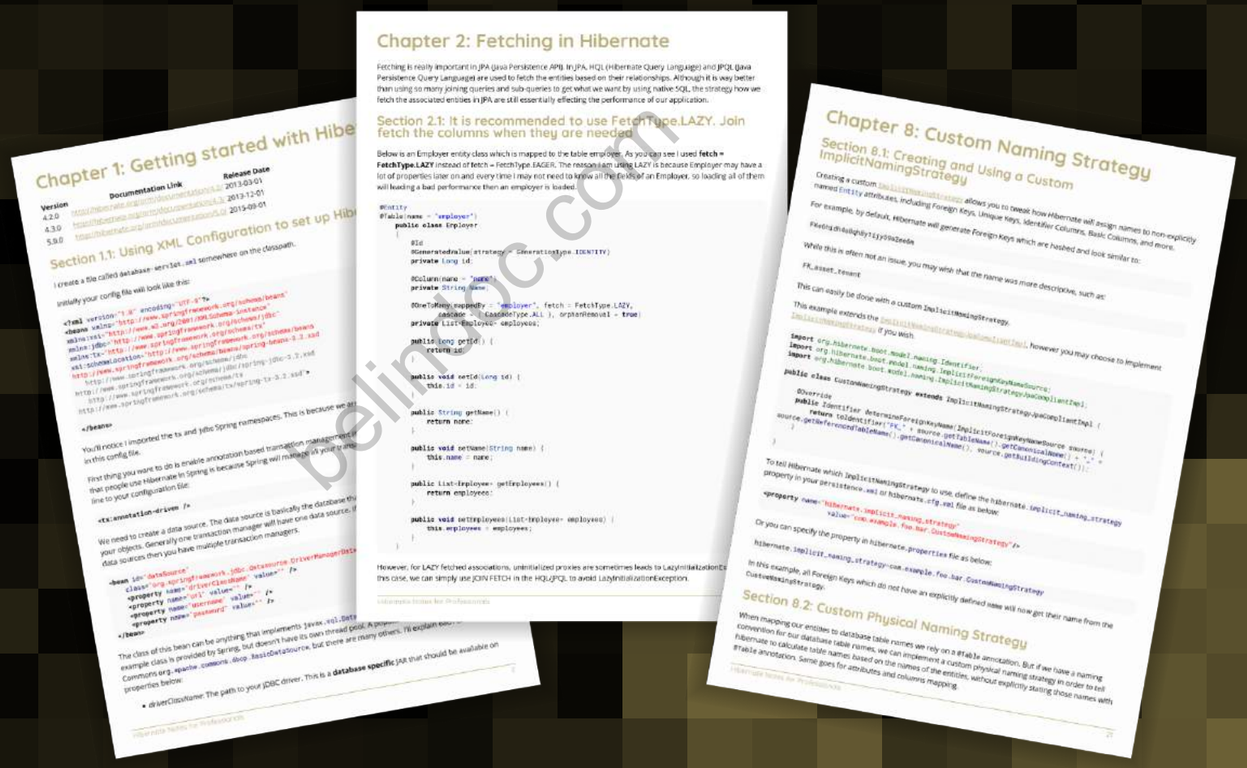
专业人员笔记

Hibernate

专业人员笔记

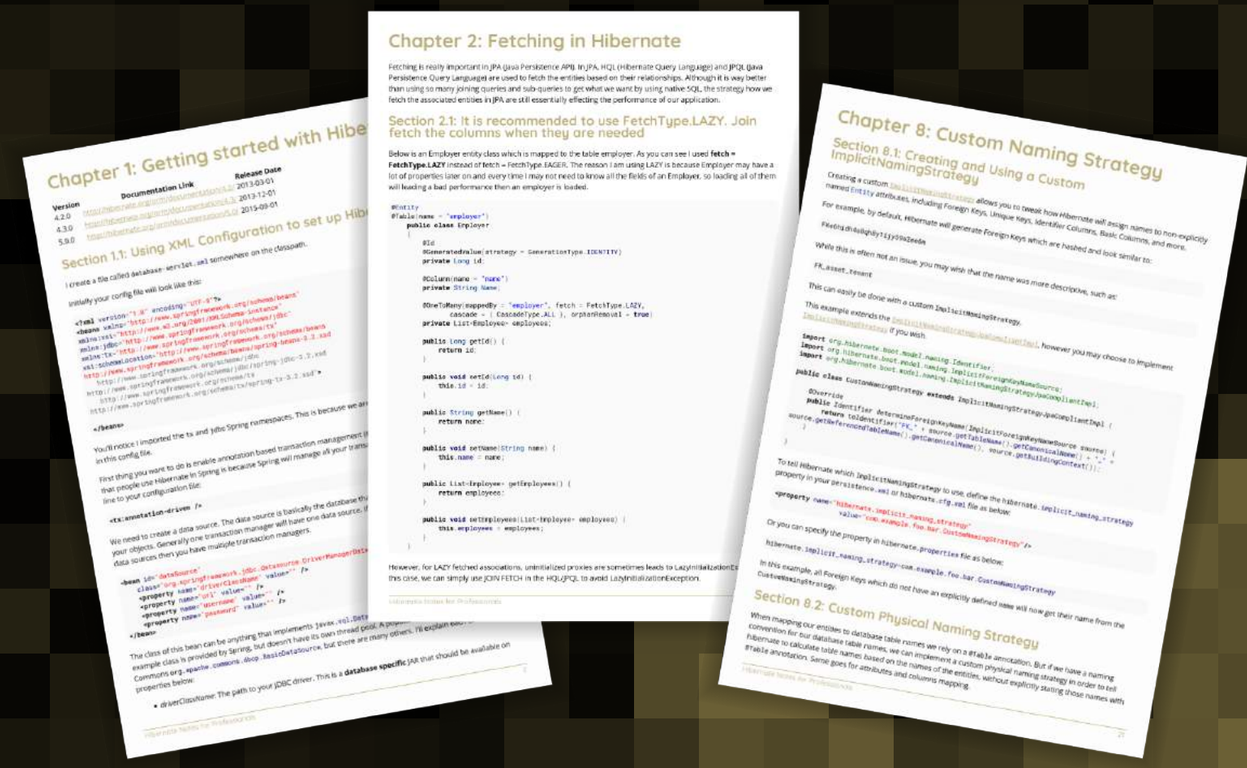


30多页

专业提示和技巧

Hibernate

Notes for Professionals



30+ pages

of professional hints and tricks

目录

Contents

关于

欢迎随意免费分享此 PDF，
本书的最新版本可从以下网址下载：
https://goalkicker.com/HibernateBook

本*Hibernate 专业人士笔记*一书汇编自Stack Overflow文档，内容由 Stack Overflow 的优秀贡献者撰写。
文本内容采用知识共享署名-相同方式共享许可协议发布，详见本书末尾对各章节贡献者的致谢。图片版权归各自所有者所有，除非另有说明。

本书为非官方免费教材，旨在教育用途，与官方 Hibernate 组织或公司及 Stack Overflow 无关。所有商标和注册商标均为其各自公司所有。

本书所提供的信息不保证正确或准确，使用风险自负。

请将反馈和更正发送至web@petercv.com

About

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
https://goalkicker.com/HibernateBook

This *Hibernate Notes for Professionals* book is compiled from Stack Overflow Documentation, the content is written by the beautiful people at Stack Overflow. Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official Hibernate group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

第1章：Hibernate入门

版本	文档链接	发布日期
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

第1.1节：使用XML配置设置Hibernate

我创建了一个名为database-servlet.xml的文件，放在类路径的某个位置。

最初你的配置文件看起来会是这样的：

```
<?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>
```

