Бесплатная электронная книга

учусь hibernate

Free unaffiliated eBook created from
Stack Overflow contributors.

#hibernate

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: hibernate

It is an unofficial and free hibernate ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official hibernate.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с спящим режимом

замечания

Компонент `SessionFactory` отвечает за создание, поддержание, закрытие и очистку всех сеансов базы данных, которые `TransactionManager` запрашивает для создания. Вот почему мы autowire `SessionFactory` в DAO и запускаем все запросы через него.

Один из самых больших вопросов, который задают новые пользователи Hibernate, - «Когда мои изменения совершаются?» и ответ имеет смысл, когда вы думаете, как `TransactionManager` работает с `SesisonFactory` . Изменения в базе данных будут `@Transactional` при выходе из метода службы, который был аннотирован с помощью `@Transactional` . Причиной этого является то, что транзакция должна представлять собой единую «единицу» непрерывной работы. Если что-то пойдет не так с устройством, тогда предполагается, что устройство потерпело неудачу, и все изменения должны быть отменены. Таким образом `SessionFactory` будет очищать и очищать сеанс при выходе из метода службы, который вы назвали первоначально.

Это не означает, что он не будет очищать и очищать сессию во время вашей транзакции. Например, если я вызову метод службы, чтобы добавить коллекцию из 5 объектов и вернуть общее количество объектов в базе данных, `SessionFactory` поймет, что запрос (`SELECT COUNT(*)`) требует, чтобы обновленное состояние было точным, и так что сбросьте добавление 5 объектов перед запуском запроса на подсчет. Исполнение может выглядеть примерно так:

Версии

Версия	Ссылка на документацию	Дата выхода
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

Использование конфигурации XML для настройки спящего режима

Я создаю файл с именем `database-servlet.xml` где `database-servlet.xml` то в пути к классам.

Первоначально ваш файл конфигурации будет выглядеть так:

```
<?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>
```

Вы заметите, что я импортировал пространства имен `tx` и `jdbc` Spring. Это связано с тем, что мы будем использовать их довольно сильно в этом файле конфигурации.

Первое, что вы хотите сделать, это включить управление транзакциями на основе аннотаций (`@Transactional`). Основная причина, по которой люди используют Hibernate весной, - это то, что Spring будет управлять всеми вашими транзакциями для вас. Добавьте в конфигурационный файл следующую строку:

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

Нам нужно создать источник данных. Источником данных является база данных, которую Hibernate собирается использовать для сохранения ваших объектов. Обычно один менеджер транзакций будет иметь один источник данных. Если вы хотите, чтобы Hibernate разговаривал с несколькими источниками данных, у вас есть несколько менеджеров транзакций.

```
<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>
```

Класс этого компонента может быть любым, что реализует `javax.sql.DataSource` чтобы вы могли написать свой собственный. Этот примерный класс предоставляется Spring, но не имеет собственного пула потоков. Популярной альтернативой является Apache Commons `org.apache.commons.dbcp.BasicDataSource` , но есть и многие другие. Я объясню каждое из свойств ниже:

- *driverClassName* : путь к вашему драйверу JDBC. Это JAR, **специфичный** для **базы данных,** который должен быть доступен в вашем пути к классам. Убедитесь, что у вас самая последняя версия. Если вы используете базу данных Oracle, вам понадобится OracleDriver. Если у вас есть база данных MySQL, вам понадобится MySQLDriver. Смотрите , если вы можете найти нужный драйвер здесь , но быстрый Google должен

дать вам правильный драйвер.

- *url* : URL-адрес вашей базы данных. Обычно это будет нечто вроде `jdbc\:oracle\:thin\:\path\to\your\database` или `jdbc:mysql://path/to/your/database` . Если вы используете Google по умолчанию для базы данных, которую используете, вы должны знать, что это должно быть. Если вы получаете `org.hibernate.HibernateException: Connection cannot be null when 'hibernate.dialect' not set HibernateException` с сообщением `org.hibernate.HibernateException: Connection cannot be null when 'hibernate.dialect' not set` и вы следуете этому руководству, вероятность 90% вашего URL неверна, вероятность 5% что ваша база данных не запущена и 5% вероятность неправильного имени пользователя / пароля.

- *имя пользователя* : имя пользователя, которое будет использоваться при аутентификации с помощью базы данных.

- *password* : пароль для использования при аутентификации с базой данных.

