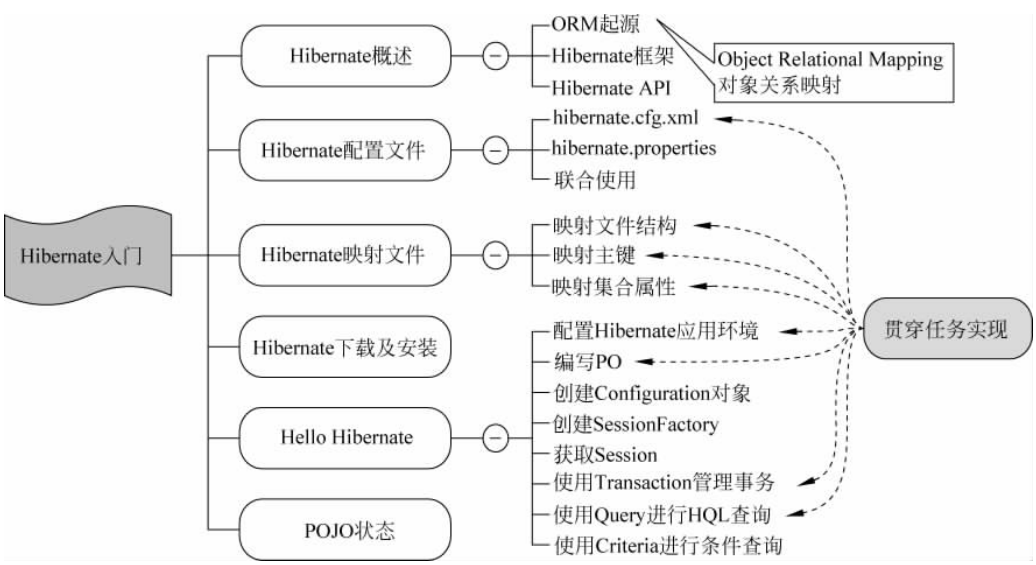
第5章 Hibernate入门

任务驱动

本章任务是实现“GIFT-EMS 礼记”系统的登录和注册功能，并完成客户端 JS 校验功能：

- 【任务 5-1】 升级任务 3-3 登录功能，并完成客户端的 JS 校验功能。
- 【任务 5-2】 实现用户注册功能，并完成客户端的 JS 校验功能。
- 【任务 5-3】 实现登录验证功能。

学习路线



本章目标

知　识　点	Listen(听)	Know(懂)	Do(做)	Revise(复习)	Master(精通)
Hibernate 概述	★	★	★		
Hibernate 下载及安装	★	★	★		
Hello Hibernate	★	★	★	★	★
Hibernate 配置文件详解	★	★	★	★	
Hibernate 映射文件详解	★	★	★	★	
POJO 持久化对象	★	★	★	★	

5.1 Hibernate 概述

Hibernate 是一个开放源码的 ORM(Object Relational Mapping,对象关系映射)框架,能够很好地解决对象与关系型数据之间的映射问题,并对 JDBC 进行最大限度的对象封装,使得程序员可以通过面向对象编程思维来操作数据库。在企业级开发应用中,Hibernate 由于其功能完备、性能优越、开源等特点已被广泛使用。

5.1.1 ORM 起源

现今流行的主流的开发语言,如 Java、C#等,都是面向对象的编程语言,对象都存在内存中,无法永久保存数据,因此要将对象永久保存就需要进行对象的持久化操作,即将对象存储到数据库中;然而目前广泛应用的主流数据库仍是关系型数据库产品,如 Oracle、SQL Server 等,依然是关系型数据库,关系型数据库中存放的是关系型数据而非对象数据。面向对象的编程语言和底层数据库的发展不协调,便带来对象与关系型数据之间映射问题,由此催生了 ORM 框架。

ORM 框架是面向对象编程语言和关系型数据库之间的桥梁。可以将 ORM 理解成一种规范,这种规范概述了此类框架的基本特征:完成面向对象的编程语言到关系型数据库的映射。当 ORM 框架完成映射后,既可以利用面向对象程序设计语言的简单易用性,又可利用关系数据库的技术优势。

ORM 框架的优势有以下几点:

- 通过面向对象的编程思想对数据库进行访问;
- 简单易用,提高开发效率,最大限度地降低代码的编写工作;
- 降低访问数据库的频率,提高应用程序性能;
- 相对独立,变动时不会影响上层的实现。

为了将针对关系型数据的操作转换成对象操作,ORM 框架需要实现关系数据到对象的映射,这种映射关系通常使用配置文件来完成。所有的 ORM 工具都遵循相同的映射思路,ORM 基本映射规律有以下几条:

- 表与类映射。数据库中的表被映射到一个持久化类上,当对该持久化类进行操作时,系统会自动转换为对数据库中的表进行添加、删除、修改等操作。受 ORM 管理的持久化类其实只是一个普通的 Java 类,也称为 POJO(Plain Ordinary Java Object)。
- 表中的行与对象(持久化类的实例)映射。持久化类会生成很多实例对象,每个实例对象都与表中的一行记录相对应。当修改某个持久化类的实例时,ORM 工具会自动转换成对相应数据库表中的特定行进行操作。
- 表中的列(字段)与对象的属性映射。当修改持久化对象的属性值时,ORM 工具会自动转换成对相应数据库表中的指定行、指定列的操作。

如图 5-1 所示,ORM 工具将数据库中的表映射到类,表中行映射到对象,行中的列映射成对象的属性。表中的每一条记录都对应一个持久化对象,表与表之间的关系也映射成对象与对象之间的关系。

如图 5-2 所示,数据库中有一个 Students 表,它与 Student 类相互映射,表中的列名(字

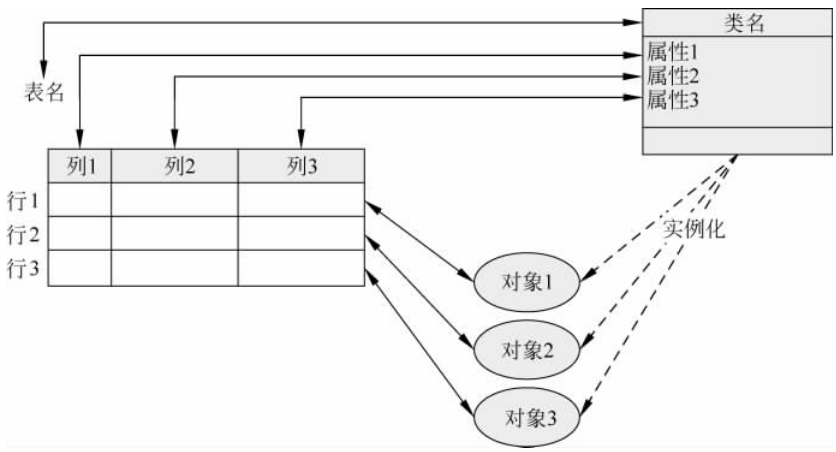


图 5-1　ORM 映射规律

段)id、name、score 与 Student 类中的 3 个同名属性相映射；objStudent 是 Student 类的一个实例对象，与 Students 表中的第 4 条记录信息相映射。

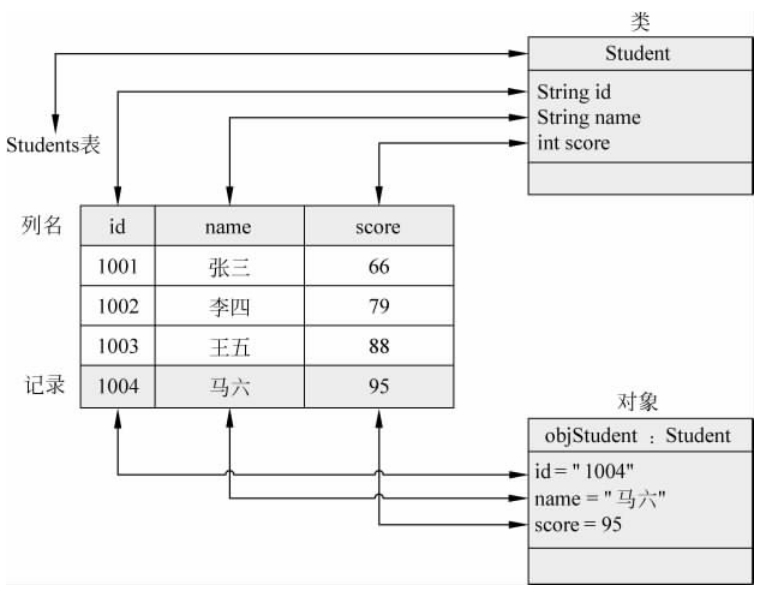


图 5-2　ORM 映射示意图

目前 ORM 框架的产品非常多，比较流行的有以下几个产品：

- JPA——JPA 本身只是一种 ORM 规范，并不是 ORM 产品。JPA 实体与 Hibernate 的 POJO 十分相似，甚至 JPA 实体完全可作为 Hibernate 的 POJO 类使用。相对于其他开源 ORM 框架，JPA 的最大优势在于是官方公布的 Java EE 规范标准，具有通用性。
- Hibernate——是目前流行的 ORM 框架，已经被选作 JBoss 的持久层解决方案，而 JBoss 又加入了 Red Hat 组织，因此 Hibernate 属于 Red Hat 组织的一部分。Hibernate 设计灵巧、性能优秀、文档丰富是其风靡全球的重要原因。
- iBATIS——Apache 软件基金组织的子项目，是一种 SQL Mapping 框架。曾经在 J2EE 开发中扮演非常重要的角色，但因为并不支持纯粹的面向对象的操作，因此现在