你会注意到我导入了 tx 和 jdbc 这两个Spring命名空间。这是因为我们将在这个配置文件中大量使用它们。

你首先要做的是启用基于注解的事务管理（@Transactional）。人们使用Spring中的Hibernate的主要原因是Spring会帮你管理所有事务。向你的配置文件中添加以下行：

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

我们需要创建一个数据源。数据源基本上是Hibernate用来持久化你的对象的数据库。通常一个事务管理器会有一个数据源。如果你想让Hibernate连接多个数据源，那么你就需要多个事务管理器。

```
<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>
```

该 Bean 的类可以是实现了 javax.sql.DataSource 的任何类，因此你可以自己编写。这个示例类由 Spring 提供，但它没有自己的线程池。一个流行的替代方案是 Apache Commons org.apache.commons.dbcp.BasicDataSource ，但还有许多其他选择。下面我将解释每个属性：

- driverClassName：您的JDBC驱动程序的路径。这是一个特定于数据库的JAR文件，应当可用

Chapter 1: Getting started with Hibernate

Version	Documentation Link	Release Date
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

Section 1.1: Using XML Configuration to set up Hibernate

I create a file called `database-servlet.xml` somewhere on the classpath.

Initially your config file will look like this:

```
<?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>
```

You'll notice I imported the `tx` and `jdbc` Spring namespaces. This is because we are going to use them quite heavily in this config file.

First thing you want to do is enable annotation based transaction management (`@Transactional`). The main reason that people use Hibernate in Spring is because Spring will manage all your transactions for you. Add the following line to your configuration file:

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

We need to create a data source. The data source is basically the database that Hibernate is going to use to persist your objects. Generally one transaction manager will have one data source. If you want Hibernate to talk to multiple data sources then you have multiple transaction managers.

```
<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>
```

The class of this bean can be anything that implements `javax.sql.DataSource` so you could write your own. This example class is provided by Spring, but doesn't have its own thread pool. A popular alternative is the Apache Commons `org.apache.commons.dbcp.BasicDataSource`, but there are many others. I'll explain each of the properties below:

- *driverClassName*: The path to your JDBC driver. This is a **database specific** JAR that should be available on

你的类路径。确保你拥有最新版本。如果你使用的是Oracle数据库，你将需要一个Oracle驱动程序。如果你使用的是MySQL数据库，你将需要一个MySQL驱动程序。看看你是否能在这里找到你需要的驱动程序但快速谷歌搜索应该能帮你找到正确的驱动程序。

- url：您的数据库的URL。通常这将是类似于 jdbc:oracle:thin:\patho\your\database 或 jdbc:mysql://path/to/your/database。如果你在网上搜索你所使用数据库的默认位置，应该能找到正确的路径。如果你遇到带有消息 org.hibernate.HibernateException 的 HibernateException 异常：

  当未设置 'hibernate.dialect' 且您正在遵循此指南时，连接 不能为 null，90%的可能性是您的URL错误，5%的可能性是您的数据库未启动，5%的可能性是您的用户名/密码错误。

- username：用于数据库身份验证的用户名。
- password：用于数据库身份验证的密码。

接下来是设置 SessionFactory。这是Hibernate用来创建和管理您的事务，并实际与数据库通信的组件。它有相当多的配置选项，下面我会尝试解释。

```
<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：您的数据源bean。如果您更改了dataSource的Id，请在此处设置。
- packagesToScan：扫描以查找您的JPA注解对象的包。这些对象是会话工厂需要管理的，通常是POJO，并带有@Entity注解。有关如何在Hibernate中设置对象关系的更多信息请参见这里。
- annotatedClasses（未显示）：如果它们不都在同一个包中，您也可以提供Hibernate扫描的类列表。您应该使用packagesToScan或annotatedClasses中的一个，而不是两个都用。声明如下：
  声明如下：

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

- hibernateProperties：这里有大量的属性，均有详细的文档说明。您主要会使用以下属性：
- hibernate.hbm2ddl.auto：这是Hibernate中最热门的问题之一，详细说明了此属性。查看更多信息。我通常使用validate，并通过SQL脚本（用于内存数据库）或预先创建数据库（已有数据库）来设置数据库。
- hibernate.show_sql：布尔标志，如果为true，Hibernate会将它生成的所有SQL打印到stdout。您还可以通过配置日志记录器来显示绑定到查询的值，方法是设置

your classpath. Ensure that you have the most up to date version. If you are using an Oracle database, you'll need a OracleDriver. If you have a MySQL database, you'll need a MySQLDriver. See if you can find the driver you need here but a quick google should give you the correct driver.

- *url*: The URL to your database. Usually this will be something like `jdbc\:oracle\:thin\:\path\to\your\database` or `jdbc:mysql://path/to/your/database`. If you google around for the default location of the database you are using, you should be able to find out what this should be. If you are getting a `HibernateException` with the message `org.hibernate.HibernateException: Connection cannot be null when 'hibernate.dialect' not set` and you are following this guide, there is a 90% chance that your URL is wrong, a 5% chance that your database isn't started and a 5% chance that your username/password is wrong.
- *username*: The username to use when authenticating with the database.
- *password*: The password to use when authenticating with the database.

The next thing, is to set up the `SessionFactory`. This is the thing that Hibernate uses to create and manage your transactions, and actually talks to the database. It has quite a few configuration options that I will try to explain below.

```
<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*: Your data source bean. If you changed the Id of the dataSource, set it here.
- *packagesToScan*: The packages to scan to find your JPA annotated objects. These are the objects that the session factory needs to manage, will generally be POJO's and annotated with `@Entity`. For more information on how to set up object relationships in Hibernate see here.
- *annotatedClasses* (not shown): You can also provide a list of classes for Hibernate to scan if they are not all in the same package. You should use either `packagesToScan` or `annotatedClasses` but not both. The declaration looks like this:

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

- *hibernateProperties*: There are a myriad of these all lovingly documented here. The main ones you will be using are as follows:
- *hibernate.hbm2ddl.auto*: One of the hottest Hibernate questions details this property. See it for more info. I generally use validate, and set up my database using either SQL scripts (for an in-memory), or create the database beforehand (existing database).
- *hibernate.show_sql*: Boolean flag, if true Hibernate will print all the SQL it generates to `stdout`. You can also configure your logger to show you the values that are being bound to the queries by setting

log4j.logger.org.hibernate.type=TRACE log4j.logger.org.hibernate.SQL=DEBUG 在你的日志管理器中（我使用log4j）。

- hibernate.format_sql：布尔标志，将使Hibernate将SQL格式化美化后输出到标准输出。
- hibernate.dialect（未显示，理由充分）：许多旧教程会教你如何设置Hibernate方言，以便它与数据库通信。Hibernate可以根据你使用的JDBC驱动自动检测使用哪种方言。由于Oracle有大约3种不同的方言，MySQL有5种不同的方言，我建议将此决定交给Hibernate。有关Hibernate支持的完整方言列表，请参见这里。

你需要声明的最后两个bean是：

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

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

PersistenceExceptionTranslator将数据库特定的HibernateException或SQLException转换为应用上下文可理解的Spring异常。

TransactionManager bean负责控制事务以及回滚。

注意：你应该将SessionFactory bean自动装配到你的DAO中。

第1.2节：使用XML的简单Hibernate示例

要使用XML配置设置一个简单的Hibernate项目，您需要3个文件：hibernate.cfg.xml、每个实体的POJO，以及每个实体对应的EntityName.hbm.xml。以下是使用MySQL的示例：

