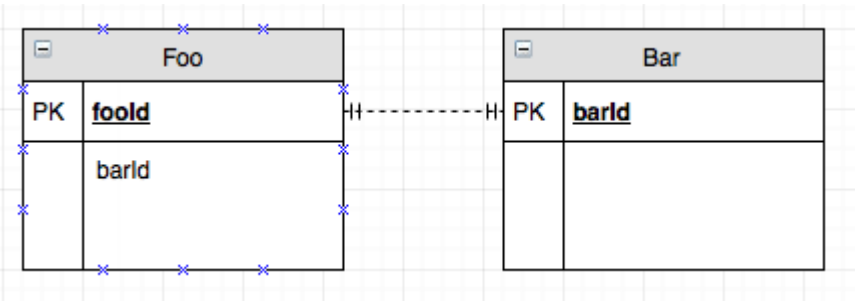
Relazione One-One unidirezionale

```
@Entity
@Table(name="FOO")
public class Foo {
    private UUID fooId;

    @OneToOne
    private Bar bar;
}

@Entity
@Table(name="BAR")
public class Bar {
    private UUID barId;
    //No corresponding mapping to Foo.class
}
```

Specifica una relazione unidirezionale tra un oggetto `Foo` e un oggetto `Bar` .

Gli oggetti `Foo` sono memorizzati come righe in una tabella chiamata `FOO` . Gli oggetti `Bar` vengono memorizzati come righe in una tabella chiamata `BAR` .

Si noti che non vi è alcuna mappatura degli oggetti `Bar` indietro agli oggetti `Foo` . `Bar` oggetti della `Bar` possono essere manipolati liberamente senza influenzare gli oggetti `Foo` .



Leggi Hibernate Entity Relationships usando le annotazioni online:
https://riptutorial.com/it/hibernate/topic/5742/hibernate-entity-relationships-usando-le-annotazioni

Capitolo 9: HQL

introduzione

HQL è Hibernate Query Language, basato su SQL e dietro le quinte è stato modificato in SQL ma la sintassi è diversa. Usi nomi di entità / classe non nomi di tabelle e nomi di campi non nomi di colonne. Permette anche molte stenografie.

Osservazioni

La cosa principale da ricordare quando si usa hql è usare il nome della classe e i nomi dei campi al posto dei nomi delle tabelle e delle colonne a cui siamo abituati in SQL.

Examples

Selezione di un'intera tabella

```
hql = "From EntityName";
```

Seleziona colonne specifiche

```
hql = "Select id, name From Employee";
```

Includere una clausola Where

```
hql = "From Employee where id = 22";
```

Aderire

```
hql = "From Author a, Book b Where a.id = book.author";
```

Leggi HQL online: https://riptutorial.com/it/hibernate/topic/9388/hql

Capitolo 10: Ibernazione e JPA

Examples

Relazione tra Hibernate e JPA

Hibernate è un'implementazione dello standard JPA . Come tale, tutto ha detto che anche per Hibernate è vero.

Hibernate ha alcune estensioni per JPA. Inoltre, il modo di configurare un provider JPA è specifico del provider. Questa sezione della documentazione dovrebbe contenere solo ciò che è specifico di Hibernate.

Leggi Ibernazione e JPA online: https://riptutorial.com/it/hibernate/topic/6313/ibernazione-e-jpa

Capitolo 11: Ottimizzazione delle prestazioni

Examples

Non utilizzare il tipo di recupero EAGER

Hibernate può usare due tipi di fetch quando si sta mappando la relazione tra due entità: `EAGER` e `LAZY` .

In generale, il tipo di recupero `EAGER` non è una buona idea, perché dice a JPA di recuperare *sempre* i dati, anche quando questi dati non sono necessari.

Ad esempio, se si dispone di un'entità `Person` e della relazione con `Address` come questa:

```
@Entity
public class Person {

  @OneToMany(mappedBy="address", fetch=FetchType.EAGER)
  private List<Address> addresses;

}
```

Ogni volta che interroghi una `Person` , verrà restituito anche l'elenco di `Address` di questa `Person` .