逐渐开始被取代，但在一些公司，依然占有一席之地。特别是一些对数据访问特别灵活的地方，iBATIS更加灵活，允许开发人员直接编写SQL语句。

- TopLink——是Oracle公司的产品，早年单独作为ORM框架使用时一直没有赢得广泛的市场，现在主要作为JPA实现。GlassFish服务器的JPA实现就是TopLink。

5.1.2 Hibernate框架

Hibernate框架是Java语言下的ORM解决方案，是一个“面向对象-关系数据库”的映射工具，用来将对象模型映射到基于SQL的关系模型数据结构中。Hibernate不仅管理对象数据到数据库表的映射（包括Java数据类型到SQL数据类型的映射），还提供数据查询和获取数据的方法，与单纯使用JDBC相比，Hibernate简单易用，大幅节省了数据持久化处理操作的开发时间。

与其他ORM框架对比，Hibernate能在众多的ORM框架中脱颖而出，是因为有如下几个方面的优势：

- 开源、免费，便于研究源代码，或修改源代码进行功能定制等；
- 轻量级封装，避免引入过多复杂问题，易调试；
- 具有可扩展性，API开放，根据研发需要可以自行扩展；
- 性能稳定，具有保障。

Hibernate的持久化解决方案将开发者从复杂JDBC访问中释放出来，用户无须关注底层的JDBC操作，而是以面向对象的方式进行持久层操作。底层数据连接的获得、数据访问的实现、事务控制都无须开发者关心。这是一种“全面解决”的体系结构方案，将应用层从底层的JDBC/JTA API中抽象出来。通过配置文件来管理底层的JDBC连接，让Hibernate解决持久化访问的实现。这种“全面解决”方案的体系结构图如图5-3所示。

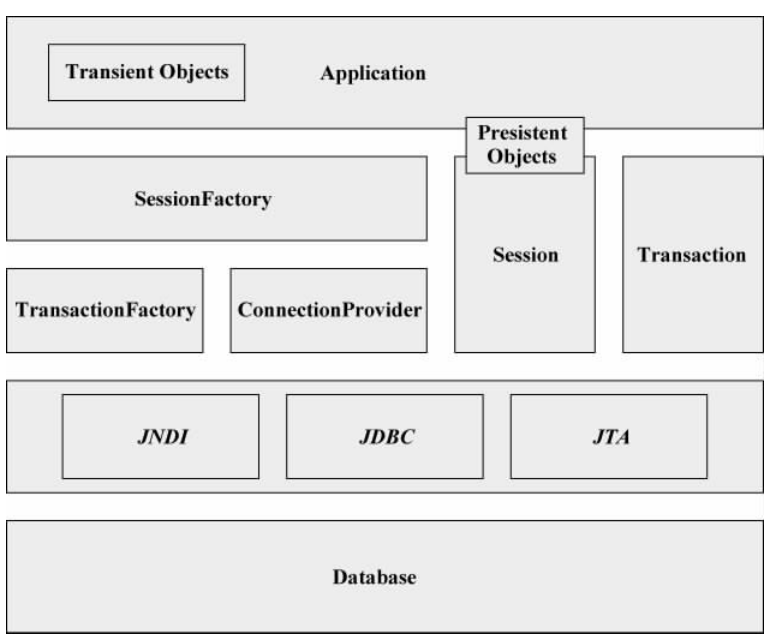


图5-3 Hibernate全面解决方案体系结构图

Hibernate 体系结构中各对象的主要功能如表 5-1 所示。

表 5-1　Hibernate 中的主要对象

对　象　名	功 能 描 述
SessionFactory	是 Hibernate 的关键对象，是单个数据库映射关系经过编译后的内存镜像，也是线程安全的。SessionFactory 是生成 Session 的工厂，本身要应用到 ConnectionProvider，该对象可以在进程和集群的级别上，为那些在事务之间可以重用的数据提供可选的二级缓存
Session	是应用程序和持久存储层之间交互操作的一个单线程对象，是 Hibernate 持久化操作的关键对象，所有的持久化对象必须在 Session 的管理下才能够进行持久化操作。此对象的生存周期很短，其隐藏了 JDBC 连接，也是 Transaction 的工厂。Session 对象有一个一级缓存，现实执行 Flush 之前，所有的持久化操作的数据都在缓存中 Session 对象处
Transaction	提供持久化中的原子操作，具有数据库事务的概念。代表一次原子操作，具有数据库事务的概念。但通过抽象，将应用程序从底层的具体的 JDBC、JTA 和 CORBA 事务中隔离开。在某些情况下，一个 Session 之内可能包含多个 Transaction 对象。虽然事务操作是可选的，但是所有的持久化操作都应该在事务管理下进行，即使是只读操作
Persistent Object	持久化对象，与 Session 关联，处于持久化状态。系统创建的 POJO 实例一旦与特定 Session 关联，并对应数据表的指定记录，那该对象就处于持久化状态，这一系列的对象都被称为持久化对象。程序中对持久化对象的修改，都将自动转换为持久层的修改。持久化对象完全可以是普通的 Java Beans/POJO，唯一的特殊性是它们正与 Session 关联着
Transient Object	瞬态对象，没有与 Session 关联，尚未持久化的对象。系统进行 new 关键字进行创建的 Java 实例，没有 Session 相关联，此时处于瞬态。瞬态实例可能是在被应用程序实例化后，尚未进行持久化的对象。如果一个曾经持久化过的实例，但因为 Session 的关闭而转换为脱管状态
ConnectionProvider	数据库连接提供者，用于生成与数据库建立连接的 JDBC 对象。是生成 JDBC 的连接的工厂，同时具备连接池的作用。他通过抽象将底层的 DataSource 和 DriverManager 隔离开。这个对象无须应用程序直接访问，仅在应用程序需要扩展时使用
TransactionFactory	是生成 Transaction 对象的工厂，实现了对事务的封装。是生成 Transaction 对象实例的工厂。该对象也无须应用程序的直接访问

5.1.3　Hibernate API

应用程序可以通过 Hibernate API 访问操作数据库。Hibernate API 中的接口可以分为以下几类：

- 提供访问数据的操作（如保存、更新、删除和查询）的接口，这些接口包括 Session、Transaction 和 Query 接口；
- 用于配置 Hibernate 的接口，如 Configuration 接口；
- 使应用程序接受 Hibernate 内部发生事件的回调接口，这些接口包括 Interceptor、Lifecycle 和 Validatable 接口；
- 用于扩展 Hibernate 功能的接口，如 UserType、CompositeUserType 和 IdentifierGenerator 接口。

Hibernate 内部封装了 JDBC、JTA（Java Transaction API）和 JNDI（Java Naming and

Directory Interface)。JDBC 提供底层的数据访问操作,只要用户提供了相应的 JDBC 驱动程序,Hibernate 就可以访问任何一个数据库系统。

本书不详细介绍 Hibernate API 的所有用法,只侧重介绍 Hibernate 常用的 5 个核心接口,所有的 Hibernate 应用都会访问这 5 个核心接口。

- Configuration 接口:配置和启动 Hibernate,创建 SessionFactory 对象,Hibernate 应用通过 Configuration 实例获得对象-关系映射文件中的元数据,以及动态配置 Hibernate 的属性,然后创建 sessionFactory 实例;
- SessionFactory 接口:初始化 Hibernate,充当数据存储源的代理,创建 Session 对象;
- Session 接口:也称为持久化管理器,提供了持久化相关的操作,如保存、更新、删除、加载和查询对象;
- Transaction 接口:管理事务;
- Query 和 Criteria 接口:执行数据库查询,Query 接口用于执行 HQL 数据库查询,而 Criteria 接口用于 QBC 检索方式。

注意

根据版本不同,Hibernate API 所属的包也不同,Hibernate3.6 的 API 在 org.hibernate 包中。

Hibernate 的 5 个核心接口的类框图如图 5-4 所示。

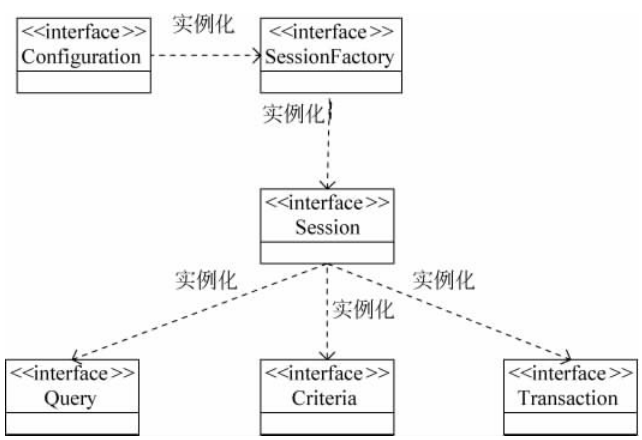


图 5-4 Hibernate 的核心接口的类框图

5.2 持久化对象

Hibernate 应用开发中通过持久化对象作为媒介,对数据库按照面向对象的方式进行操作。即应用程序无须直接访问数据库,只需创建、修改或删除持久化对象,Hibernate 则会负责将这些操作转换成相应的对数据库表的操作。如图 5-5 所示为 Hibernate 简要的体系结构,在该体系结构中,Hibernate 需要两个文件:

- Hibernate 配置文件,该文件用于配置 Hibernate 和数据库之间的连接信息;

- Hibernate 映射文件，该文件用于确定之久类和数据表、数据列之间的对应关系。

Hibernate 采用低侵入式设计持久化类，这种设计对持久化类不进行任何要求，完全使用 POJO (Plain Old Java Object，普通传统的 Java 对象)作为持久化对象。虽然 Hibernate 对 POJO 没有明确要求，但程序员在编写 POJO 时约定俗成地需要遵守如下几个规则：

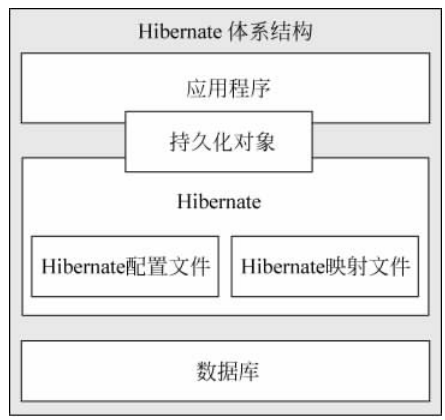


图 5-5　Hibernate 简要体系结构

- 持久化类需要实现 Serializable 接口，使持久化对象可序列化，便于数据传递；
- 持久化类不能是最终类，即非 final 类；
- 持久化类需要提供一个无参数的构造方法(默认构造方法)；
- 持久化类需要提供一个标识属性，通常映射到数据库表中的主键；
- 每个属性都提供相应的 setter 和 getter 方法。

以 Student 学生类为例，来演示 Hibernate 中的持久化对象的创建，代码如下所示。

【代码 5-1】　Student.java

```
package com.qst.chapter05.pojos;

public class Student {
    //属性
    private String id;
    private String name;
    private int score;

    //属性的 getter 和 setter 方法
    public String getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

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

    public int getScore() {
        return score;
    }

    public void setScore(int score) {
```

```
        this.score = score;
    }

}
```

从上述代码中可以看出，POJO 跟普通的 JavaBean 一样。Hibernate 直接采用 POJO 作为 PO，不需要持久化类继承任何父类，或者实现任何接口，以低侵入方式保证了代码的简单性、独立性和可重用性。

为使 POJO 具备可持久化操作的能力，Hibernate 采用 XML 格式的文件来制定 POJO 类和数据库表的映射。在程序运行时，Hibernate 根据这个映射文件生成各种 SQL 语句。映射文件的扩展名为 hbm.xml，需要强调的是，映射文件应和其对应的持久化类放在同一目录中。通过配置上文中 Student 类的映射文件来演示 POJO 对应的映射文件示例如下。

【代码 5-2】 Student.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="com.qst.chapter05.pojos.Student" table="STUDENTS">
        <id name="id" column="ID">
            <generator class="uuid.hex" />
        </id>
        <property name="name" column="NAME" type="string" not-null="true" />
        <property name="score" column="SCORE" type="java.lang.Integer"
            not-null="true" />
    </class>
</hibernate-mapping>
```

上述配置文件中，<hibernate-mapping>元素是 Hibernate 映射文件的根元素，<class>元素描述类和表之间的映射，这样每个<class>元素将映射成一个 PO，即：

```
PO = POJO + 映射文件
```

通过 Hibernate 映射文件中的配置信息，清晰地表达了持久化类和数据库表之间的对应关系。

5.3 Hibernate 配置文件

Hibernate 配置文件用于配置访问数据库的一些参数，如连接数据库的 URL 连接字符串、用户名、密码以及是否创建或更新表等信息，这些信息对于所有持久化类都是通用的。

要使用 Hibernate 配置文件，可以采用如下 3 种形式：

- hibernate.cfg.xml 文件形式，该文件采用 XML 文件形式，是 Hibernate 最常用的配置方式；
- hibernate.properties 文件形式，采用“键/值”对的属性文件形式，能够快速配置

Hibernate 的信息；

- hibernate. cfg. xml 和 hibernate. properties 结合使用形式，联合各自的优势，一起作为配置文件。

在 hibernate. properties 和 hibernate. cfg. xml 文件中都可以对 Hibernate 的属性信息进行配置，其中 Hibernate 常用的配置属性如表 5-2 所示。

表 5-2　Hibernate 配置文件常用属性

属　　性	功 能 描 述
hibernate. dialect	针对不同的数据库提供不同的方言类，允许 Hibernate 针对特定的数据库生成优化的 SQL 语句
hibernate. connection. driver_class	数据库驱动类
hibernate. connection. datasource	数据源的 JNDI 名字
hibernate. connection. url	JNDI 数据库提供者的 URL
hibernate. connection. username	连接数据库的用户名
hibernate. connection. password	连接数据库的密码
hibernate. connection. pool_size	数据库连接池的最大容量
hibernate. show_sql	是否输出 Hibernate 操作数据库使用的 SQL 语句
hibernate. format_sql	是否格式化输出的 SQL 语句
hibernate. hbm2ddl. auto	是否根据映射文件自动建立数据库表，该属性可以是 create、create-drop 和 update 三个值：值为 create 时会根据 POJO 创建表，但每次运行都要重新生成表；值为 create-drop 时，则关闭 sessionFactory 时，自动删除创建的表；update 是最常用的属性，不会删除以前的行记录

Hibernate 可以连接多种不同的数据库，因不同的数据库在支持标准 SQL 的基础上增加一些特有的扩展，所以连接不同的数据库需要使用不同的数据库方言。在 Hibernate 配置文件中，通过设置 hibernate. dialect 属性值来指定访问不同数据库的方言类，不同数据库所对应的不同数据库方言类如表 5-3 所示。

表 5-3　数据库方言类

数　据　库	数据方言类
Oracle 9i/10g/11g	org. hibernate. dialect. OracleDialect
MySQL	org. hibernate. dialect. MySQLDialect
DB2	org. hibernate. dialect. DB2Dialect
Microsoft SQL Server	org. hibernate. dialect. SQLServerDialect
Sybase	org. hibernate. dialect. SybaseDialect

5.3.1　hibernate. cfg. xml

hibernate. cfg. xml 文件是 Hibernate 配置文件常用的形式，因为 XML 文件的结构性强、容易读取以及配置灵活等优势，且可以直接配置 POJO 所对应的映射文件。

在 hibernate. cfg. xml 文件中配置 Hibernate 属性时，可以省略 hibernate 前缀，例如，配置 hibernate. dialect 属性可以直接简化成 dialect。

【示例】 在 hibernate.cfg.xml 中配置属性

```
<property name = "dialect">
    org.hibernate.dialect.OracleDialect
```

</property>hibernate.cfg.xml 配置文件通常放在 Java 类文件的根目录，即 src 根目录下，如图 5-6 所示。

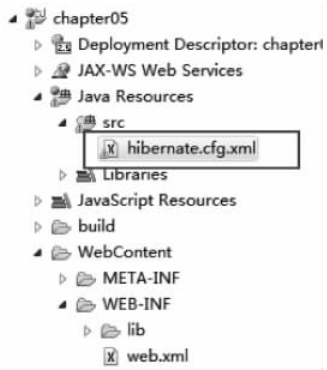


图 5-6 hibernate.cfg.xml 配置文件的位置

【代码 5-3】 hibernate.cfg.xml

```
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- 配置访问 Oracle 数据库参数信息 -->
        <property name = "dialect">
            org.hibernate.dialect.OracleDialect
        </property>
        <property name = "connection.driver_class">
            oracle.jdbc.driver.OracleDriver
        </property>
        <property name = "connection.url">
            jdbc:oracle:thin:@localhost:1521:orcl
        </property>
        <property name = "connection.username"> scott </property>
        <property name = "connection.password"> scott123 </property>
        <!-- 在控制台显示 SQL 语句 -->
        <property name = "show_sql"> true </property>
        <!-- 根据需要自动生成、更新数据表 -->
        <property name = "hbm2ddl.auto"> update </property>
        <!-- 注册所有 ORM 映射文件 -->
        <mapping resource = "com/qst/chapter05/pojos/Student.hbm.xml" />

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

Hibernate 配置文件中有如下几个元素：

- <hibernate-configuration>元素，Hibernate 配置文件的根元素；
- <session-factory>元素，是<hibernate-configuration>根元素的子元素；
- <property>元素，是<session-factory>元素的子元素，用于配置访问数据库参数信息，可以有多个<property>子元素；
- <mapping>元素，是<session-factory>元素的子元素，用于注册所有 ORM 映射文件，每个<mapping>子元素注册一个 Hibernate 映射文件，可以有多个<mapping>子元素。

5.3.2 hibernate.properties

hibernate.properties 文件是以“key=value”对形式出现的，虽然 hibernate.properties 和 hibernate.cfg.xml 这两个配置文件的格式不同，但其本质完全一样，只是在 hibernate.properties 文件中不能对 Hibernate 的映射文件进行配置。