Следующим шагом будет создание `SessionFactory` . Это то, что Hibernate использует для создания и управления транзакциями и фактически ведет переговоры с базой данных. В нем есть несколько вариантов конфигурации, которые я попытаюсь объяснить ниже.

```
<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* : ваш компонент источника данных. Если вы изменили Идентификатор источника данных, установите его здесь.

- *packagesToScan* : пакеты для сканирования, чтобы найти ваши аннотированные объекты JPA. Это те объекты, которые требуется управлять фабрикой сеансов, обычно будут POJO и аннотируются с `@Entity` . Дополнительные сведения о настройке отношений объектов в Hibernate см. Здесь .

- *annotatedClasses* (не показано): вы также можете предоставить список классов для Hibernate для сканирования, если они не все находятся в одном пакете. Вы должны использовать либо `packagesToScan` либо `annotatedClasses` но не оба. Декларация выглядит так:

```
<property name="annotatedClasses">
    <list>
```

```
        <value>foo.bar.package.model.Person</value>
        <value>foo.bar.package.model.Thing</value>
    </list>
</property>
```

- *hibernateProperties* : Существует огромное количество всех этих любовно документированных здесь . Основными из них будут следующие:
- *hibernate.hbm2ddl.auto* : Один из самых жарких вопросов Hibernate описывает это свойство. См. Дополнительную информацию . Обычно я использую validate и настраиваю свою базу данных с использованием SQL-скриптов (для внутренней памяти) или заранее создавая базу данных (существующая база данных).
- *hibernate.show_sql* : Boolean flag, если true Hibernate будет печатать весь SQL, который он генерирует в `stdout` . Вы также можете настроить регистратор для отображения значений, привязанных к запросам, путем установки `log4j.logger.org.hibernate.type=TRACE` `log4j.logger.org.hibernate.SQL=DEBUG` в вашем менеджере журналов (я использую log4j).
- *hibernate.format_sql* : Boolean flag, приведет к тому, что Hibernate будет корректно печатать ваш SQL в stdout.
- *hibernate.dialect* (Не показано, по уважительной причине): Многие старые учебники показывают, как установить диалоги Hibernate, которые он будет использовать для связи с вашей базой данных. Hibernate **может** автоматически определять, какой диалект использовать на основе драйвера JDBC, который вы используете. Поскольку существует около 3 различных диалектов Oracle и 5 разных диалектов MySQL, я бы оставил это решение до Hibernate. Полный список диалектов Hibernate поддерживает смотрите здесь .

Последние 2 бобов, которые вам нужно объявить:

```
<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` переводит исключительные исключения `HibernateException` или `SQLExceptions` в Spring, которые могут быть поняты контекстом приложения.

`TransactionManager` - это то, что контролирует транзакции, а также откаты.

Примечание. Вы должны авторизовать свой компонент `SessionFactory` в своих DAO.

Конфигурация Hibernate без XML

Этот пример был взят отсюда

```
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();
    }


}
```

Обратите внимание, что при использовании новейшего Hibernate этот подход работает неэффективно (выпуск Hibernate 5.2 все еще допускает эту конфигурацию)

Простой пример спящего режима с использованием XML

Чтобы настроить простой спящий проект с использованием XML для конфигураций, вам нужно 3 файла, hibernate.cfg.xml, POJO для каждого объекта и EntityName.hbm.xml для каждого объекта. Ниже приведен пример использования 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 и testPassword будут заменены. Обязательно используйте полное имя ресурса, если оно находится в пакете.

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>
```

Опять же, если класс находится в пакете, используйте полное имя класса packageName.className.

После того, как у вас есть эти три файла, вы готовы использовать спящий режим в своем проекте.

Прочитайте Начало работы с спящим режимом онлайн:
https://riptutorial.com/ru/hibernate/topic/907/начало-работы-с-спящим-режимом

глава 2: HQL

Вступление

HQL - это язык запросов Hibernate, он основан на SQL, а за кулисами - в SQL, но синтаксис отличается. Вы используете имена сущностей / классов, а не имена таблиц и имена полей, а не имена столбцов. Он также позволяет много сокращений.

замечания

Главное, что следует помнить при использовании hql, - это использование имени класса и имен полей вместо имен таблиц и столбцов, к которым мы привыкли в SQL.

Examples

Выбор всей таблицы

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

Выберите определенные столбцы

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

Включить предложение Where

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

Присоединиться

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

Прочитайте HQL онлайн: https://riptutorial.com/ru/hibernate/topic/9388/hql

глава 3: Включить / отключить SQL-журнал

замечания

Регистрация этих запросов происходит **медленно** , даже медленнее, чем обычно Hibernate. Он также использует огромное количество бревен. Не используйте запись в сценариях, где требуется производительность. Используйте это только при тестировании запросов, которые генерирует Hibernate.

Examples

Использование файла конфигурации ведения журнала

В выбранном вами файле конфигурации регистрации выполните регистрацию следующих пакетов до указанных уровней:

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

Вероятно, для этого потребуются некоторые префиксы регистратора.

Конфигурация Log4j:

```
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"/>
```

Использование свойств гибернации

Это покажет вам сгенерированный SQL, но не покажет вам значения, содержащиеся в запросах.