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>

    <!-- XML映射文件列表 -->
```


log4j.logger.org.hibernate.type=TRACE log4j.logger.org.hibernate.SQL=DEBUG in your log manager (I use log4j).

- *hibernate.format_sql*: Boolean flag, will cause Hibernate to pretty print your SQL to stdout.
- *hibernate.dialect* (Not shown, for good reason): A lot of old tutorials out there show you how to set the Hibernate dialect that it will use to communicate to your database. Hibernate **can** auto-detect which dialect to use based on the JDBC driver that you are using. Since there are about 3 different Oracle dialects and 5 different MySQL dialects, I'd leave this decision up to Hibernate. For a full list of dialects Hibernate supports see here.

The last 2 beans you need to declare are:

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

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

The `PersistenceExceptionTranslator` translates database specific `HibernateException` or `SQLExceptions` into Spring exceptions that can be understood by the application context.

The `TransactionManager` bean is what controls the transactions as well as roll-backs.

Note: You should be autowiring your `SessionFactory` bean into your DAO's.

Section 1.2: Simple Hibernate example using XML

To set up a simple hibernate project using XML for the configurations you need 3 files, hibernate.cfg.xml, a POJO for each entity, and a EntityName.hbm.xml for each entity. Here is an example of each using 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 和 testPassword 都会被替换。确保如果资源在包中，使用完整的资源名称。

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">
该类包含员工信息。
      </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"/>
```



```
   <mapping resource="HibernatePractice/Employee.hbm.xml"/>

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

DBSchemaName, testUserName, and testPassword would all be replaced. Make sure to use the full resource name if it is in a package.

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>
```

如果类在某个包中，请使用完整的类名 packageName.className。

当你拥有这三个文件后，就可以在项目中使用 Hibernate 了。

第1.3节：无XML的Hibernate配置

此示例取自 here

```
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);

        /*
* 配置连接池
         */

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");


        /**
* 资源映射
```



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

Again, if the class is in a package use the full class name packageName.className.

After you have these three files you are ready to use hibernate in your project.

Section 1.3: XML-less Hibernate configuration

This example has been taken from here

```
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();
    }


}
```

请注意，使用最新的Hibernate时，这种方法效果不佳（Hibernate 5.2版本仍允许此配置）


```
         */

//        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();
    }


}
```

Please note, that with latest Hibernate this approach doesn't work well (Hibernate 5.2 release still allow this configuration)

第二章：Hibernate中的获取

获取在JPA（Java持久化API）中非常重要。在JPA中，HQL（Hibernate查询语言）和JPQL（Java持久化查询语言）用于根据实体之间的关系获取实体。虽然这比使用大量连接查询和子查询通过原生SQL获取所需数据要好得多，但我们在JPA中获取关联实体的策略仍然本质上影响着应用程序的性能。

第2.1节：建议使用FetchType.LAZY。仅在需要时联接获取列

下面是一个映射到employer表的Employer实体类。如你所见，我使用了fetch = FetchType.LAZY而不是fetch = FetchType.EAGER。我使用LAZY的原因是，Employer以后可能有很多属性，而我不一定每次都需要知道Employer的所有字段，因此加载所有字段会导致加载Employer时性能变差。

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

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

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

        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;
        }
    }
```

但是，对于 LAZY（延迟）加载的关联，未初始化的代理有时会导致 LazyInitializationException。在这种情况下，我们可以简单地在 HQL/JPQL 中使用 JOIN FETCH 来避免 LazyInitializationException。

Chapter 2: Fetching in Hibernate

Fetching is really important in JPA (Java Persistence API). In JPA, HQL (Hibernate Query Language) and JPQL (Java Persistence Query Language) are used to fetch the entities based on their relationships. Although it is way better than using so many joining queries and sub-queries to get what we want by using native SQL, the strategy how we fetch the associated entities in JPA are still essentially effecting the performance of our application.

Section 2.1: It is recommended to use FetchType.LAZY. Join fetch the columns when they are needed

Below is an Employer entity class which is mapped to the table employer. As you can see I used **fetch = FetchType.LAZY** instead of fetch = FetchType.EAGER. The reason I am using LAZY is because Employer may have a lot of properties later on and every time I may not need to know all the fields of an Employer, so loading all of them will leading a bad performance then an employer is loaded.

```
@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;
        }
    }
```

However, for LAZY fetched associations, uninitialized proxies are sometimes leads to LazyInitializationException. In this case, we can simply use JOIN FETCH in the HQL/JPQL to avoid 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
```



```
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
```

第3章：使用注解的 Hibernate 实体关系

注释	详情
`@OneToOne`	指定与对应对象的一对一关系。
@OneToMany	指定映射到多个对象的单个对象。
@ManyToOne	指定映射到单个对象的对象集合。
`@Entity`	指定映射到数据库表的对象。
`@Table`	指定该对象映射到哪个数据库表。
@JoinColumn	指定外键存储在哪一列。
@JoinTable	指定存储外键的中间表。

第3.1节：使用用户管理的连接表对象实现双向多对多

```
@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;

    //你可以在此表中存储其他对象/字段。
}
```

指定了许多Foo对象与许多Bar对象之间的双向关系，使用用户管理的中间连接表。

Foo对象作为行存储在名为FOO的表中。Bar对象作为行存储在名为BAR的表中。Foo和Bar对象之间的关系存储在名为FOO_BAR的表中。应用程序中包含一个FooBar对象。

Chapter 3: Hibernate Entity Relationships using Annotations

Annotation	Details
`@OneToOne`	Specifies a one to one relationship with a corresponding object.
`@OneToMany`	Specifies a single object that maps to many objects.
`@ManyToOne`	Specifies a collection of objects that map to a single object.
`@Entity`	Specifies an object that maps to a database table.
`@Table`	Specifies which database table this object maps too.
`@JoinColumn`	Specifies which column a foregin key is stored in.
`@JoinTable`	Specifies an intermediate table that stores foreign keys.

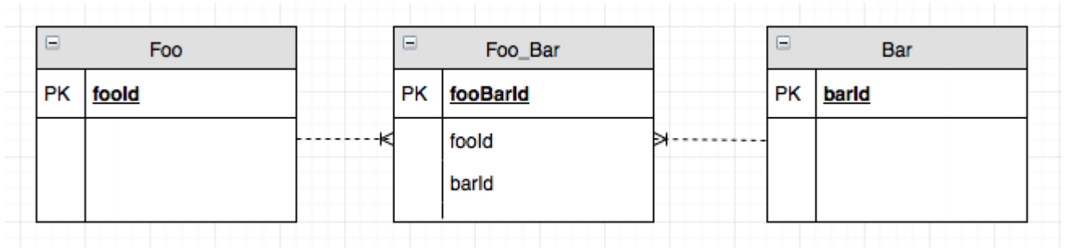
Section 3.1: Bi-Directional Many to Many using user managed join table object

```
@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.
}
```

Specifies a two-way relationship between many `Foo` objects to many `Bar` objects using an intermediate join table that the user manages.

The `Foo` objects are stored as rows in a table called `FOO`. The `Bar` objects are stored as rows in a table called `BAR`. The relationships between `Foo` and `Bar` objects are stored in a table called `FOO_BAR`. There is a `FooBar` object as part of the application.