下述代码使用 hibernate.properties 文件配置 Hibernate 相关信息。

【示例】 在 hibernate.properties 中配置属性

```
#Oracle 方言
hibernate.dialect = org.hibernate.dialect.OracleDialect
#Oracle 驱动
hibernate.connection.driver_class = oracle.jdbc.driver.OracleDriver
#连接数据库的 URL
hibernate.connection.url = jdbc:oracle:thin:@localhost:1521:orcl
#连接数据库的用户名
hibernate.connection.username = scott
#连接数据库的密码
hibernate.connection.password = scott123
#显示 Sql
hibernate.show_sql = true
#格式化 Sql
hibernate.format_sql = true
#自动创建表
hibernate.hbm2ddl.auto = update
```

hibernate.properties 文件也是放在类文件的根目录(与 hibernate.cfg.xml 同目录)。

因为 hibernate.properties 文件没有对映射文件进行配置，在 Hibernate 应用程序中必须调用 Configuration 对象的 addResource()方法添加映射文件，示例代码如下所示。

【示例】 添加映射文件

```
//实例化 Configuration
Configuration configuration = new Configuration();
//添加映射文件
configuration.addResource("com/qst/chapter05/pojos/Student.hbm.xml");
```

5.3.3 联合使用

hibernate.cfg.xml 和 hibernate.properties 各自具有自己的优势，可以将 hibernate.properties 和 hibernate.cfg.xml 这两种文件结合使用，例如，在 hibernate.properties 文件中只对数据库的相关配置信息进行配置，而在 hibernate.cfg.xml 文件中只对 Hibernate 映射文

件进行配置，示例代码如下所示。

【示例】 在 hibernate.cfg.xml 中配置映射文件

```
<?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 >
        <! -- 映射文件的配置 -- >
        < mapping resource = "com/qst/chapter05/pojos/Student.hbm.xml"/>
    </session - factory >
</hibernate - configuration >
```

5.4 Hibernate 映射文件

5.4.1 映射文件结构

Hibernate 映射文件的根元素是＜hibernate-mapping＞，该元素下可以有多个＜class＞子元素，每个＜class＞元素对应一个 POJO 持久化类的映射，将类和表之间的关系进行映射。

【示例】 映射文件结构

```
< hibernate - mapping 属性 = "值">
    < class name = "类名" table = "表名">
        <! -- 主键 -- >
        < id name = "主键名" column = "主键列">
            <! -- 主键生成器 -- >
            < generator class = "生成策略" />
        </id >
        <! -- 属性列表 -- >
        < property name = "属性名" column = "列名" type = "数据类型"/>
        ...
    </class >
...
</hibernate - mapping >
```

其中＜hibernate-mapping＞元素的可选属性如表 5-4 所示。

表 5-4 ＜hibernate-mapping＞元素的可选属性

属 性	功 能 描 述
auto-import	是否允许在查询语言中使用非全限定的类名，默认值为 true
catalog	所映射数据库的 Catalog 名
default-cascade	Hibernate 默认的级联风格，默认值为 none
default-access	默认属性访问策略，默认值为 property
default-lazy	默认延迟加载策略，默认值为 true
auto-import	是否允许使用非全限定的类名，默认值为 true
package	指定包名，对于映射文件中非全限定的类名，默认在该包下
schema	映射数据库的 schema 名

＜class＞元素常用的可选属性如表 5-5 所示。

表 5-5 ＜class＞元素常用的可选属性

属　性	功 能 描 述
name	持久化类的类名
table	持久化类映射的表名
discriminator-value	区分不同子类的值
mutable	指定持久化类的实例是否可变，默认值为 true
proxy	延迟装载时的代理，可以是该类自己的名字

5.4.2 映射主键

Hibernate 中持久化类会有一个标识属性，用于标识唯一的持久化实例，而标识属性则要映射到底层的主键。在 Hibernate 映射文件中，标识属性是通过＜id＞元素来指定，＜id＞元素常用的可选属性如表 5-6 所示。

表 5-6 ＜id＞元素的可选属性

属　性	功 能 描 述
name	标识属性名
type	标识属性的数据类型，该类型既可以是 Hibernate 内建类型，也可以是 Java 类型(带包名)
column	标识属性所映射的数据库中表的列名
unsaved-value	指定刚创建、未保存的某个实例的标识属性值
access	指定访问标识属性的访问策略，默认是 property

通常情况下，推荐在 Hibernate 中使用逻辑主键，尽量避免使用复杂的物理主键，使用物理主键会增加数据库维护的复杂度。Hibernate 为逻辑主键提供了主键生成器，以便为每个持久化实例生成唯一的逻辑主键值。在 Hibernate 中内置了多种主键生成器，如表 5-7 所示。

表 5-7 主键生成器列表

类　型　名	功 能 描 述
increment	获取数据库表中所有主键中的最大值，在最大值基础上加 1，为最新记录的主键
identity	自动增长。MS SQL Server、MySQL 和 DB2 等数据库中可以设置表的某个字段(列)的数值自动增长。此种方式生成主键的数据类型可以是 long、short、int 及其对应的封装类的类型
sequence	序列。Oracle、DB2 等数据库可以创建一个序列，然后从序列中获取当前序号作为主键值
hilo	“高/低位”高效算法产生主键值。此种方式生成主键的数据类型可以是 long、short、int 及其对应的封装类的类型
seqhilo	与 hilo 类似，但使用指定的 sequence 获取高位值
uuid	采用 128 位的 UUID 算法生成一个字符串类型的主键
guid	采用 GUID 字符串产生主键值
native	由 Hibernate 根据所使用的数据库支持能力从 identity、sequence 或者 hilo 中选择一种，例如，Oracle 中使用 sequence，MySQL 中使用 identity
assigned	指派值
foreign	通过的关联持久化对象为主键赋值

下述代码采用 native 生成主键。

【示例】 指定主键生成器

```
<id name = "id" column = "ID">
    <generator class = "native" />
</id>
```

在选择 Hibernate 的主键生成器策略时，可参考以下两个原则：

- 如果应用系统不需要分布式部署，在数据库支持的情况下使用 sequence、identity、hilo、seqhilo 和 uuid；
- 如果应用需要使用多个数据库或者进行分布式的部署，则 uuid 是最佳的选择。

5.4.3 映射集合属性

集合属性也是常见的属性，例如，一个部门有多个员工，那么员工就是部门类中的集合属性。集合属性是现实中非常普遍的属性关系。

集合属性大致有两种：

- 第一种是单纯的集合属性，例如，List、Set 或数组；
- 另外一种是 Map 结构的集合属性，每个属性都是“键/值”对。

Hibernate 要求集合属性必须声明为接口，例如 List、Set、Map 接口等。Hibernate 之所以要求使用集合接口声明集合属性，是因为当程序持久化某个实例时，Hibernate 会自动把程序中的集合实现类替换成 Hibernate 自己的集合实现类。对于不同的集合接口，在 Hibernate 映射文件中需要采用不同的集合映射元素。不同的集合具有不同的特性，选择正确的集合映射元素在实际开发中非常重要。表 5-8 列出了 Hibernate 中的集合映射元素、对应的集合属性及其特性。

表 5-8 Hibernate 集合映射元素

集合映射元素	集 合 属 性	特 征
<list>	java.util.List	集合中的元素可以重复，可以通过索引存取元素
<set>	java.util.Set	集合中的元素不重复
<map>	java.util.Map	集合中的元素是以键/值对形式存放
<array>	数组	可以是对象数组或基本数据类型的数组
<primitive-array>	基本数据类型的数组	基本数据类型的数组，例如，int[]、char[]等
<bag>	java.util.Collection	无序集合
<idbag>	java.util.Collection	无序集合，但可以为集合增加逻辑次序

集合属性其实是 Hibernate 关联关系的体现，该属性对应另一个数据表中的数据。有关集合映射元素的应用参见第 6 章内容。

5.5 Hibernate 下载及安装

登录 Hibernate 官方网站并进入下载页面，下载 Hibernate 最新产品，下载网址是“http://hibernate.org/orm/downloads”，如图 5-7 所示，下载 Hibernate 4.3.8.Final 版。

图 5-7　下载 Hibernate 发布版

将下载的压缩包解压，解压缩得到一个名为 hibernate-release-4.3.8.Final 的文件夹，该文件夹下的文件目录如图 5-8 所示。

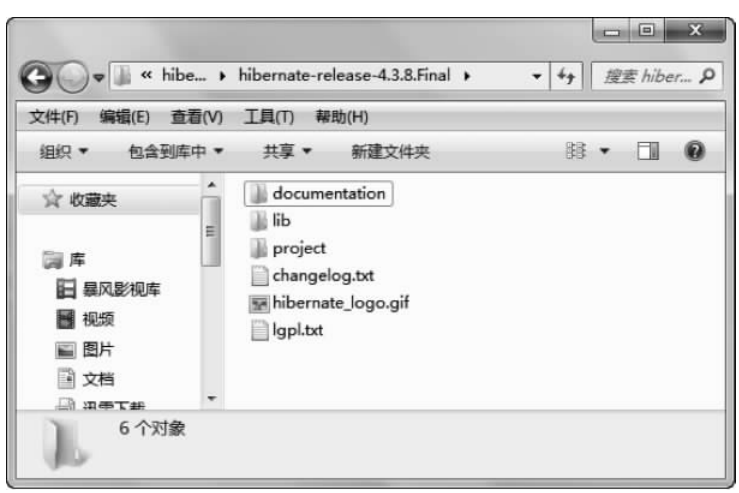


图 5-8　Hibernate 压缩包中的文件目录

其中：

- documentation 目录下存放了 Hibernate 的相关文档，包括 Hibernate 的参考文档和 API 等；

- lib 目录下存放了 Hibernate 编译和运行所依赖的第三方类库(即 jar 文件);
- project 目录下存放了 Hibernate 各种相关项目的源代码。