```
<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>
```

Включить / отключить отладку SQL-журнала

Некоторые приложения, использующие Hibernate, генерируют огромное количество SQL при запуске приложения. Иногда лучше включить / отключить SQL-журнал в определенных точках при отладке.

Чтобы включить, просто запустите этот код в своей среде IDE при отладке приложения:

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

Отключить:

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

Прочитайте Включить / отключить SQL-журнал онлайн:
https://riptutorial.com/ru/hibernate/topic/3548/включить---отключить-sql-журнал

глава 4: Извлечение в спящий режим

Вступление

Извлечение действительно важно в JPA (Java Persistence API). В JPA для извлечения сущностей на основе их отношений используются HQL (язык запросов Hibernate) и JPQL (Java Query Language Query Language). Хотя это гораздо лучше, чем использование так много присоединяющихся запросов и подзапросов, чтобы получить то, что мы хотим, используя собственный SQL, стратегия, как мы получаем ассоциированные объекты в JPA, по-прежнему существенно влияет на производительность нашего приложения.

Examples

Рекомендуется использовать FetchType.LAZY. Присоедините выборку столбцов, когда они понадобятся.

Ниже представлен класс сущностей Employer, который сопоставляется с работодателем таблицы. Как вы видите, я использовал **fetch = FetchType.LAZY** вместо fetch = FetchType.EAGER. Причина, по которой я использую LAZY, заключается в том, что у Работодателя может быть много свойств позже, и каждый раз, когда мне может не понадобиться знать все поля Работодателя, поэтому загрузка всех из них приведет к плохой производительности, тогда загружается работодатель.

```
@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;
        }
    }
```

Тем не менее, для ассоциаций LAZY, неинициализированные прокси иногда приводят к исключению LazyInitializationException. В этом случае мы можем просто использовать JOIN FETCH в HQL / JPQL, чтобы избежать 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
```

Прочитайте Извлечение в спящий режим онлайн:
https://riptutorial.com/ru/hibernate/topic/9475/извлечение-в-спящий-режим

глава 5: Критерий и прогнозы

Examples

Список с использованием ограничений

Предполагая, что у нас есть таблица TravelReview с названиями городов в качестве столбца "title"

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

Мы можем добавить ограничения к критериям, связав их следующим образом:

```
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();
```

Использование прогнозов

Если мы хотим получить только несколько столбцов, мы можем использовать класс Projections для этого. Например, следующий код извлекает столбец заголовка

```
// 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();
```

Использовать фильтры

`@Filter` используется как лагерь `WHERE` , вот некоторые примеры

Студенческая организация

```
@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 */
}
```

Учебное предприятие

```
@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 */
}
```

Студенческая студия

```
@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 */
}
```

Таким образом, каждый раз, когда включен «активный» фильтр,

-Каждый запрос, который мы делаем в студенческом суде, вернет **ВСЕХ** студентов **ТОЛЬКО** их `state = true` исследования

-Каждый запрос, который мы делаем на учебном объекте, вернет **ВСЕ** `state = true`

исследования

-Каждый запрос, который мы делаем на StudentStudy entiy, будет возвращать **ТОЛЬКО** те, у которых `state = true` отношение исследования

Обратите внимание, что study_id - это имя поля в таблице sql StudentStudy

Прочитайте Критерий и прогнозы онлайн: https://riptutorial.com/ru/hibernate/topic/3939/критерий-и-прогнозы

глава 6: Кэширование

Examples

Включение кэширования Hibernate в WildFly

Чтобы включить кэширование второго уровня для спящего режима в WildFly, добавьте это свойство в файл `persistence.xml` :

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

Вы также можете включить Query Caching с этим свойством:

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

WildFly не требует, чтобы вы определяли поставщика кэша при включении второго уровня кэша Hibernate, поскольку по умолчанию используется Infinispan. Однако, если вы хотите использовать альтернативный поставщик кэшей, вы можете сделать это с помощью свойства `hibernate.cache.provider_class` .

Прочитайте Кэширование онлайн: https://riptutorial.com/ru/hibernate/topic/3462/кэширование

глава 7: Лётная загрузка против желаемой загрузки

Examples

Лётная загрузка против желаемой загрузки

Извлечение или загрузка данных можно в первую очередь классифицировать на два типа: нетерпеливые и ленивые.

Чтобы использовать Hibernate, обязательно добавьте последнюю версию этого файла в раздел зависимостей вашего файла pom.xml:

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

1. Желание загрузки и ленивой загрузки

Первое, что мы должны обсудить здесь, - это то, что ленивая загрузка и интенсивная загрузка:

Eager Loading - это шаблон проектирования, в котором происходит инициализация данных на месте. Это означает, что коллекции извлекаются полностью во время получения родительских прав (выборка сразу)

Lazy Loading - это шаблон дизайна, который используется для отсрочки инициализации объекта до точки, в которой он необходим. Это может эффективно влиять на производительность приложения.