Quindi, invece di mappare la tua entità con:

```
@ManyToMany(mappedBy="address", fetch=FetchType.EAGER)
```

Uso:

```
@ManyToMany(mappedBy="address", fetch=FetchType.LAZY)
```

Un'altra cosa a cui prestare attenzione è la relazione `@OneToOne` e `@ManyToOne` . Entrambi sono EAGER *di default* . Quindi, se sei preoccupato delle prestazioni della tua applicazione, devi impostare il recupero per questo tipo di relazione:

```
@ManyToOne(fetch=FetchType.LAZY)
```

E:

```
@OneToOne(fetch=FetchType.LAZY)
```

Usa la composizione invece dell'ereditarietà

Hibernate ha alcune strategie di ereditarietà. Il tipo di ereditarietà `JOINED` esegue un JOIN tra l'entità child e l'entità parent.

Il problema con questo approccio è che Hibernate porta **sempre** i dati di tutte le tabelle coinvolte nell'eredità.

Ad esempio, se le entità `Bicycle` e `MountainBike` utilizzano il tipo di ereditarietà `JOINED` :

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

}
```

E:

```
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class MountainBike extends Bicycle {

}
```

Qualsiasi query JPQL che colpisca `MountainBike` porterà i dati della `Bicycle` , creando una query SQL come:

```
select mb.*, b.* from MountainBike mb JOIN Bicycle b ON b.id = mb.id WHERE ...
```

Se hai un altro genitore per `Bicycle` (come `Transport` , per esempio), questa query sopra porta i dati anche da questo genitore, facendo un JOIN extra.

Come puoi vedere, anche questa è una sorta di mappatura `EAGER` . Non hai la possibilità di portare solo i dati della tabella `MountainBike` usando questa strategia di ereditarietà.

Il meglio per le prestazioni è utilizzare la composizione anziché l'ereditarietà.

Per fare ciò, puoi mappare l'entità `MountainBike` per avere una `bicycle` campo:

```
@Entity
public class MountainBike {

    @OneToOne(fetchType = FetchType.LAZY)
    private Bicycle bicycle;

}
```

E `Bicycle` :

```
@Entity
public class Bicycle {

}
```

Ogni query ora porterà solo i dati `MountainBike` per impostazione predefinita.

Leggi Ottimizzazione delle prestazioni online:

https://riptutorial.com/it/hibernate/topic/2326/ottimizzazione-delle-prestazioni

Capitolo 12: Query SQL native

Examples

Query semplice

Supponendo che tu abbia un handle sull'oggetto Hibernate `Session` , in questo caso named `session` :

```
List<Object[]> result = session.createNativeQuery("SELECT * FROM some_table").list();
for (Object[] row : result) {
    for (Object col : row) {
        System.out.print(col);
    }
}
```

Ciò recupererà tutte le righe in `some_table` e le `some_table` nella variabile `result` e stamperà ogni valore.

Esempio per ottenere un risultato unico

```
Object pollAnswered = getCurrentSession().createSQLQuery(
        "select * from TJ_ANSWERED_ASW where pol_id = "+pollId+" and prf_log =
'"+logid+"'").uniqueResult();
```

con questa query, ottieni un risultato unico quando sai che il risultato della query sarà sempre univoco.

E se la query restituisce più di un valore, otterrai un'eccezione

    org.hibernate.NonUniqueResultException

Controlla anche i dettagli in questo link qui con più descrizioni

Quindi, assicurati di sapere che la query restituirà risultati unici

Leggi Query SQL native online: https://riptutorial.com/it/hibernate/topic/6978/query-sql-native

Capitolo 13: Recupero in ibernazione

introduzione

Il recupero è molto importante in JPA (Java Persistence API). In JPA, HQL (Hibernate Query Language) e JPQL (Java Persistence Query Language) vengono utilizzati per recuperare le entità in base alle loro relazioni. Anche se è molto meglio che utilizzare così tante query e sottocomponenti per ottenere ciò che vogliamo usando l'SQL nativo, la strategia con cui recuperiamo le entità associate in JPA sta comunque essenzialmente influenzando le prestazioni della nostra applicazione.

Examples

Si consiglia di utilizzare FetchType.LAZY. Unisciti a recuperare le colonne quando sono necessarie.

Di seguito è riportata una classe di entità Employer che viene associata al datore di lavoro della tabella. Come puoi vedere ho usato **fetch = FetchType.LAZY** invece di fetch = FetchType.EAGER. La ragione per cui sto usando LAZY è perché il datore di lavoro può avere molte proprietà in seguito e ogni volta che potrei non aver bisogno di conoscere tutti i campi di un datore di lavoro, così caricandoli tutti porterà a una cattiva prestazione allora un datore di lavoro è caricato.