将 lib 目录下的 required 和 jpa 子目录中的所有 jar 包添加到应用的类加载路径中,便可以在应用程序中使用 Hibernate 框架的功能。

5.6 Hello Hibernate

开发 Hibernate 应用程序的方式有以下 3 种:

- 第一种方式,自顶向下从持久化类到数据库表,即先编写持久化类,再编写映射文件,进而生成数据库表结构。
- 第二种方式,自底向上从数据库表到持久化类,即根据数据库表结构生成对应的映射文件和持久化类。
- 第三种方式,从中间出发向上与向下同时发展,即先编写映射文件,然后根据映射文件向上生成持久化类,向下生成数据库表结构。

注意

本书采用第一种自顶向下的开发方式,该方式符合面向对象的编程思路。由于 Hibernate 底层依然是基于 JDBC 的,因此在应用程序中使用 Hibernate 执行持久化时一定少不了 JDBC 驱动。本书底层采用 Oracle 数据库,因此还需要将 Oracle 数据库的驱动 jar 文件添加到应用的类加载路径中。

自顶向下开发 Hibernate 应用程序的步骤如下:

(1) 配置 Hibernate 应用环境,在应用中添加 Hibernate 所需的 jar 包,并创建 Hibernate 配置文件;

(2) 编写 PO;

(3) 创建 Configuration 对象;

(4) 创建 SessionFactory 对象;

(5) 获取 Session 对象;

(6) 使用 Transaction 管理事务;

(7) 使用 Query 进行 HQL 查询或利用 Criteria 实现条件查询。

下述内容通过 Hello Hibernate 示例来演示如何在 Eclipse 中开发 Hibernate 应用程序的详细过程。

5.6.1 配置 Hibernate 应用环境

使用 Hibernate 框架前,首先需要将下载的 Hibernate 压缩包中 lib 目录下的 required、jpa 子目录中的所有 jar 包添加到应用的类加载路径中。如果是 Web 应用程序,则只需将 Hibernate 所必需的核心 jar 文件复制到 WEB-INF/lib 目录下,如图 5-9 所示。这些 jar 文件都是必需的,其中 hibernate-core-4.3.8.Final.jar 文件是 Hibernate 的核心类库文件,其他文件是 Hibernate 框架本身需要引用的 jar 文件。

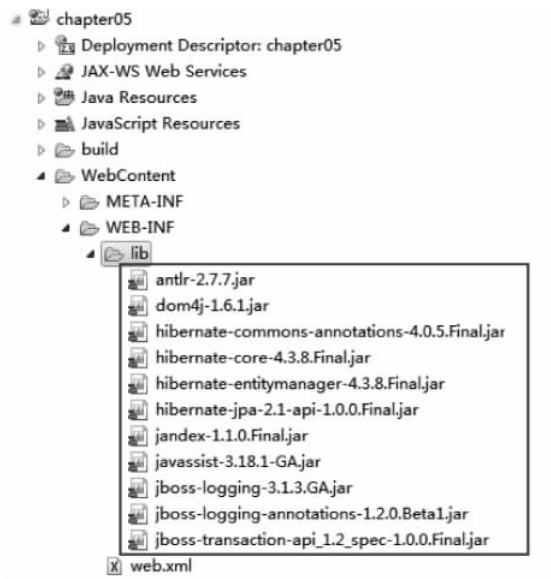


图 5-9　Hibernate 所必需的 jar 包

注意

本书使用 Hibernate4.3.8 Final 版，因 Hibernate 各版本之间可能存在一些细节差异，在应用中不必一次性将所有 jar 文件都复制到应用程序中，而是根据需要添加相应的 jar 文件即可，避免在 S2SH 框架整合时出现版本冲突。

添加完 Hibernate 所必需的 jar 包后，还需要创建 Hibernate 的配置文件 hibernate.cfg.xml，并将该文件放在类文件的根目录（即 src 目录）下。

5.6.2　编写 PO

Hibernate 中每个 PO 是由“POJO ＋映射文件”组成，因此编写 PO 需要编写 POJO 持久化类，以及 POJO 所对应的映射文件。

在 com.qst.chapter05.pojos 包下创建一个持久化类 User。

【代码 5-4】　User.java

```
package com.qst.chapter05.pojos;

import java.io.Serializable;

public class User implements Serializable {
    /* 用户 ID */
    private Integer id;
    /* 用户名 */
    private String userName;
    /* 密码 */
    private String userPwd;
    /* 权限 */
    private Integer role;
```

```
    /* 默认构造方法 */
    public User() {
    }
    /* 根据属性创建构造方法 */
    public User(String userName, String userPwd,Integer role) {
        this.userName = userName;
        this.userPwd = userPwd;
        this.role = role;
    }
    ...省略 getter 和 setter 方法
}
```

上述代码创建了一个持久化实体类 User,该类中分别提供了带参数和不带参数的构造方法,其属性有用户 ID、用户名、密码和权限 4 项,并对这些属性提供相应的 getter 和 setter 方法。

创建 User 类所对应的映射文件 User.hbm.xml,该映射文件与 User 类放在同一目录下,如图 5-10 所示。

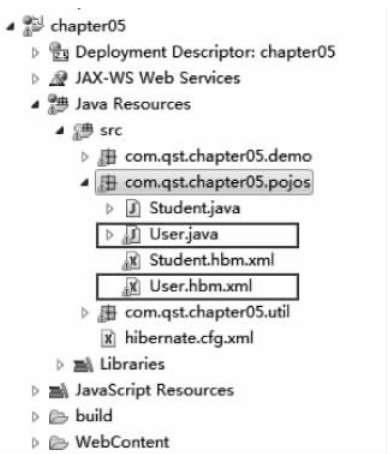


图 5-10 映射文件的位置

【代码 5-5】 User.hbm.xml

```
<?xml version = "1.0"?>
<!DOCTYPE hibernate - mapping PUBLIC
    " - //Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate - mapping - 3.0.dtd">
<hibernate - mapping>
    <class name = "com.qst.chapter05.pojos.User" table = "tbUsers">
        <!-- 主键 -->
        <id name = "id" column = "ID">
            <generator class = "native" />
        </id>
        <!-- 用户名 -->
        <property name = "userName" column = "USERNAME" type = "string"
            not - null = "true" />
        <!-- 密码 -->
        <property name = "userPwd" column = "USERPWD" type = "string"
```

```
            not-null="true" />
        <!-- 权限 -->
        <property name="role" column="ROLE" type="java.lang.Integer"
            not-null="true" />

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

上述配置文件将 User 类和数据库中的 tbUsers 表进行映射。

在 hibernate.cfg.xml 配置文件中注册 User.hbm.xml 映射文件，代码如下所示。

【代码 5-6】 hibernate.cfg.xml

```
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!-- 配置访问 Oracle 数据库参数信息 -->
        <property name="dialect">
            org.hibernate.dialect.OracleDialect
        </property>
        <property name="connection.driver_class">
            oracle.jdbc.driver.OracleDriver
        </property>
        <property name="connection.url">
            jdbc:oracle:thin:@localhost:1521:orcl
        </property>
        <property name="connection.username">scott</property>
        <property name="connection.password">scott123</property>
        <!-- 在控制台显示 SQL 语句 -->
        <property name="show_sql">true</property>
        <!--根据需要自动生成、更新数据表 -->
        <property name="hbm2ddl.auto">update</property>
        <!-- 注册所有 ORM 映射文件 -->
        <mapping resource="com/qst/chapter05/pojos/Student.hbm.xml" />
        <mapping resource="com/qst/chapter05/pojos/User.hbm.xml" />

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

5.6.3 创建 Configuration 对象

Configuration 对象代表一个应用程序到数据库的映射配置。根据 Hibernate 使用的配置文件的不同，创建 Configuration 对象的方式也不同，通常采用 hibernate.cfg.xml 文件作为 Hibernate 的配置文件，此时创建 Configuration 对象的示例代码如下所示。

【示例】 创建 Configuration 对象

```
//实例化 Configuration
Configuration configuration = new Configuration();
//加载 hibernate.cfg.xml 文件
configuration.configure("/hibernate.cfg.xml");
```

上述代码也可以进行简化，代码如下：

```
//实例化 Configuration 对象，并加载 hibernate.cfg.xml 文件
Configuration configuration = new Configuration()
                                    .configure("/hibernate.cfg.xml");
```

Configuration 对象可以产生一个不可变的 SessionFactory 对象，Configuration 对象只存在于系统的初始化阶段，所有持久化操作都通 SessionFactory 实例来完成。

通过 new 关键字创建 Configuration 对象之后，不要忘记调用 configure()方法。SessionFactory 与 Configration 对象之间不存在反向的关联关系。

5.6.4 创建 SessionFactory

通过 Configuration 对象的 buildSessionFactory()方法可以创建一个 SessionFactory 对象，SessionFactory 对象是 Hibernate 进行持久化操作所必需的对象，该对象是整个数据库映射关系经编译后形成的内存镜像。通常一个应用程序只有一个 SessionFactory 的实例，并且 SessionFactory 的实例是唯一的、不可改变的。

【示例】 创建 SessionFactory 对象

```
//Hibernate4.3 创建 SessionFactory 的方式
StandardServiceRegistryBuilder standardServiceRegistryBuilder =
        new StandardServiceRegistryBuilder();
standardServiceRegistryBuilder.applySettings(configuration.getProperties());
SessionFactory sessionFactory = configuration.buildSessionFactory(
        standardServiceRegistryBuilder.build());
```