2. Использование разных типов загрузки

Ленивую загрузку можно включить с помощью следующего параметра XML:

```
lazy="true"
```

Давайте рассмотрим пример. Сначала у нас есть класс 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
    }
```

Посмотрите на Set of OrderDetail, который у нас есть. Теперь давайте посмотрим на **класс OrderDetail** :

```
public class OrderDetail implements Serializable {

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

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

Важная часть, которая связана с установкой ленивой загрузки в `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>
```

Так разрешена ленивая загрузка. Чтобы отключить ленивую загрузку, мы можем просто использовать: `lazy = "false"` и это, в свою очередь, позволит загружать с нетерпением. Ниже приведен пример настройки активной загрузки в другом файле 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>
```

Объем

Для тех, кто не играл с этими двумя проектами, объем ленивых и нетерпеливых находится на определенной **сессии** *SessionFactory* . *Легко* загружает все мгновенно, означает, что нет необходимости называть что-либо для его получения. Но ленивый выбор обычно требует некоторых действий для извлечения сопоставленной коллекции / объекта. Иногда это проблематично, когда вы получаете ленивую выборку за пределами *сеанса* . Например, у вас есть представление, которое показывает детали некоторого отображаемого POJO.

```
@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();
    }
}
```

Когда вы попытаетесь получить **ленивый выбор** вне *сеанса,* вы получите *lazyinitializeException* . Это связано с тем, что по умолчанию стратегия выборки для всего oneToMany или любого другого отношения является *ленивым* (вызов по требованию по требованию), а когда вы закрыли сеанс, у вас нет возможности связываться с базой данных. поэтому наш код пытается получить подборку *понравившегося файла* и он выдает исключение, потому что не существует связанного сеанса для рендеринга БД.

Решением для этого является использование:

1. Открыть сеанс в представлении - в котором вы держите сессию открытой даже на визуализированном представлении.
2. `Hibernate.initialize(user.getLikedPage())` перед закрытием сессии. Это говорит о спящем режиме для инициализации элементов коллекции

Прочитайте Лётная загрузка против желаемой загрузки онлайн:
https://riptutorial.com/ru/hibernate/topic/7249/лётная-загрузка-против-желаемой-загрузки

глава 8: Настройка производительности

Examples

Не используйте тип выборки EAGER

Спящий режим может использовать два типа извлечения при сопоставлении отношений между двумя объектами: `EAGER` и `LAZY` .

В общем `EAGER` тип `EAGER` выборки не является хорошей идеей, потому что он сообщает JPA, чтобы *всегда* извлекать данные, даже если эти данные не нужны.

Например, если у вас есть объект `Person` и отношения с `Address` например:

```
@Entity
public class Person {

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

}
```

В любое время, когда вы запрашиваете `Person` , будет возвращен и список `Address` этого `Person` .

Таким образом, вместо того, чтобы сопоставлять вашу сущность с:

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

Использование:

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

Еще одна вещь, на которую нужно обратить внимание, - отношения `@OneToOne` и `@ManyToOne` . *По умолчанию* оба они EAGER. Итак, если вас беспокоит производительность вашего приложения, вам нужно установить выборку для этого типа отношений:

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

А также:

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

Использовать композицию вместо наследования

Hibernate имеет несколько стратегий наследования. `JOINED` наследования `JOINED` выполняет JOIN между дочерним объектом и родительским объектом.

Проблема с этим подходом заключается в том, что Hibernate **всегда** выводит данные всех вовлеченных таблиц в наследование.

Например, если у вас есть объекты `Bicycle` и `MountainBike` используя `JOINED` наследования `JOINED` :

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

}
```

А также:

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

}
```

Любой запрос JPQL, который попадает в `MountainBike` приведет к данным `Bicycle` , создав SQL-запрос, например:

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

Если у вас есть другой родительский элемент для `Bicycle` (например, `Transport` , например), этот вышеприведенный запрос также приведет данные от этого родителя, делая дополнительный JOIN.

Как вы можете видеть, это своего рода сопоставление `EAGER` . У вас нет выбора приносить только данные таблицы `MountainBike` используя эту стратегию наследования.

Лучшим для исполнения является использование композиции вместо наследования.

Для этого вы можете сопоставить объект `MountainBike` с полевым `bicycle` :

```
@Entity
public class MountainBike {

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

}
```

И `Bicycle` :

```
@Entity
```

```
public class Bicycle {

}
```

В каждом запросе теперь будут отображаться только данные `MountainBike` по умолчанию.