通常用于当你想在连接对象上存储额外信息时，例如关系创建的日期。

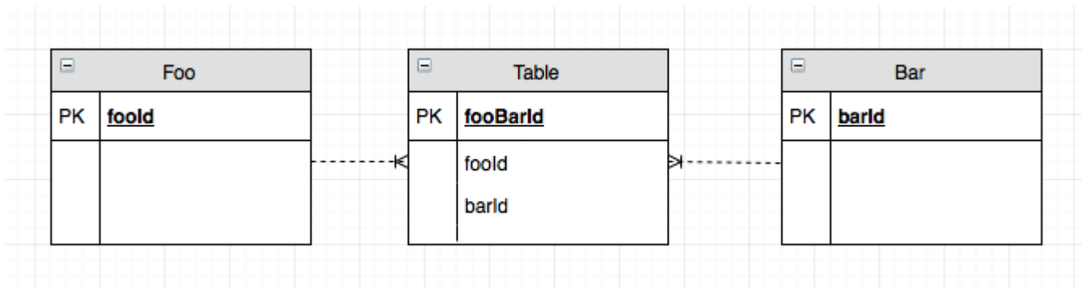


第3.2节：使用Hibernate管理的连接表的双向多对多关系

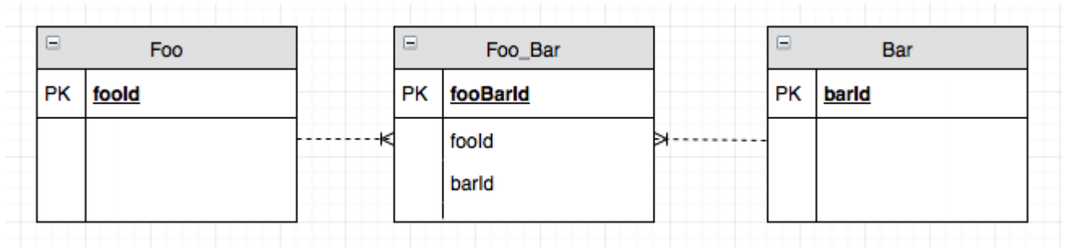
```
@Entity
@Table(name="FOO")
public class Foo {
    private UUID fooId;

    @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;
}
```

指定了使用Hibernate管理的中间连接表，在多个Foo对象和多个Bar对象之间的关系。

Foo对象存储为名为FOO的表中的行。Bar对象存储为名为BAR的表中的行。Foo和Bar对象之间的关系存储在名为FOO_BAR的表中。然而，这意味着应用程序中没有FooBar对象。



Commonly used when you want to store extra information on the join object such as the date the relationship was created.

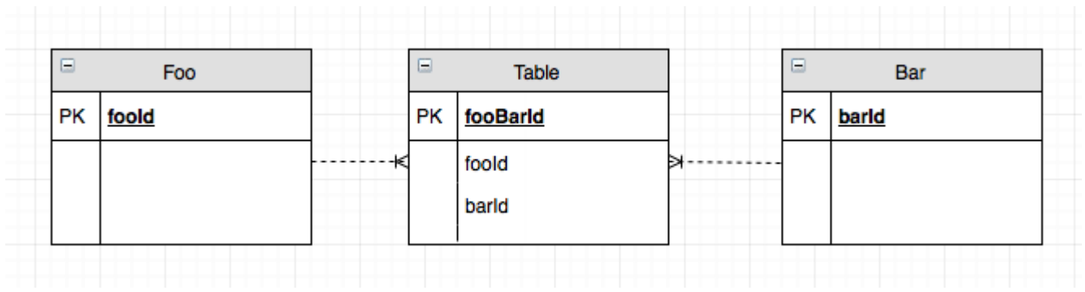


Section 3.2: Bi-Directional Many to Many using Hibernate managed join table

```
@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;
}
```

Specifies a relationship between many `Foo` objects to many `Bar` objects using an intermediate join table that Hibernate manages.

The `Foo` objects are stored as rows in a table called `FOO`. The `Bar` objects are stored as rows in a table called `BAR`. The relationships between `Foo` and `Bar` objects are stored in a table called `FOO_BAR`. However this implies that there is no `FooBar` object as part of the application.

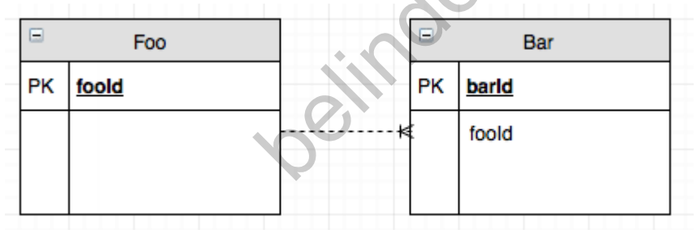


第3.3节：使用外键映射的双向一对多关系

```
@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;
}
```

指定了使用外键的一个Foo对象与多个Bar对象之间的双向关系。

Foo对象存储为名为FOO的表中的行。Bar对象存储为名为BAR的表中的行。外键存储在BAR表中名为fooId的列中。



第3.4节：由Foo.class管理的双向一对一关系

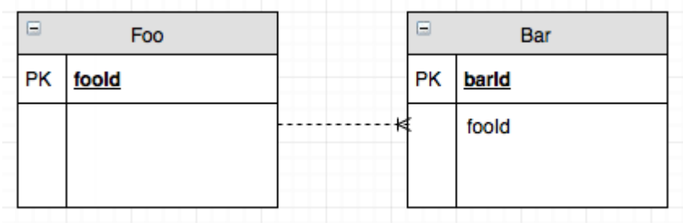
```
@Entity
@Table(name="FOO")
public class Foo {
    private UUID fooId;

@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;
}
```


Section 3.3: Bi-directional One to Many Relationship using foreign key mapping

```
@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;
}
```

Specifies a two-way relationship between one Foo object to many Bar objects using a foreign key.

The Foo objects are stored as rows in a table called FOO. The Bar objects are stored as rows in a table called BAR. The foreign key is stored on the BAR table in a column called fooId.



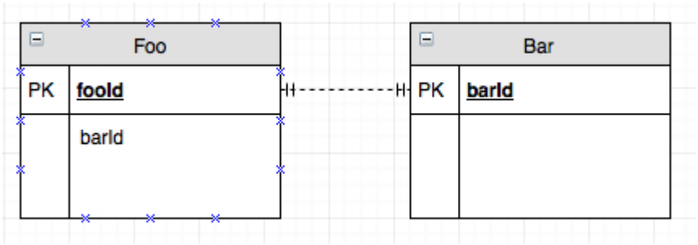
Section 3.4: Bi-Directional One to One Relationship managed by Foo.class

```
@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;
}
```

指定一个双向关系，使用外键将一个Foo对象关联到一个Bar对象。

Foo对象存储为名为FOO的表中的行。Bar对象存储为名为BAR的表中的行。
外键存储在FOO表中，列名为barId。

注意，mappedBy的值是对象上的字段名，而不是列名。



第3.5节：使用用户管理的连接表的单向一对多关系

```
@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;

    //此处未指定映射。
}

@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;

    //你可以在此表中存储其他对象/字段。
}
```

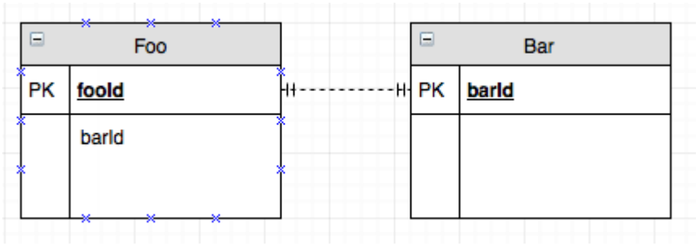
指定一个单向关系，使用用户管理的中间连接表，将一个Foo对象关联到多个Bar对象。

这类似于ManyToMany关系，但如果你在目标外键上添加unique约束，则可以

Specifies a two-way relationship between one Foo object to one Bar object using a foreign key.

The Foo objects are stored as rows in a table called FOO. The Bar objects are stored as rows in a table called BAR. The foreign key is stored on the FOO table in a column called barId.