上述代码中，先创建一个 StandardServiceRegistryBuilder 对象，该对象具有注册表功能，可以对服务进行统一加载、初始化、存放和获取。Configuration 对象会根据配置文件的内容构建 SessionFactory 实例，即 SessionFactory 一旦构建完毕，就会包含配置文件的信息，之后任何对 Configuration 实例的改变都不会影响到已经构建的 SessionFactory 实例对象。

5.6.5 获取 Session

Session 对象是 Hibernate 持久化操作的基础，是应用程序与数据库之间交互操作的一个关键对象。持久化对象的生命周期、事务的管理、对象的查询、更新和删除等操作都是通过 Session 对象完成的。

通过调用 SessionFactory 对象中的 openSession()或 getCurrentSession()方法都可以获取一个 Session 对象，代码如下所示。

【示例】 获取 Session 对象

```
//获取 Session
Session session = sessionFactory.openSession();
```

Session 对象封装了 JDBC 连接，具有一个一级缓存，在显式执行 flush()方法之前，所有持久化操作的数据都在 Session 对象的缓存中。

Session 对象中常用的方法及功能如表 5-9 所示。

表 5-9　Session 中的方法

方　法	功 能 描 述
save()	保存持久化对象，在数据库中新增一条记录
get()	获取数据库中的一条记录，当未找到符合条件的持久化对象时返回 null
load()	获取数据库中的一条记录，当未找到符合条件的持久化对象时会抛出异常
update()	更新数据库中对应的数据
delete()	删除数据库中的一条记录

get()和 load()方法都可以根据标识符属性查询并获取一个持久化对象，但是在未找到符合条件的持久化对象时，get()方法返回 null，而 load()方法抛出一个 HibernateException 异常。另外，get()方法查找对象时先从 Hibernate 一级缓存中查询，找不到则直接从数据库中查找记录；而 load()方法在一级缓存中找不到的情况下，还会查找 Hibernate 的二级缓存，仍未找到才查找数据库。

5.6.6　使用 Transaction 管理事务

Transaction 对象主要用于管理事务，所有持久化操作都需要在事务管理下进行。Transaction 通过抽象将应用程序从底层的 JDBC、JTA 以及 CORBA 事务中隔离开，允许开发人员使用一个统一的事务操作让自己的项目可以在不同的环境和容器之间迁移。

通过 Session 对象的 beginTransaction()方法可以获得一个 Transaction 对象的实例。

【示例】　获取 Transaction 对象

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

Transaction 中主要定义了 commit()和 rollback()两个方法：

- commit()方法，用于提交事务；
- rollback()方法，用于回滚事务。

一个 Transaction 对象的事务可能包括多个持久化操作，程序员可以根据需要将多个持久化操作放在开始事务和提交事务之间，从而形成一个完整的事务。

【示例】　事务

```
//开始一个事务
Transaction trans = session.beginTransaction();
//多个持久化操作
...
//提交事务
trans.commit();
```

通常 Hibernate 执行数据库事务的时序图如图 5-11 所示。

下述代码将 User 对象信息保存到数据库中。

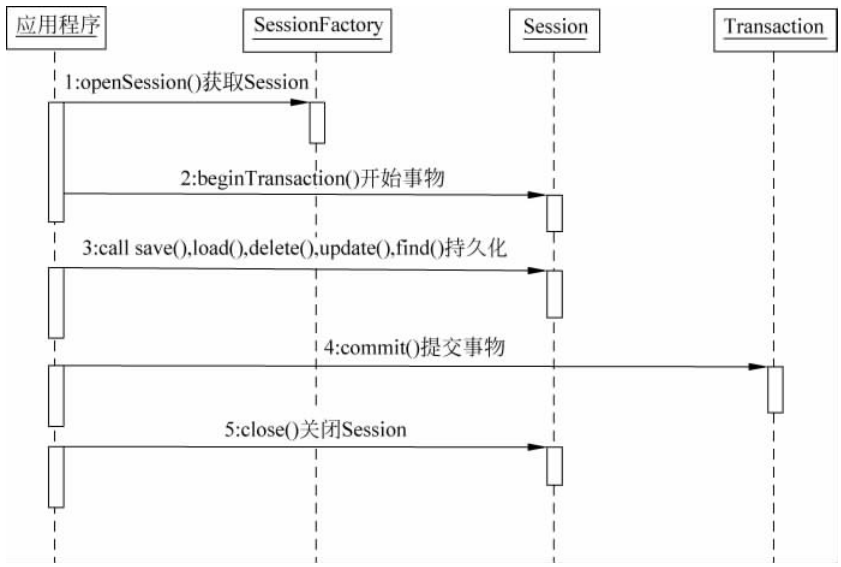


图 5-11 执行数据库事务的时序图

【代码 5-7】 UserDemo.java

```
package com.qst.chapter05.demo;

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

import com.qst.chapter05.pojos.User;

public class UserDemo {

    public static void main(String[] args) {
        //创建 User 对象
        User user = new User("zhangsan", "123", 1);
        //实例化 Configuration
        Configuration configuration = new Configuration();
        //加载 hibernate.cfg.xml 文件
        configuration.configure("/hibernate.cfg.xml");
        //创建 SessionFactory
        //Hibernate4.3 创建 SessionFactory 的方式
        StandardServiceRegistryBuilder standardServiceRegistryBuilder =
            new StandardServiceRegistryBuilder();
        standardServiceRegistryBuilder.applySettings(configuration
                .getProperties());
        SessionFactory sessionFactory = configuration
                .buildSessionFactory(standardServiceRegistryBuilder.build());
        //打开 Session
        Session session = sessionFactory.openSession();
        //开始一个事务
        Transaction trans = session.beginTransaction();
        //持久化操作
```

```
            session.save(user);
            //提交事务
            trans.commit();
            //关闭 Session
            session.close();
        }

    }
```

上述代码中先实例化一个 User 对象，再使用 Hibernate 将此对象保存到 Oracle 数据库中。运行上述代码，则在 tbUsers 表中插入一条新的记录，如图 5-12 所示。

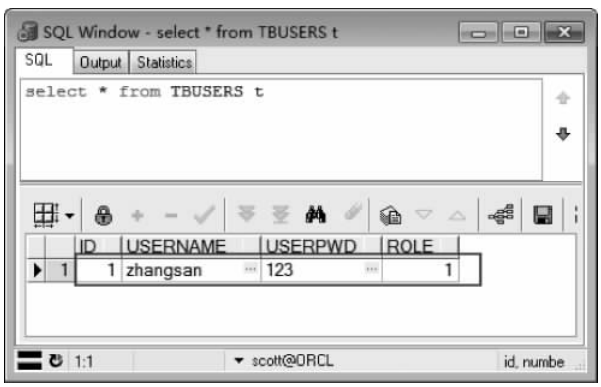


图 5-12　添加一个用户记录

第一次执行时，如果数据库中没有 User 表，Hibernate 会根据配置文件和映射文件自动生成 User 表，以后再次执行时则更新 User 表。这是因为在前面配置 hibernate.cfg.xml 文件时配置了如下一条属性：

```
<!-- 根据需要自动生成、更新数据表 -->
<property name = "hbm2ddl.auto"> update </property>
```

hbm2ddl.auto 属性值为 update，这是最常用的属性值，该属性可以根据 POJO 类及其映射文件生成表，即使表结构改变了，表中的记录仍然存在，不会删除以前的数据。

下述代码在本章创建的持久化类 Student 及其映射文件 Student.hbm.xml 基础上，将学生对象信息保存到数据库中。

【代码 5-8】　StudentDemo.java

```
package com.qst.chapter05.demo;

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

import com.qst.chapter05.pojos.Student;

public class StudentDemo {

    public static void main(String[] args) {
```

```
        //创建 Student 对象
        Student stu = new Student();
        stu.setName("张三");
        stu.setScore(98);
        //实例化 Configuration
        Configuration configuration = new Configuration();
        //加载 hibernate.cfg.xml 文件
        configuration.configure("/hibernate.cfg.xml");
        //创建 SessionFactory
        //Hibernate4.3 创建 SessionFactory 的方式
        StandardServiceRegistryBuilder standardServiceRegistryBuilder =
            new StandardServiceRegistryBuilder();
        standardServiceRegistryBuilder.applySettings(configuration
                .getProperties());
        SessionFactory sessionFactory = configuration
                .buildSessionFactory(standardServiceRegistryBuilder.build());
        //打开 Session
        Session session = sessionFactory.openSession();
        //开始一个事务
        Transaction trans = session.beginTransaction();
        //持久化操作
        session.save(stu);
        //提交事务
        trans.commit();
        //关闭 Session
        session.close();
    }

}
```