```
@Entity
@Table(name = "employer")
    public class Employer
    {
        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        private Long id;

        @Column(name = "name")
        private String Name;

        @OneToMany(mappedBy = "employer", fetch = FetchType.LAZY,
               cascade = { CascadeType.ALL }, orphanRemoval = true)
        private List<Employee> employees;

        public Long getId() {
            return id;
        }

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

        public String getName() {
            return name;
        }

        public void setName(String name) {
```

```
            this.name = name;
        }

        public List<Employee> getEmployees() {
            return employees;
        }

        public void setEmployees(List<Employee> employees) {
            this.employees = employees;
        }
    }
```

Tuttavia, per le associazioni recuperate LAZY, i proxy non inizializzati a volte portano a LazyInitializationException. In questo caso, possiamo semplicemente utilizzare JOIN FETCH in HQL / JPQL per evitare LazyInitializationException.

```
SELECT Employer employer FROM Employer
       LEFT JOIN FETCH employer.name
       LEFT JOIN FETCH employer.employee employee
       LEFT JOIN FETCH employee.name
       LEFT JOIN FETCH employer.address
```

Leggi Recupero in ibernazione online: https://riptutorial.com/it/hibernate/topic/9475/recupero-in-ibernazione

Capitolo 14: Strategia di denominazione personalizzata

Examples

Creazione e utilizzo di una piattaforma ImplicitNaming personalizzata

La creazione di una `ImplicitNamingStrategy` personalizzata consente di modificare il modo in cui Hibernate assegnerà i nomi agli attributi di `Entity` non esplicitamente denominati, incluse chiavi esterne, chiavi univoche, colonne identificatore, colonne di base e altro.

Ad esempio, per impostazione predefinita, Hibernate genererà chiavi esterne che sono hash e assomigliano a:

```
FKe6hidh4u0qh8y1ijy59s2ee6m
```

Anche se questo spesso non è un problema, potresti desiderare che il nome fosse più descrittivo, come ad esempio:

```
FK_asset_tenant
```

Questo può essere fatto facilmente con una `ImplicitNamingStrategy` personalizzata.

Questo esempio estende `ImplicitNamingStrategyJpaCompliantImpl` , tuttavia, se lo desideri, puoi scegliere di implementare `ImplicitNamingStrategy` .

```
import org.hibernate.boot.model.naming.Identifier;
import org.hibernate.boot.model.naming.ImplicitForeignKeyNameSource;
import org.hibernate.boot.model.naming.ImplicitNamingStrategyJpaCompliantImpl;

public class CustomNamingStrategy extends ImplicitNamingStrategyJpaCompliantImpl {

    @Override
    public Identifier determineForeignKeyName(ImplicitForeignKeyNameSource source) {
        return toIdentifier("FK_" + source.getTableName().getCanonicalName() + "_" +
source.getReferencedTableName().getCanonicalName(), source.getBuildingContext());
    }

}
```

Per indicare a Hibernate quale `ImplicitNamingStrategy` utilizzare, definire la proprietà `hibernate.implicit_naming_strategy` nel file `persistence.xml` o `hibernate.cfg.xml` come di seguito:

```
<property name="hibernate.implicit_naming_strategy"
                  value="com.example.foo.bar.CustomNamingStrategy"/>
```

Oppure puoi specificare la proprietà nel file `hibernate.properties` come di seguito:

```
hibernate.implicit_naming_strategy=com.example.foo.bar.CustomNamingStrategy
```

In questo esempio, tutte le chiavi `CustomNamingStrategy` che non hanno un `name` definito esplicitamente riceveranno il loro nome da `CustomNamingStrategy` .

Strategia di denominazione fisica personalizzata

Quando mappiamo le nostre entità ai nomi delle tabelle del database, ci basiamo su un'annotazione `@Table` . Ma se abbiamo una convenzione di denominazione per i nostri nomi di tabelle di database, possiamo implementare una strategia di denominazione fisica personalizzata per dire in ibernato di calcolare i nomi delle tabelle in base ai nomi delle entità, senza specificare esplicitamente tali nomi con `@Table` annotazione `@Table` . Lo stesso vale per gli attributi e la mappatura delle colonne.

Ad esempio, il nome della nostra entità è:

```
ApplicationEventLog
```

E il nostro nome della tabella è:

```
application_event_log
```

La nostra strategia di denominazione fisica ha bisogno di convertire da nomi di entità che sono case cammello ai nostri nomi di tabelle db che sono caso di serpente. Possiamo raggiungere questo obiettivo estendendo il `PhysicalNamingStrategyStandardImpl` di hibernate:

```
import org.hibernate.boot.model.naming.Identifier;
import org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl;
import org.hibernate.engine.jdbc.env.spi.JdbcEnvironment;

public class PhysicalNamingStrategyImpl extends PhysicalNamingStrategyStandardImpl {

    private static final long serialVersionUID = 1L;
    public static final PhysicalNamingStrategyImpl INSTANCE = new
PhysicalNamingStrategyImpl();

    @Override
    public Identifier toPhysicalTableName(Identifier name, JdbcEnvironment context) {
        return new Identifier(addUnderscores(name.getText()), name.isQuoted());
    }

    @Override
    public Identifier toPhysicalColumnName(Identifier name, JdbcEnvironment context) {
        return new Identifier(addUnderscores(name.getText()), name.isQuoted());
    }

    protected static String addUnderscores(String name) {
        final StringBuilder buf = new StringBuilder(name);
        for (int i = 1; i < buf.length() - 1; i++) {
            if (Character.isLowerCase(buf.charAt(i - 1)) &&
                    Character.isUpperCase(buf.charAt(i)) &&
                    Character.isLowerCase(buf.charAt(i + 1))) {
                buf.insert(i++, '_');
```

```
            }
        }
        return buf.toString().toLowerCase(Locale.ROOT);
    }
}
```

Stiamo sovrascrivendo il comportamento predefinito dei metodi `toPhysicalTableName` e `toPhysicalColumnName` per applicare la nostra convenzione di denominazione `toPhysicalColumnName` .

Per utilizzare la nostra implementazione personalizzata dobbiamo definire la proprietà `hibernate.physical_naming_strategy` e dargli il nome della nostra classe `PhysicalNamingStrategyImpl` .

```
hibernate.physical_naming_strategy=com.example.foo.bar.PhysicalNamingStrategyImpl
```

In questo modo possiamo alleviare il nostro codice dalle annotazioni `@Table` e `@Column` , quindi la nostra classe di entità:

```
@Entity
public class ApplicationEventLog {
    private Date startTimestamp;
    private String logUser;
    private Integer eventSuccess;

    @Column(name="finish_dtl")
    private String finishDetails;
}
```

sarà correttamente mappato alla tabella db:

```
CREATE TABLE application_event_log (
  ...
  start_timestamp timestamp,
  log_user varchar(255),
  event_success int(11),
  finish_dtl varchar(2000),
  ...
)
```

Come visto nell'esempio sopra, possiamo ancora dichiarare esplicitamente il nome dell'oggetto db se non lo è, per qualche ragione, secondo la nostra convenzione di denominazione generale: `@Column(name="finish_dtl")`

Leggi Strategia di denominazione personalizzata online: https://riptutorial.com/it/hibernate/topic/3051/strategia-di-denominazione-personalizzata

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con l'ibernazione	Community, JamesENL, Michael Piefel, Naresh Kumar, Reborn, user7491506
2	Abilita / Disabilita il log SQL	Daniel Käfer, Dherik, JamesENL, Michael Piefel
3	Associazioni di associazioni tra entità	StanislavL, user7491506
4	Associazioni di mappatura	Dherik, omkar sirra
5	caching	Mitch Talmadge
6	Caricamento pigro vs caricamento Eager	BELLIL, Pramod, Pritam Banerjee, vicky
7	Criteri e proiezioni	Saifer, Sameer Srivastava
8	Hibernate Entity Relationships usando le annotazioni	Aleksei Loginov, JamesENL
9	HQL	Daniel Käfer, user7491506
10	Ibernazione e JPA	Michael Piefel
11	Ottimizzazione delle prestazioni	Dherik, Michael Piefel
12	Query SQL native	Daniel Käfer, Nathaniel Ford, Sandeep Kamath
13	Recupero in ibernazione	rObOtAndChalie
14	Strategia di denominazione personalizzata	Mitch Talmadge, Naresh Kumar, veljkost