Note that the mappedBy value is the field name on the object, not the column name.



Section 3.5: Uni-Directional One to Many Relationship using user managed join table

```
@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.
}
```

Specifies a one-way relationship between one Foo object to many Bar objects using an intermediate join table that the user manages.

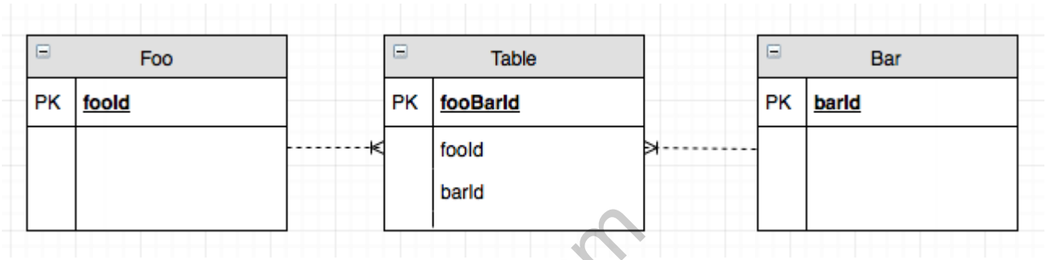
This is similar to a ManyToMany relationship, but if you add a unique constraint to the target foreign key you can

强制其为OneToMany关系。

Foo对象作为行存储在名为FOO的表中。Bar对象作为行存储在名为BAR的表中。Foo和Bar对象之间的关系存储在名为FOO_BAR的表中。应用程序中包含一个FooBar对象。

注意没有将Bar对象映射回Foo对象。可以自由操作Bar对象而不影响Foo对象。

在使用Spring Security设置拥有一组可执行Role的User对象时非常常用。你可以向用户添加或移除角色，而无需担心级联删除Role。



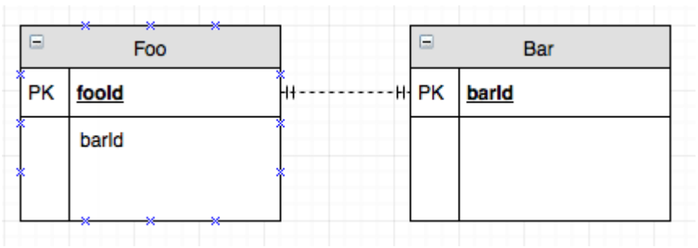
第3.6节：单向一对一关系

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

    @OneToOne
    private Bar bar;
}

@Entity
@Table(name="BAR")
public class Bar {
    private UUID barId;
    //没有对应映射到 Foo.class
}
```

指定一个 Foo 对象到一个 Bar 对象的单向关系。

Foo 对象存储为名为 FOO 的表中的行。Bar 对象存储为名为 BAR 的表中的行。

注意没有将Bar对象映射回Foo对象。可以自由操作Bar对象而不影响Foo对象。

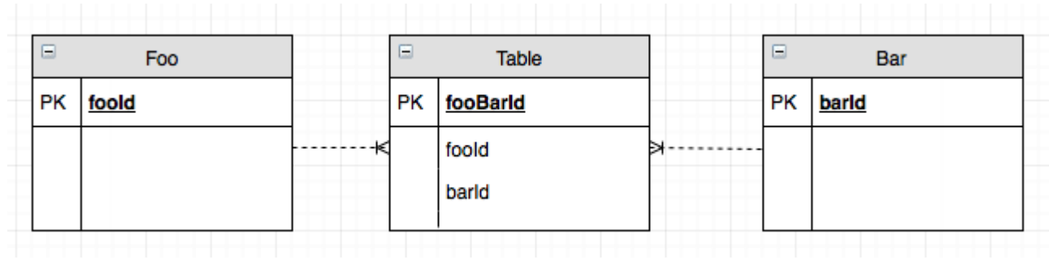


enforce that it is `OneToMany`.

The `Foo` objects are stored as rows in a table called `FOO`. The `Bar` objects are stored as rows in a table called `BAR`. The relationships between `Foo` and `Bar` objects are stored in a table called `FOO_BAR`. There is a `FooBar` object as part of the application.

Notice that there is no mapping of `Bar` objects back to `Foo` objects. `Bar` objects can be manipulated freely without affecting `Foo` objects.

Very commonly used with Spring Security when setting up a `User` object who has a list of `Role`'s that they can perform. You can add and remove roles to a user without having to worry about cascades deleting `Role`'s.



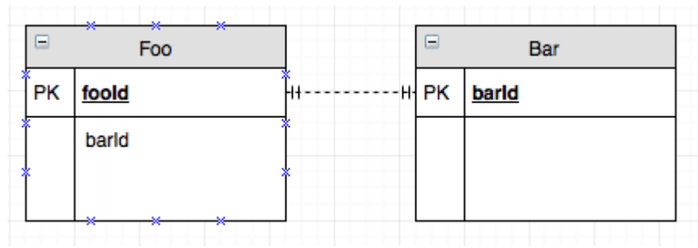
Section 3.6: Uni-directional One to One Relationship

```
@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
}
```

Specifies a one-way relationship between one `Foo` object to one `Bar` object.

The `Foo` objects are stored as rows in a table called `FOO`. The `Bar` objects are stored as rows in a table called `BAR`.

Notice that there is no mapping of `Bar` objects back to `Foo` objects. `Bar` objects can be manipulated freely without affecting `Foo` objects.



第4章：HQL

HQL 是 Hibernate 查询语言，它基于 SQL，幕后会被转换成 SQL，但语法不同。你使用实体/类名而不是表名，使用字段名而不是列名。它还允许许多简写。

第4.1节：选择整个表

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

第4.2节：选择特定列

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

第4.3节：包含 Where 子句

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

第4.4节：连接

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


Chapter 4: HQL

HQL is Hibernate Query Language, it based on SQL and behind the scenes it is changed into SQL but the syntax is different. You use entity/class names not table names and field names not column names. It also allows many shorthands.

Section 4.1: Selecting a whole table

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

Section 4.2: Select specific columns

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

Section 4.3: Include a Where clause

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

Section 4.4: Join

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

第5章：原生SQL查询

第5.1节：简单查询

假设你已经掌握了Hibernate的Session对象，这里命名为session：

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

这将检索 some_table中的所有行，并将它们放入result变量中，打印每个值。

第5.2节：获取唯一结果的示例

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

使用此查询，当您知道查询结果始终唯一时，您将获得唯一结果。

如果查询返回多个值，您将收到异常

> org.hibernate.NonUniqueResultException

您也可以在此链接中查看详细信息 here，附有更多描述

所以，请确保您知道查询将返回唯一结果

Chapter 5: Native SQL Queries

Section 5.1: Simple Query

Assuming you have a handle on the Hibernate `Session` object, in this case named `session`:

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

This will retrieve all rows in `some_table` and place them into the `result` variable and print every value.

Section 5.2: Example to get a unique result

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

with this query, you get a unique result when you know the result of the query is always going to be unique.

And if the query returns more than one value, you will get an exception

> org.hibernate.NonUniqueResultException

You also check the details in this link here with more discription

So, please be sure that you know the query will return unique result

第6章：映射关联

第6.1节：一对一Hibernate映射

每个国家有一个首都。每个首都有一个国家。

Country.java

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

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

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

@Column(name = "national_language")
    private String 官方语言;

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

    //构造函数

    //getter和setter方法

   }
```

Capital.java

```
包 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;
```


Chapter 6: Mapping associations

Section 6.1: One to One Hibernate Mapping

Every Country has one Capital. Every Capital has one Country.