执行上述代码，则在 Students 表中添加一个学生记录，如图 5-13 所示。

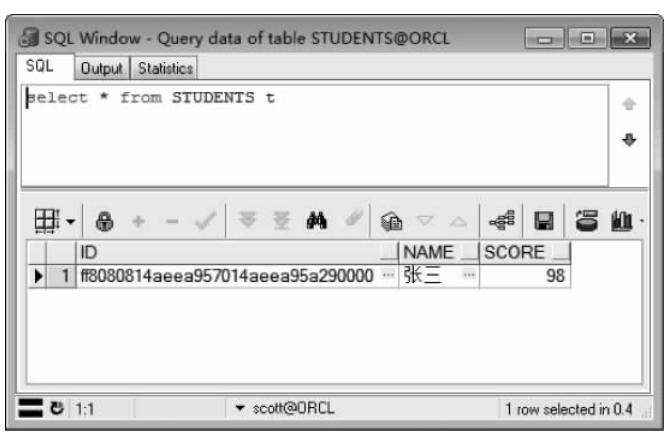


图 5-13 添加一个学生记录

5.6.7 使用 Query 进行 HQL 查询

Hibernate 提供功能强大的查询体系，使用 Hibernate 有多种查询方式，包括 HQL（Hibernate Query Language）查询、SQL 查询等。其中 HQL 是完全面向对象查询语言，所操

作的对象是类、实例、属性等。

HQL 被广泛地使用，主要在于其具备以下几个方面的优势：

- 支持继承、多态等特性；
- 支持各种条件查询、连接查询和子查询；
- 支持分页、分组查询；
- 支持各种聚集函数和自定义函数；
- 支持动态绑定查询参数。

HQL 查询依赖于 Query 类，每个 Query 实例对应一个查询对象。

利用 Query 进行 HQL 查询的步骤如下：

(1) 获取 Hibernate Session 对象；

(2) 编写 HQL 语句；

(3) 以 HQL 语句作为参数，调用 Session 的 createQuery()方法创建 Query 查询对象；

(4) 如果 HQL 语句包含参数，则调用 Query 的 setXxx()方法为参数赋值；

(5) 调用 Query 对象的 list 等方法返回查询结果。

Query 对象通过 Session 对象的 createQuery()方法创建，示例代码如下所示。

【示例】 创建 Query 对象

```
Query query = session.createQuery("from User");
```

其中 createQuery()方法的参数“from User”是 HQL 语句，表示读取所用 User 类型的对象，即将 User 类对应表中的所有记录封装成 User 对象并保存到 List 集合中。

下述代码利用 Query 查询所有用户信息。

【代码 5-9】 UserHQLDemo.java

```
package com.qst.chapter05.demo;

import java.util.List;

import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

import com.qst.chapter05.pojos.User;

public class UserHQLDemo {

    public static void main(String[] args) {
        //实例化 Configuration
        Configuration configuration = new Configuration();
        //加载 hibernate.cfg.xml 文件
        configuration.configure("/hibernate.cfg.xml");
        //创建 SessionFactory
        //Hibernate4.3 创建 SessionFactory 的方式
        StandardServiceRegistryBuilder standardServiceRegistryBuilder =
            new StandardServiceRegistryBuilder();
        standardServiceRegistryBuilder.applySettings(configuration
```

```
                .getProperties());
        SessionFactory sessionFactory = configuration
                .buildSessionFactory(standardServiceRegistryBuilder.build());
        //打开 Session
        Session session = sessionFactory.openSession();
        //开始一个事务
        Transaction trans = session.beginTransaction();
        //查询 Customer 表
        Query query = session.createQuery("from User");
        //执行查询
        List<User> list = query.list();
        //遍历输出
        for (User u : list) {
            System.out.println(u.getId() + "\t" + u.getUserName() + "\t"
                    + u.getUserPwd() + "\t" + u.getRole());
        }
        //提交事务
        trans.commit();
        //关闭 Session
        session.close();

    }

}
```

上述代码使用 Query 对象的 list()方法执行查询并返回查询结果的集合，再遍历集合中的每个元素然后输出，运行结果如下所示：

```
Hibernate: select user0_.ID as ID1_1_, user0_.USERNAME as USERNAME2_1_, user0_.USERPWD as
USERPWD3_1_, user0_.ROLE as ROLE4_1_ from tbUsers user0_
1  zhangsan   123     1
2  李四       123456  1
```

5.6.8 使用 Criteria 进行条件查询

上述的 Query 方式使用非常方便，但是缺点也非常明显，即 Query 所使用的 HQL 是字符串字面值，这不太符合面向对象的原则，并且 HQL 编写过程中也容易出错。为此，Hibernate 提供了一种更加面向对象的查询方式，即 Criteria 方式，Criteria 类提供了丰富的方法，能够以面向对象方式组装查询逻辑。通过 Session 对象的 createCriteria()方法可以创建 Criteria 的示例，代码格式如下：

```
Criteria criteria = session.createCriteria(Customer.class);
```

下述代码利用 Criteria 查询所有用户信息。

【代码 5-10】 StudentCriteriaDemo.java

```
package com.qst.chapter05.demo;

import java.util.List;

import org.hibernate.Criteria;
import org.hibernate.Session;
```

```
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

import com.qst.chapter05.pojos.Student;

public class StudentCriteriaDemo {

    public static void main(String[] args) {
        //实例化 Configuration
        Configuration configuration = new Configuration();
        //加载 hibernate.cfg.xml 文件
        configuration.configure("/hibernate.cfg.xml");
        //创建 SessionFactory
        //Hibernate 4.3 创建 SessionFactory 的方式
        StandardServiceRegistryBuilder standardServiceRegistryBuilder =
            new StandardServiceRegistryBuilder();
        standardServiceRegistryBuilder.applySettings(configuration
                .getProperties());
        SessionFactory sessionFactory = configuration
                .buildSessionFactory(standardServiceRegistryBuilder.build());
        //打开 Session
        Session session = sessionFactory.openSession();
        //开始一个事务
        Transaction trans = session.beginTransaction();
        //创建一个 Criteria 查询对象,查询 Student 类的所有对象
        Criteria criteria = session.createCriteria(Student.class);
        //执行查询
        List<Student> list = criteria.list();
        //遍历输出
        for (Student stu : list) {
            System.out.println(stu.getId() + "\t" +
                    stu.getName() + "\t" + stu.getScore());
        }
        //提交事务
        trans.commit();
        //关闭 Session
        session.close();
    }

}
```

使用 Criteria 进行条件查询与 Query 类似,执行上述代码,控制台输出结果如下所示。

```
Hibernate: select this_.ID as ID1_0_0_, this_.NAME as NAME2_0_0_, this_.SCORE as SCORE3_0_0_
from STUDENTS this_
402881e44af6dc0a014af6dc2e6e0000  张三  98
402881e44af6e690014af6e6ae6e0000  李四  98
```

使用 Query、Criteria 进行复杂的查询的详细内容参见本书第 6 章。

5.7 POJO 状态

当应用程序通过 new 语句创建一个 POJO 对象时，这个对象的生命周期就开始了，JVM 会为该对象分配一块内存空间，只要该对象被引用变量引用，就一直存在内存中；当 POJO 对象不被任何引用变量引用，则该对象的生命周期结束，JVM 的垃圾回收器将收回其所占用的内存空间。

对于需要被持久化的 POJO 对象，其生命周期有以下三个状态：

- **瞬时状态(Transient)**——刚刚用 new 关键字创建，还没有被持久化，且尚未与 Hibernate Session 关联，不处于 Session 的缓存中，此时的对象处于瞬时状态。处于瞬时状态的对象被称为瞬态对象，瞬态对象不会被持久化到数据库，也不会被赋予持久化标识，如果程序中失去了瞬态对象的引用，瞬态对象将被垃圾回收机制销毁。
- **持久化状态(Persistent)**——已经被持久化，加入到 Session 缓存中，与数据库中表的一条记录对应，并拥有一个持久化标识。持久化状态的对象可以是刚刚保存的，也可以是刚被加载的。Hibernate 会检测到持久化状态的对象的任何改动，并且在 Session 关闭或 Transaction 提交的同时更新数据库中的对应数据，而不需要手动更新。
- **脱管状态**(Detached)——已经被持久化，但不再处于 Session 的缓存中。如果对象曾经处于持久化状态，但与之关联的 Session 关闭后，则该对象就变成脱管状态；如果重新让脱管对象与某个 Session 发生了关联，则这个脱管对象将会转换成持久化状态。

POJO 持久化对象的状态转换如图 5-14 所示。

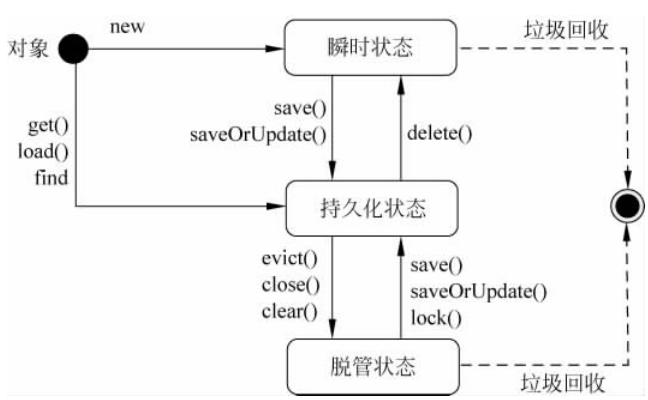


图 5-14 Hibernate 持久化对象状态转换图

注意

> 当调用 Session 的 close()、clear()方法时，所有与该 Session 有关联的对象都将受到影响。

调用 Hibernate Session 的不同方法会引起持久化对象的状态改变，Hibernate Session 中的方法及对持久化对象状态所产生的影响如表 5-10 所示。