Прочитайте Настройка производительности онлайн:
https://riptutorial.com/ru/hibernate/topic/2326/настройка-производительности

глава 9: Связывание сущностей Hibernate с использованием аннотаций

параметры

аннотирование	подробности
`@OneToOne`	Определяет отношение «один к одному» с соответствующим объектом.
`@OneToMany`	Указывает один объект, который сопоставляется со многими объектами.
`@ManyToOne`	Задает набор объектов, которые отображаются на один объект.
`@Entity`	Задает объект, который сопоставляется с таблицей базы данных.
`@Table`	Задает таблицу базы данных, к которой относится этот объект.
`@JoinColumn`	Указывает, в каком столбце хранится ключ foregin.
`@JoinTable`	Задает промежуточную таблицу, в которой хранятся внешние ключи.

Examples

Двунаправленные от многих до многих, используя пользовательский интерфейс.

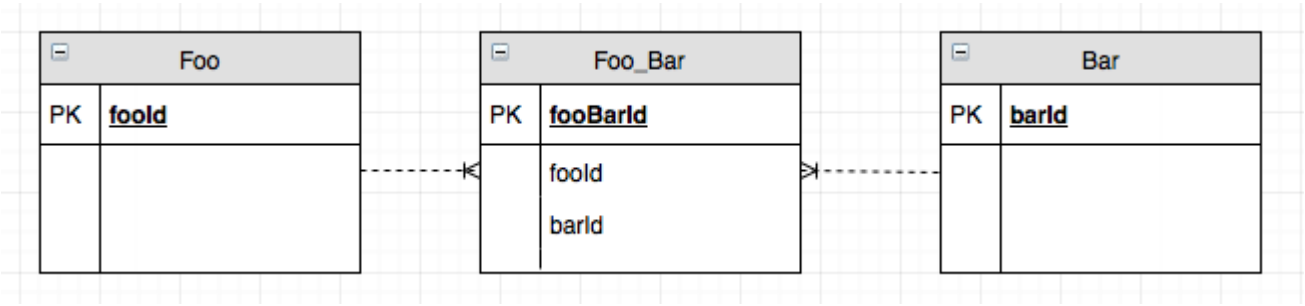
```
@Entity
@Table(name="FOO")
public class Foo {
    private UUID fooId;

    @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.
}
```

Определяет двустороннюю связь между многими объектами `Foo` для многих объектов `Bar` используя промежуточную таблицу соединений, которую управляет пользователь.

Объекты `Foo` хранятся в виде строк в таблице под названием `FOO` . Объекты `Bar` хранятся в виде строк в таблице под названием `BAR` . Отношения между объектами `Foo` и `Bar` хранятся в таблице `FOO_BAR` . В приложении есть объект `FooBar` .

Обычно используется, когда вы хотите хранить дополнительную информацию об объекте объединения, например дату создания отношения.



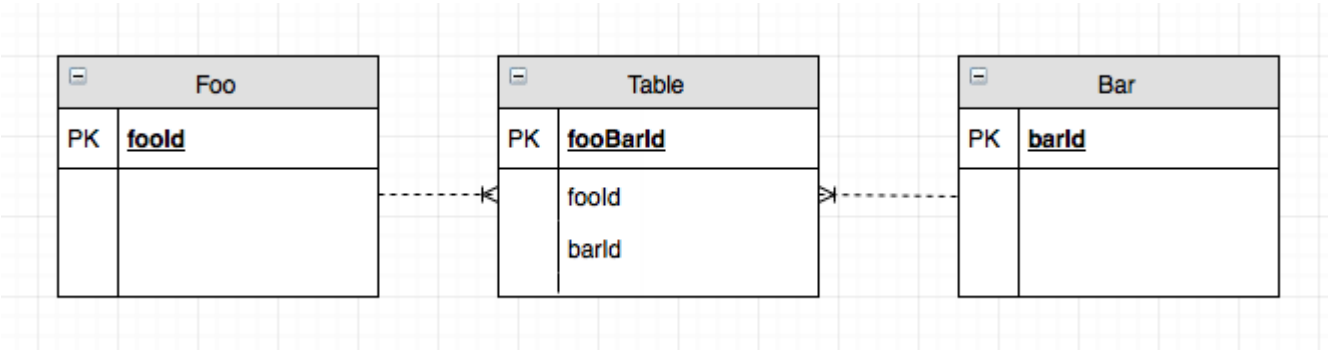
Двунаправленное много для многих, используя таблицу управления соединением Hibernate

```
@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;
}
```

Задает связь между многими объектами `Foo` для многих объектов `Bar` используя промежуточную таблицу соединений, которую управляет Hibernate.