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;

    //构造函数

    //getter和setter方法

}
```

HibernateDemo.java

```
包 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();
  }

}
```



```
    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();
  }

}
```

第7章：条件和投影

第7.1节：使用过滤器

`@Filter` 用作 `WHERE` 条件，这里有一些示例

学生实体

```
@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 */
}
```

Study 实体

```
@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 方法 */
}
```

StudentStudy 实体

```
@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 */
```


Chapter 7: Criterias and Projections

Section 7.1: Use Filters

`@Filter` is used as a `WHERE` camp, here some examples

Student Entity

```
@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 */
}
```

Study Entity

```
@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 */
```

```
}
```

这样，每次启用“active”过滤器时，

-我们对学生实体进行的每个查询都会返回所有状态=true的学生学习记录

-我们对学习实体进行的每个查询都会返回**所有**状态=**true**的学习记录

-我们对StudentStudy实体进行的每个查询将只返回状态= true的学习关系

请注意，study_id是SQL StudentStudy表中的字段名

第7.2节：使用限制条件的列表

假设我们有一个TravelReview表，城市名称作为列“title”

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

我们可以通过链式调用向criteria添加限制条件，如下所示：

```
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();
```

第7.3节：使用投影

如果我们只想检索少数几列，可以使用 Projections 类来实现。例如，以下代码检索标题列

```
 // 选择所有标题列
  List review = session.createCriteria(TravelReview.class)
        .setProjection(Projections.property("title"))
    .list();
  // 获取行数
review = session.createCriteria(TravelReview.class)
    .setProjection(Projections.rowCount())
.list();
  // 获取标题数量
review = session.createCriteria(TravelReview.class)
    .setProjection(Projections.count("title"))
.list();
```



```
}
```

This way, every time the "active" filter is enabled,

-Every query we do on the student entity will return **ALL** Students with **ONLY** their `state = true` studies

-Every query we do on the Study entity will return **ALL** `state = true` studies

-Every query we do on the StudentStudy entiy will return **ONLY** the ones with a `state = true` Study relationship

Pls note that study_id is the name of the field on the sql StudentStudy table

Section 7.2: List using Restrictions

Assuming we have a TravelReview table with City names as column "title"

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

We can add restrictions to the criteria by chaining them as follows:

```
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();
```

Section 7.3: Using Projections

Should we wish to retrieve only a few columns, we can use the Projections class to do so. For example, the following code retrieves the title column

```
// 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();
```

第8章：自定义命名策略

第8.1节：创建和使用自定义隐式命名策略

创建自定义的ImplicitNamingStrategy可以让您调整Hibernate如何为未显式命名的Entity属性分配名称，包括外键、唯一键、标识列、基本列等。

例如，默认情况下，Hibernate 会生成哈希的外键，类似于：

```
FKe6hidh4u0qh8y1ijy59s2ee6m
```

虽然这通常不是问题，但您可能希望名称更具描述性，例如：

```
FK_asset_tenant
```

这可以通过自定义ImplicitNamingStrategy轻松实现。

此示例扩展了ImplicitNamingStrategyJpaCompliantImpl，但如果您愿意，也可以选择实现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());
     }

}
```

要告诉 Hibernate 使用哪个ImplicitNamingStrategy，请在您的persistence.xml或hibernate.cfg.xml文件中定义hibernate.implicit_naming_strategy属性，如下所示：

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

或者，您也可以在hibernate.properties文件中指定该属性，如下所示：

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

在此示例中，所有未显式定义name的外键现在将从CustomNamingStrategy获取名称。

第8.2节：自定义物理命名策略

在将实体映射到数据库表名时，我们依赖于@Table注解。但如果我们对数据库表名有命名约定，可以实现自定义物理命名策略，告诉hibernate基于实体名称计算表名，而无需通过

@Table注解显式声明这些名称。属性和列的映射也是同理。

Chapter 8: Custom Naming Strategy

Section 8.1: Creating and Using a Custom ImplicitNamingStrategy

Creating a custom `ImplicitNamingStrategy` allows you to tweak how Hibernate will assign names to non-explicitly named `Entity` attributes, including Foreign Keys, Unique Keys, Identifier Columns, Basic Columns, and more.

For example, by default, Hibernate will generate Foreign Keys which are hashed and look similar to:

```
FKe6hidh4u0qh8y1ijy59s2ee6m
```

While this is often not an issue, you may wish that the name was more descriptive, such as:

```
FK_asset_tenant
```

This can easily be done with a custom `ImplicitNamingStrategy`.

This example extends the `ImplicitNamingStrategyJpaCompliantImpl`, however you may choose to implement `ImplicitNamingStrategy` if you wish.

```
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());
    }

}
```

To tell Hibernate which `ImplicitNamingStrategy` to use, define the `hibernate.implicit_naming_strategy` property in your `persistence.xml` or `hibernate.cfg.xml` file as below:

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

Or you can specify the property in `hibernate.properties` file as below:

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

In this example, all Foreign Keys which do not have an explicitly defined `name` will now get their name from the `CustomNamingStrategy`.

Section 8.2: Custom Physical Naming Strategy

When mapping our entities to database table names we rely on a `@Table` annotation. But if we have a naming convention for our database table names, we can implement a custom physical naming strategy in order to tell hibernate to calculate table names based on the names of the entities, without explicitly stating those names with `@Table` annotation. Same goes for attributes and columns mapping.

例如，我们的实体名称是：

```
ApplicationEventLog
```

我们的表名是：

```
application_event_log
```

我们的物理命名策略需要将实体名称从驼峰命名转换为数据库表名的蛇形命名。
我们可以通过扩展 Hibernate 的 `PhysicalNamingStrategyStandardImpl` 来实现这一点：

```
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 标识符 toPhysicalColumnName(标识符 name, Jdbc环境 context) {
        return new Identifier(addUnderscores(name.getText()), name.isQuoted());
    }

    protected static 字符串 addUnderscores(字符串 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);
    }
}
```

我们重写了方法 toPhysicalTableName 和 toPhysicalColumnName 的默认行为，以应用我们的数据库命名规范。

为了使用我们的自定义实现，我们需要定义属性 hibernate.physical_naming_strategy 并赋值为我们的 PhysicalNamingStrategyImpl 类的名称。

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

这样我们就可以免去代码中 @Table 和 @Column 注解的使用，因此我们的实体类：

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


For example, our entity name is:

```
ApplicationEventLog
```

And our table name is:

```
application_event_log
```

Our Physical naming strategy needs to convert from entity names that are camel case to our db table names which are snake case. We can achieve this by extending 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);
    }
}
```

We are overriding default behavior of methods `toPhysicalTableName` and `toPhysicalColumnName` to apply our db naming convention.

In order to use our custom implementation we need to define `hibernate.physical_naming_strategy` property and give it the name of our `PhysicalNamingStrategyImpl` class.

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

This way we can alleviate our code from `@Table` and `@Column` annotations, so our entity class:

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

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

将正确映射到数据库表：

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

如上例所示，如果出于某种原因数据库对象的名称不符合我们的通用命名规范，我们仍然可以显式声明数据库对象的名称：@Column(name="finish_dtl")


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

will be correctly be mapped to db table:

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

As seen in the example above, we can still explicitly state the name of the db object if it is not, for some reason, in accordance with our general naming convention: `@Column(name="finish_dtl")`

第9章：缓存

第9.1节：在WildFly中启用Hibernate缓存

要在WildFly中为Hibernate启用二级缓存，请在您的persistence.xml文件中添加此属性：

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

您还可以通过此属性启用查询缓存：

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

启用Hibernate的二级缓存时，WildFly不要求您定义缓存提供者，因为默认使用的是Infinispan。如果您想使用其他缓存提供者，可以通过hibernate.cache.provider_class属性来实现。

Chapter 9: Caching

Section 9.1: Enabling Hibernate Caching in WildFly

To enable Second Level Caching for Hibernate in WildFly, add this property to your `persistence.xml` file:

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

You may also enable Query Caching with this property:

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

WildFly does not require you to define a Cache Provider when enabling Hibernate's Second-Level Cache, as Infinispan is used by default. If you would like to use an alternative Cache Provider, however, you may do so with the `hibernate.cache.provider_class` property.

第10章：实体之间的关联映射

第10.1节：使用XML的一对多关联

这是一个使用XML进行一对多映射的示例。我们将以作者和书籍为例，假设一个作者可能写了多本书，但每本书只有一个作者。

作者类：