Объекты `Foo` хранятся в виде строк в таблице под названием `FOO` . Объекты `Bar` хранятся в виде строк в таблице под названием `BAR` . Отношения между объектами `Foo` и `Bar` хранятся в таблице `FOO_BAR` . Однако это означает, что объект `FooBar` не `FooBar` частью приложения.



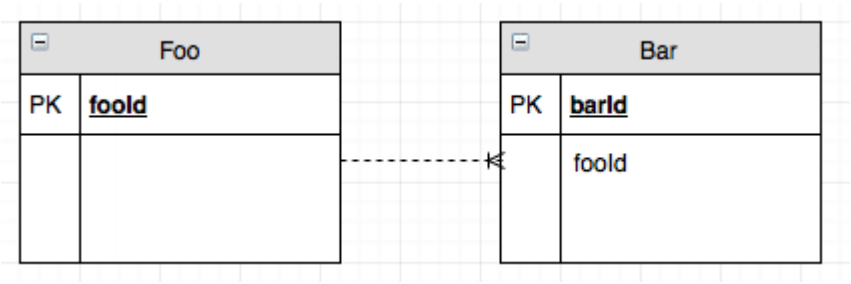
Двунаправленные отношения «один ко многим» с использованием сопоставления внешних ключей

```
@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;
}
```

Задает двухстороннюю связь между одним объектом `Foo` и многими объектами `Bar` с использованием внешнего ключа.

Объекты `Foo` хранятся в виде строк в таблице под названием `FOO` . Объекты `Bar` хранятся в виде строк в таблице под названием `BAR` . Внешний ключ хранится в таблице `BAR` в столбце `fooId`

.



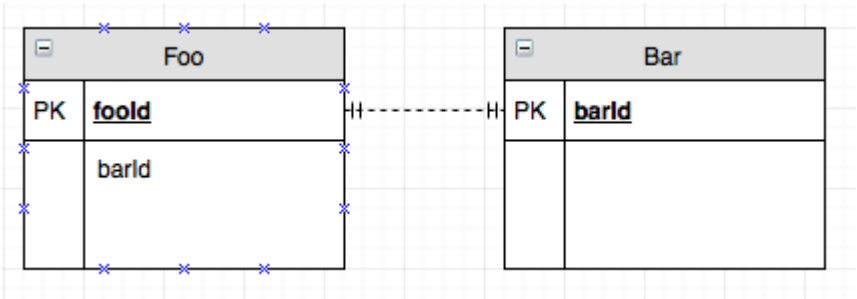
Двунаправленные отношения «один к одному», управляемые Foo.class

```
@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;
}
```

Указывает двустороннюю связь между одним объектом `Foo` и одним объектом `Bar` с использованием внешнего ключа.

Объекты `Foo` хранятся в виде строк в таблице под названием `FOO` . Объекты `Bar` хранятся в виде строк в таблице под названием `BAR` . Внешний ключ хранится в таблице `FOO` в столбце `barId` .

Обратите внимание, что значение `mappedBy` - это имя поля для объекта, а не имя столбца.



Однонаправленные отношения «один ко многим» с использованием таблицы соединений с управляемым пользователем

```
@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.
}
```

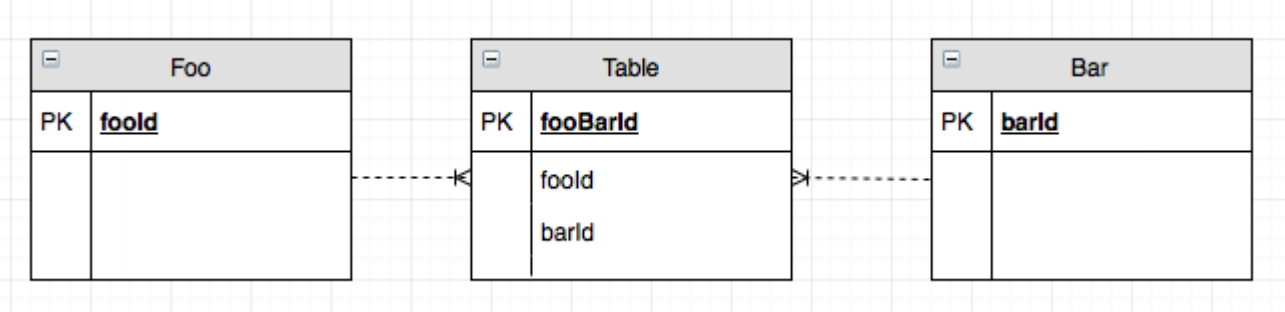
Задает одностороннюю связь между одним объектом `Foo` и многими объектами `Bar` используя промежуточную таблицу соединений, которую управляет пользователь.

Это похоже на отношения `ManyToMany` , но если вы добавите `unique` ограничение к целевому внешнему ключу, вы можете `OneToMany` его использовать `OneToMany` .

Объекты `Foo` хранятся в виде строк в таблице под названием `FOO` . Объекты `Bar` хранятся в виде строк в таблице под названием `BAR` . Отношения между объектами `Foo` и `Bar` хранятся в таблице `FOO_BAR` . В приложении есть объект `FooBar` .

Обратите внимание, что отображение объектов `Bar` обратно на объекты `Foo` . Объекты `Bar` могут свободно манипулировать, не затрагивая объекты `Foo` .

Очень часто используется с Spring Security при настройке объекта `User` которого есть список `Role` , который они могут выполнять. Вы можете добавлять и удалять роли для пользователя, не беспокоясь о том, что каскады удаляют `Role` .



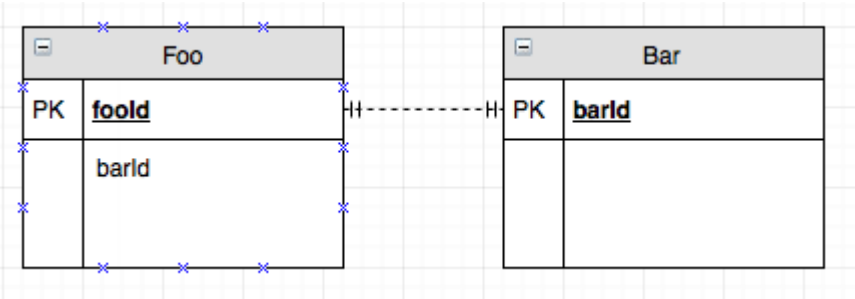
Однонаправленные отношения «один к одному»

```
@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
}
```

Задает одностороннюю связь между одним объектом `Foo` и одним объектом `Bar` .

Объекты `Foo` хранятся в виде строк в таблице под названием `FOO` . Объекты `Bar` хранятся в виде строк в таблице под названием `BAR` .

Обратите внимание, что отображение объектов `Bar` обратно на объекты `Foo` . Объекты `Bar` могут свободно манипулировать, не затрагивая объекты `Foo` .



Прочитайте Связывание сущностей Hibernate с использованием аннотаций онлайн: https://riptutorial.com/ru/hibernate/topic/5742/связывание-сущностей-hibernate-с-использованием-аннотаций

глава 10: Собственные SQL-запросы

Examples

Простой запрос

Предполагая, что у вас есть дескриптор объекта `Session` Hibernate, в этом случае называется `session` :

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

Это будет извлекать все строки в `some_table` и помещать их в переменную `result` и печатать каждое значение.

Пример получения уникального результата

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

с этим запросом вы получаете уникальный результат, когда знаете, что результат запроса всегда будет уникальным.

И если запрос возвращает более одного значения, вы получите исключение

    org.hibernate.NonUniqueResultException

Вы также можете проверить подробности в этой ссылке здесь с большим количеством описаний

Поэтому, пожалуйста, убедитесь, что вы знаете, что запрос вернет уникальный результат

Прочитайте Собственные SQL-запросы онлайн: https://riptutorial.com/ru/hibernate/topic/6978/собственные-sql-запросы

глава 11: Сопоставительные ассоциации

Examples

От одного до одного спящего режима

Каждая страна имеет один капитал. Каждая столица имеет одну страну.

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();
  }

}
```

Прочитайте Сопоставительные ассоциации онлайн:
https://riptutorial.com/ru/hibernate/topic/6478/сопоставительные-ассоциации

глава 12: Сопоставление ассоциаций между объектами

Examples

Ассоциация OneToMany

Чтобы проиллюстрировать отношение OneToMany, нам нужны 2 объекта, например «Страна и город». Одна страна имеет несколько городов.

В CountryEntity beloww мы определяем множество городов для страны.

```
@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
}
```

Теперь город.

```
@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
}
```

Связь между многими пользователями с использованием XML

Это пример того, как сделать сопоставление от одного до многих с помощью XML. Мы будем использовать Автору и Книгу в качестве нашего примера и предположим, что автор написал много книг, но в каждой книге будет только один автор.

Авторский класс:

```
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;
    }
}
```

Класс книги:

```
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>
```

То, что делает соединение одним из многих, заключается в том, что класс Book содержит Author, а xml имеет тег <many-to-one>. Атрибут cascade позволяет вам установить, как дочерний объект будет сохранен / обновлен.

Прочитайте Сопоставление ассоциаций между объектами онлайн:
https://riptutorial.com/ru/hibernate/topic/6165/сопоставление-ассоциаций-между-объектами

глава 13: Спящий режим и JPA

Examples

Связь между Hibernate и JPA

Hibernate - это реализация стандарта JPA . Таким образом, все сказано, что для Hibernate также справедливо.

Hibernate имеет некоторые расширения для JPA. Кроме того, способ создания поставщика JPA зависит от поставщика. Этот раздел документации должен содержать только то, что характерно для Hibernate.

Прочитайте Спящий режим и JPA онлайн: https://riptutorial.com/ru/hibernate/topic/6313/спящий-режим-и-jpa

глава 14: Стратегия пользовательских именований

Examples

Создание и использование пользовательской ImplicitNamingStrategy

Создание пользовательской `ImplicitNamingStrategy` позволяет вам настроить, как Hibernate присваивает имена неявным образом атрибутам `Entity` , включая внешние ключи, уникальные ключи, столбцы идентификаторов, базовые столбцы и т. Д.

Например, по умолчанию Hibernate генерирует внешние ключи, которые хэшируются и выглядят похожими на:

```
FKe6hidh4u0qh8y1ijy59s2ee6m
```

Хотя это часто не является проблемой, вы можете пожелать, чтобы имя было более наглядным, например:

```
FK_asset_tenant
```

Это легко сделать с помощью пользовательской `ImplicitNamingStrategy` .

Этот пример расширяет `ImplicitNamingStrategyJpaCompliantImpl` , однако вы можете выбрать `ImplicitNamingStrategy` если хотите.

```
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());
    }

}
```

Для того, чтобы сказать , спящий режим , который `ImplicitNamingStrategy` использовать, определить `hibernate.implicit_naming_strategy` свойство в ваших `persistence.xml` или `hibernate.cfg.xml` файл , как показано ниже:

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

Или вы можете указать свойство в файле `hibernate.properties` как показано ниже:

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

В этом примере все внешние ключи, которые не имеют явно определенного `name` , теперь получат свое имя из `CustomNamingStrategy` .

Пользовательская стратегия физического наименования

При сопоставлении наших объектов с именами таблиц базы данных мы полагаемся на аннотацию `@Table` . Но если у нас есть соглашение об именах для имен таблиц базы данных, мы можем реализовать пользовательскую стратегию физического именования, чтобы сообщить hibernate рассчитать имена таблиц на основе имен сущностей без явного указания этих имен с `@Table` аннотации `@Table` . То же самое касается отображения атрибутов и столбцов.

Например, название нашей организации:

```
ApplicationEventLog
```

И наше имя таблицы:

```
application_event_log
```

Наша стратегия физического именования должна конвертироваться из имен сущностей, которые являются верблюжьим футляром, к нашим именам таблиц db, которые представляют собой случай змей. Мы можем добиться этого, расширив `PhysicalNamingStrategyStandardImpl` Hibernate's `PhysicalNamingStrategyStandardImpl` :

```
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);
    }
}
```

Мы переопределяем поведение по умолчанию методов `toPhysicalTableName` и `toPhysicalColumnName` для применения нашего соглашения об именах db.

Чтобы использовать нашу пользовательскую реализацию, нам нужно определить свойство `hibernate.physical_naming_strategy` и присвоить ей имя нашего класса `PhysicalNamingStrategyImpl` .

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

Таким образом, мы можем облегчить наш код из `@Table` и `@Column` аннотаций, поэтому наш класс сущностей:

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

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

будет правильно отображаться в таблице db:

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

Как видно из приведенного выше примера, мы можем явно указать имя объекта db, если по какой-либо причине оно не соответствует нашему общему соглашению об именах: `@Column(name="finish_dtl")`

Прочитайте Стратегия пользовательских именований онлайн: https://riptutorial.com/ru/hibernate/topic/3051/стратегия-пользовательских-именований

кредиты

S. No	Главы	Contributors
1	Начало работы с спящим режимом	Community, JamesENL, Michael Piefel, Naresh Kumar, Reborn , user7491506
2	HQL	Daniel Käfer, user7491506
3	Включить / отключить SQL-журнал	Daniel Käfer, Dherik, JamesENL, Michael Piefel
4	Извлечение в спящий режим	rObOtAndChalie
5	Критерий и прогнозы	Saifer, Sameer Srivastava
6	Кэширование	Mitch Talmadge
7	Лётная загрузка против желаемой загрузки	BELLIL, Pramod, Pritam Banerjee, vicky
8	Настройка производительности	Dherik, Michael Piefel
9	Связывание сущностей Hibernate с использованием аннотаций	Aleksei Loginov, JamesENL
10	Собственные SQL-запросы	Daniel Käfer, Nathaniel Ford, Sandeep Kamath
11	Сопоставительные ассоциации	Dherik, omkar sirra
12	Сопоставление ассоциаций между объектами	StanislavL, user7491506
13	Спящий режим и	Michael Piefel

	JPA	
14	Стратегия пользовательских именований	Mitch Talmadge, Naresh Kumar, veljkost