```
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;
    }
}
```

书籍类：

```
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;
```


Chapter 10: Association Mappings between Entities

Section 10.1: One to many association using XML

This is an example of how you would do a one to many mapping using XML. We will use Author and Book as our example and assume an author may have written many books, but each book will only have one author.

Author class:

```
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;
    }
}
```

Book class:

```
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">
        该类包含作者的信息。
      </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">
        该类包含书籍信息。
      </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"/>
```



```
    }
    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>
```

一对多连接的特点是 Book 类包含一个 Author，并且 xml 中有 <many-to-one> 标签。cascade 属性允许你设置子实体如何保存/更新。

第10.2节：一对多关联

为了说明一对多关系，我们需要两个实体，例如国家（Country）和城市（City）。一个国家拥有多个城市。

在下面的 CountryEntity 中，我们为国家定义了一组城市。

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

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

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

@OneToMany(mappedBy="country", fetch=FetchType.LAZY)
   private Set<城市实体> 城市集合 = new HashSet<>();

   //未显示Getter和Setter方法
}
```

现在是城市实体。

```
@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
}
```



```
        <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>
```

What makes the one to many connection is that the Book class contains an Author and the xml has the <many-to-one> tag. The cascade attribute allows you to set how the child entity will be saved/updated.

Section 10.2: OneToMany association

To illustrate relation OneToMany we need 2 Entities e.g. Country and City. One Country has multiple Cities.

In the CountryEntity beloww we define set of cities for Country.

```
@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
}
```

Now the city entity.

```
@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
}
```

第11章：延迟加载与急切加载

第11.1节：延迟加载与急切加载

数据的获取或加载主要可以分为两种类型：急切加载和延迟加载。

为了使用Hibernate，请确保将其最新版本添加到pom.xml文件的依赖部分：

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

1. 急加载和懒加载

我们首先要讨论的是懒加载和急加载的定义：

急加载是一种设计模式，其中数据初始化会立即发生。也就是说，集合会在其父对象被获取时被完全加载（立即获取）。

懒加载是一种设计模式，用于推迟对象的初始化，直到真正需要该对象时才进行初始化。这可以有效提升应用程序的性能。

2. 使用不同类型的加载

可以通过以下XML参数启用懒加载：

```
lazy="true"
```

让我们深入看一个例子。首先是一个用户类：

```
public class User implements Serializable {

    private Long 用户ID;
    private String 用户名;
    private String 名字;
    private String 姓氏;
    private Set<订单详情> 订单详情 = new HashSet<>();

    // 设置器和获取器
    // equals 和 hashcode
    }
```

看看我们拥有的订单详情集合。现在让我们看看订单详情类：

```
public class 订单详情 implements Serializable {

    private Long 订单ID;
    private Date 订单日期;
    private String 订单描述;
    private 用户 user;

    // 设置器和获取器
    // equals 和 hashcode
```


Chapter 11: Lazy Loading vs Eager Loading

Section 11.1: Lazy Loading vs Eager Loading

Fetching or loading data can be primarily classified into two types: eager and lazy.

In order to use Hibernate make sure you add the latest version of it to the dependencies section of your pom.xml file:

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

1. Eager Loading And Lazy Loading

The first thing that we should discuss here is what lazy loading and eager loading are:

Eager Loading is a design pattern in which data initialization occurs on the spot. It means that collections are fetched fully at the time their parent is fetched (fetch immediately)

Lazy Loading is a design pattern which is used to defer initialization of an object until the point at which it is needed. This can effectively contribute to application's performance.

2. Using The Different Types Of Loading

Lazy loading can be enabled using the following XML parameter:

```
lazy="true"
```

Let's delve into the example. First we have a User class:

```
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
    }
```

Look at the Set of orderDetail that we have. Now let's have a look at the **OrderDetail class**:

```
public class OrderDetail implements Serializable {

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

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

```
}
```

设置UserLazy.hbm.xml中延迟加载的重要部分：

```
<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>
```

这就是启用延迟加载的方式。要禁用延迟加载，我们只需使用：`lazy = "false"`，这样将启用急加载。以下是在另一个文件 User.hbm.xml 中设置急加载的示例：

```
<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>
```

第11.2节：范围

对于那些没有使用过这两种设计的人来说，延迟加载和急加载的范围是在特定的Session内SessionFactory中。急加载会立即加载所有内容，意味着不需要调用任何东西来获取它。但延迟加载通常需要某些操作来检索映射的集合/对象。有时在session之外进行延迟加载会有问题。例如，你有一个视图显示某些映射POJO的详细信息。

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

    // 这里是 getter 和 setter
}

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

    // 这里是 getter 和 setter
}

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();

        // 这里出现了懒加载问题
user.getLikedPage();
    }
```



```
}
```

The important part that is involved in setting the lazy loading in the `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>
```

This is how the lazy loading is enabled. To disable lazy loading we can simply use: `lazy = "false"` and this in turn will enable eager loading. The following is the example of setting up eager loading in another 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>
```

Section 11.2: Scope

For those who haven't played with these two designs, the scope of lazy and eager is within a specific **Session** of *SessionFactory*. *Eager* loads everything instantly, means there is no need to call anything for fetching it. But lazy fetch usually demands some action to retrieve mapped collection/object. This sometimes is problematic getting lazy fetch outside the *session*. For instance, you have a view which shows the detail of the some mapped 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();
    }
```

```
}
```

当你尝试在会话（session）之外获取懒加载（lazy fetched）时，会抛出懒初始化异常（lazyinitializeException）。这是因为默认情况下，所有一对多（oneToMany）或其他关系的获取策略都是懒加载（lazy）（按需调用数据库），而当你关闭会话后，就无法与数据库通信。因此，当我们的代码尝试获取likedPage集合时，会抛出异常，因为没有关联的会话来渲染数据库。

解决方案是使用：

1. 视图中打开会话（Open Session in View）- 即使在渲染视图时也保持会话打开。
2. 在关闭会话之前调用 Hibernate.initialize(user.getLikedPage()) - 这告诉 Hibernate 初始化集合元素


```
}
```

When you will try to get **lazy fetched** outside the *session* you will get the *lazyinitializeException*. This is because by default fetch strategy for all oneToMany or any other relation is *lazy*(call to DB on demand) and when you have closed the session, you have no power to communicate with database. so our code tries to fetch collection of *likedPage* and it throws exception because there is no associated session for rendering DB.

Solution for this is to use:

1. Open Session in View - In which you keep the session open even on the rendered view.
2. `Hibernate.initialize(user.getLikedPage())` before closing session - This tells hibernate to initialize the collection elements

第12章：启用/禁用SQL日志

第12.1节：使用日志配置文件

在你选择的日志配置文件中，将以下包的日志级别设置为如下所示：

```
# 记录 SQL 语句
org.hibernate.SQL=DEBUG
# 记录参数
org.hibernate.type=TRACE
```

可能需要一些特定于日志记录器的前缀。

Log4j 配置：

```
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"/>
```

第12.2节：使用Hibernate属性

这将显示生成的SQL，但不会显示查询中包含的值。

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate4.LocalSessionFactoryBean">
    <property name="hibernateProperties">
        <props>
<!-- 显示不带参数的 SQL -->
            <prop key="hibernate.show_sql">true</prop>
            <!-- 格式化 SQL -->
            <prop key="hibernate.format_sql">true</prop>
            <!-- 以注释形式显示 HQL -->
            <prop key="use_sql_comments">true</prop>
        </props>
    </property>
</bean>
```

第12.3节：启用/禁用调试中的 SQL 日志

一些使用 Hibernate 的应用程序在启动时会生成大量 SQL。有时在调试时最好在特定点启用或禁用 SQL 日志。

要启用，只需在调试应用程序时在 IDE 中运行以下代码：

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


Chapter 12: Enable/Disable SQL log

Section 12.1: Using a logging config file

In the logging configuration file of your choice set the logging of the following packages to the levels shown.:

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

There will probably be some logger specific prefixes that are required.

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"/>
```

Section 12.2: Using Hibernate properties

This will show you the generated SQL, but will not show you the values contained within the queries.

```
<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>
```

Section 12.3: Enable/Disable SQL log in debug

Some applications that use Hibernate generate a huge amount of SQL when the application is started. Sometimes it's better to enable/disable the SQL log in specific points when debugging.

To enable, just run this code in your IDE when you are debugging the aplication:

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

```
.setLevel(org.apache.log4j.Level.DEBUG)
```

要禁用：

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



```
    .setLevel(org.apache.log4j.Level.DEBUG)
```

To disable:

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

第13章：Hibernate与JPA

第13.1节：Hibernate与JPA的关系

Hibernate是JPA标准的一个实现。因此，那里所说的一切也适用于Hibernate。

Hibernate对JPA有一些扩展。此外，设置JPA提供者的方式是特定于提供者的。本节文档应仅包含Hibernate特有的内容。

Chapter 13: Hibernate and JPA

Section 13.1: Relationship between Hibernate and JPA

Hibernate is an implementation of the JPA standard. As such, everything said there is also true for Hibernate.

Hibernate has some extensions to JPA. Also, the way to set up a JPA provider is provider-specific. This documentation section should only contain what is specific to Hibernate.

第14章：性能调优

第14.1节：使用组合代替继承

Hibernate有一些继承策略。JOINED继承类型会对子实体和父实体进行JOIN操作。

这种方法的问题在于Hibernate总是会带来继承中所有相关表的数据。

例如，如果你有实体Bicycle和MountainBike，且使用JOINED继承类型：

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

}
```

并且：

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

}
```

任何针对MountainBike的JPQL查询都会带出Bicycle的数据，生成类似如下的SQL查询：

```
SELECT mb.*, b.* FROM MountainBike mb JOIN Bicycle b ON b.id = mb.id WHERE ...
```

如果Bicycle还有另一个父类（例如Transport），上述查询也会带出该父类的数据，执行额外的JOIN操作。

如你所见，这也是一种EAGER映射。你无法仅使用此继承策略只获取MountainBike表的数据。

为了性能最佳，建议使用组合而非继承。

为此，你可以将MountainBike实体映射为包含一个字段bicycle：

```
@Entity
public class MountainBike {

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

}
```

和自行车：

```
@Entity
public class Bicycle {

}
```

现在每个查询默认只会带来MountainBike数据。

Chapter 14: Performance tuning

Section 14.1: Use composition instead of inheritance

Hibernate has some strategies of inheritance. The `JOINED` inheritance type do a JOIN between the child entity and parent entity.

The problem with this approach is that Hibernate **always** bring the data of all involved tables in the inheritance.

Per example, if you have the entities `Bicycle` and `MountainBike` using the `JOINED` inheritance type:

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

}
```

And:

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

}
```

Any JPQL query that hit `MountainBike` will brings the `Bicycle` data, creating a SQL query like:

```
SELECT mb.*, b.* FROM MountainBike mb JOIN Bicycle b ON b.id = mb.id WHERE ...
```

If you have another parent for `Bicycle` (like `Transport`, per example), this above query will brings the data from this parent too, doing an extra JOIN.

As you can see, this is a kind of `EAGER` mapping too. You don't have the choice to bring only the data of the `MountainBike` table using this inheritance strategy.

The best for performance is use composition instead of inheritance.

To accomplish this, you can mapping the `MountainBike` entity to have a field `bicycle`:

```
@Entity
public class MountainBike {

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

}
```

And `Bicycle`:

```
@Entity
public class Bicycle {

}
```

Every query now will bring only the `MountainBike` data by default.

致谢

非常感谢所有来自Stack Overflow Documentation的人员帮助提供此内容，
更多更改可发送至web@petercv.com以发布或更新新内容

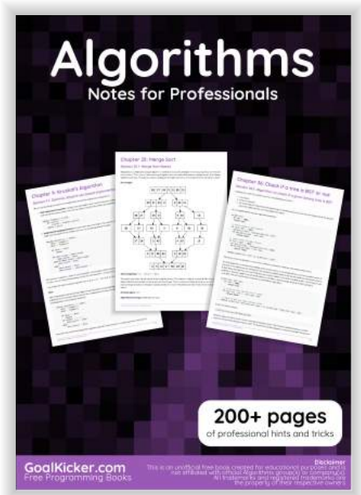
阿列克谢·洛吉诺夫	第3章
BELLIL	第11章
丹尼尔·凯弗	第5章和第12章
德里克	第12章和第14章
詹姆斯ENL	第1章、第3章和第12章
迈克尔·皮费尔	第12章和第13章
米奇·塔尔马奇	第8章和第9章
纳雷什·库马尔	第1章和第8章
内森尼尔·福特	第5章
omkar sirra	第6章
普里塔姆·班纳吉	第11章
重生	第1章
rObOtAndChalie	第2章
赛弗	第7章
萨米尔·斯里瓦斯塔瓦	第7章
桑迪普·卡马斯	第5章
斯坦尼斯拉夫L	第10章
用户7491506	第1、4和10章
veljkost	第8章
维基	第11章

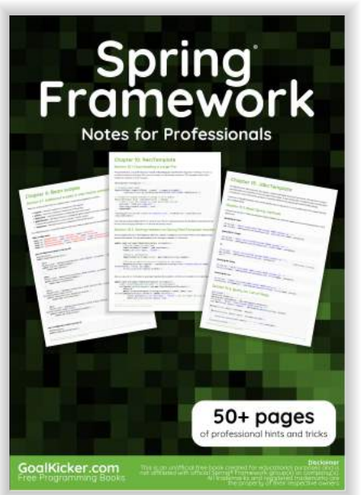
Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content,
more changes can be sent to web@petercv.com for new content to be published or updated

Aleksei Loginov	Chapter 3
BELLIL	Chapter 11
Daniel Käfer	Chapters 5 and 12
Dherik	Chapters 12 and 14
JamesENL	Chapters 1, 3 and 12
Michael Piefel	Chapters 12 and 13
Mitch Talmadge	Chapters 8 and 9
Naresh Kumar	Chapters 1 and 8
Nathaniel Ford	Chapter 5
omkar sirra	Chapter 6
Pritam Banerjee	Chapter 11
Reborn	Chapter 1
rObOtAndChalie	Chapter 2
Saifer	Chapter 7
Sameer Srivastava	Chapter 7
Sandeep Kamath	Chapter 5
StanislavL	Chapter 10
user7491506	Chapters 1, 4 and 10
veljkost	Chapter 8
vicky	Chapter 11

你可能也喜欢





You may also like