表 5-10　Hibernate Session 中的方法及影响

状　态	方 法 名	功 能 描 述
持久化状态	save()	该方法保存持久化对象，进而在数据库中新增一条数据
	saveOrUpdate()	保存或更新，该方法根据映射文件中的＜id＞标签的 unsaved-value 属性值决定执行新增或更新
	get()	根据标识符属性值获取一个持久化对象，如果未找到，则返回 null
	load()	根据标识符属性值加载一个持久化对象，如果未找到，则抛出异常
	update()	对脱管状态的对象重新完成持久化，并更新数据库中对应的数据
瞬时状态	delete()	用于删除数据库表中的一条记录，在删除时，首先需要 get()或 load()方法获取要删除记录对应的持久化对象，然后调用 delete()方法删除
脱管状态	close()	关闭当前 Session 对象，并清空该对象中的数据
	evict()	清除 Session 缓存中的某个对象
	clear()	清除 Session 中所有缓存对象

在执行 Hibernate 持久化操作时都需要创建 Configuration 对象、SessionFactory 对象和 Session 对象，因此可以将这部分代码封装成一个工具类，以便重复调用该工具类中的相应方法。

下述代码创建一个 HibernateUtils 工具类，该类对 Hibernate 的常用操作进行封装，代码如下所示。

【代码 5-11】　HibernateUtils. java

```
package com.qst.chapter05.util;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.boot.registry.StandardServiceRegistryBuilder;
import org.hibernate.cfg.Configuration;

public class HibernateUtils {
    private static String CONFIG_FILE_LOCATION = "/hibernate.cfg.xml";
    private static final ThreadLocal<Session> threadLocal
                                    = new ThreadLocal<Session>();
    private static Configuration configuration = new Configuration();
    private static StandardServiceRegistryBuilder standardServiceRegistryBuilder = new
StandardServiceRegistryBuilder();
    private static SessionFactory sessionFactory;
    private static String configFile = CONFIG_FILE_LOCATION;
    /* 静态代码块创建 SessionFactory */
    static {
        try {
            configuration.configure(configFile);
            //Hibernate 4.3 创建 SessionFactory 的方式

    standardServiceRegistryBuilder.applySettings(configuration.getProperties());
            sessionFactory = configuration.buildSessionFactory(standardServiceRegistryBuilder
.build());
        } catch (Exception e) {
            System.err.println("%%%% Error Creating SessionFactory %%%%");
            e.printStackTrace();
```

```
        }
    }
    private HibernateUtils() {
    }
    /**
     * 返回 ThreadLocal 中的 session 实例
     */
    public static Session getSession() throws HibernateException {
        Session session = (Session) threadLocal.get();
        if (session == null || !session.isOpen()) {
            if (sessionFactory == null) {
                rebuildSessionFactory();
            }
            session = (sessionFactory != null) ? sessionFactory.openSession()
                    : null;
            threadLocal.set(session);
        }
        return session;
    }
    /**
     * 返回 Hibernate 的 SessionFactory
     */
    public static void rebuildSessionFactory() {
        try {
            configuration.configure(configFile);
            sessionFactory = configuration.buildSessionFactory(standardServiceRegistryBuilder
.build());
        } catch (Exception e) {
            System.err.println("%%%% Error Creating SessionFactory %%%%");
            e.printStackTrace();
        }
    }
    /**
     * 关闭 Session 实例并且把 ThreadLocal 中的副本清除
     */
    public static void closeSession() throws HibernateException {
        Session session = (Session) threadLocal.get();
        threadLocal.set(null);
        if (session != null) {
            session.close();
        }
    }
    /**
     * 返回 SessionFactory
     */
    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
    public static void setConfigFile(String configFile) {
        HibernateUtils.configFile = configFile;
        sessionFactory = null;
    }
    public static Configuration getConfiguration() {
        return configuration;
    }
}
```

上述代码提供了获取 SessionFactory、获取 Session 和关闭 Session 等操作的静态方法。该工具类定义后，Hibernate 应用中只需直接调用该类中的相应方法即可。

下述代码演示 Session 接口中其他核心方法的使用，代码如下所示。

【代码 5-12】 HibernateSessionDemo.java

```
package com.qst.chapter05.demo;

import org.hibernate.Session;
import org.hibernate.Transaction;

import com.qst.chapter05.pojos.User;
import com.qst.chapter05.util.HibernateUtils;

public class HibernateSessionDemo {

    public static void main(String[] args) {
        //调用 getUser()方法获取用户对象
        User user = getUser(new Integer(1));
        System.out.println("--------原始数据----------");
        System.out.println(user.getId() + "\t" + user.getUserName()
                + "\t" + user.getUserPwd() + "\t" +user.getRole());
        user.setUserPwd("987654");
        //调用 changeUser()方法修改用户对象信息
        changeUser(user);
        System.out.println("--------修改后的数据----------");
        System.out.println(user.getId() + "\t" + user.getUserName()
                + "\t" + user.getUserPwd() + "\t" +user.getRole());
    }
    /* 获取用户 */
    public static User getUser(Integer key) {
        Session session = HibernateUtils.getSession();
        Transaction trans = session.beginTransaction();
        //根据主键获取用户对象
        User user = (User) session.get(User.class, key);
        trans.commit();
        HibernateUtils.closeSession();
        return user;
    }
    /* 修改用户信息 */
    public static void changeUser(User user) {
        Session session = HibernateUtils.getSession();
        Transaction trans = session.beginTransaction();
        //更新
        session.update(user);
        trans.commit();
        HibernateUtils.closeSession();
    }

}
```

上述代码中定义了获取用户和修改用户信息的两个静态方法，并在方法中直接调用 HibernateUtils 类中的方法获取、关闭 Session 对象。在 main()方法中首先调用 getUser()方法获取 ID 是 1 的用户对象并输出，再调用 changeUser()方法修改此用户密码。

执行 HibernateSessionDemo.java 程序，在控制台将输出如下所示的结果。

```
Hibernate: select user0_.ID as ID1_1_0_, user0_.USERNAME as USERNAME2_1_0_, user0_.USERPWD as
USERPWD3_1_0_, user0_.ROLE as ROLE4_1_0_ from tbUsers user0_ where user0_.ID = ?
-------- 原始数据 ----------
1  zhangsan  123  1
Hibernate: update tbUsers set USERNAME = ?, USERPWD = ?, ROLE = ? where ID = ?
-------- 修改后的数据 ----------
1  zhangsan  987654  1
```

查看数据库中 tbUsers 表，密码已经更改成新的密码，如图 5-15 所示。

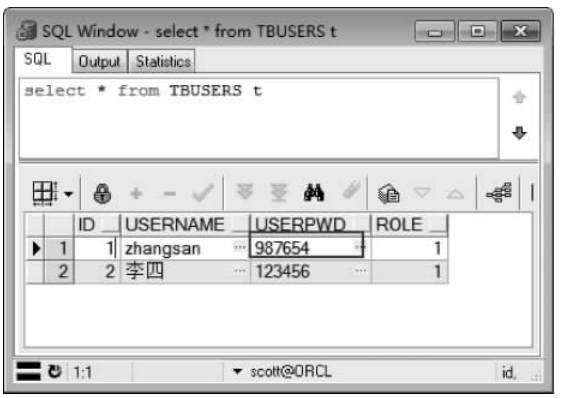


图 5-15　tbUsers 表中记录的更新

5.8　贯穿任务实现

5.8.1　实现任务 5-1

下述内容升级任务 3-3 的登录功能，并完成客户端的 JS 校验功能，步骤如下：

(1) 在 UserService 类中编写 findUserByNameOrMobile()方法，可以根据用户名或手机号查询用户对象。

(2) 调用 HibernateUtils 类提供的静态方法以获得 Session 对象，处理完后并关闭 Session 对象。

(3) 在 UserDao 类中采用 Hibernate 的方式操作数据库，完成对 User 对象的查询方法。

1. 创建 UserService 类

【任务 5-1】　UserService.java

```
package com.qst.giftems.user.service;
import com.qst.giftems.user.daos.UserDao;
import com.qst.giftems.user.pojos.User;
public class UserService {
    /** UserDao **/
    private UserDao userDao = new UserDao();
    /**
```

```
     * 根据用户名或手机号查询用户对象
     * @param name
     * @return
     */
    public User findUserByNameOrMobile(String name) {
        return userDao.findUserByNameMobile(name);
    }
…省略代码
```

上述代码中，使用 UserDao 对象时，需要显示的创建一个对象进行赋值。

在后面章节中，与 Spring 框架进行集成后，将采用 IOC 以注入的方式进行对象的赋值。

2. 创建 UserDao 类

【任务 5-1】 UserDao.java

```
package com.qst.giftems.user.daos;
import org.hibernate.Query;
import org.hibernate.Session;
import com.qst.core.utils.HibernateUtils;
import com.qst.giftems.user.pojos.User;
public class UserDao {
    /**
     * 根据姓名或电话进行查询
     * @param name
     * @return
     */
    public User findUserByNameMobile(String name) {
        Session session = null;
        try {
            //获取 Session 对象
            session = HibernateUtils.getSession();
            //创建 Query 对象，并使用 HQL 进行查询
            Query query = session
                    .createQuery("from User u where u.userName = ? or u.mobile = ?");
            //设置参数
            query.setString(0, name);
            query.setString(1, name);
            //返回唯一 User 对象
            User user = (User) query.uniqueResult();
            return user;
        } finally {
            //释放 Session 对象
            HibernateUtils.closeSession();
        }
    }
}
```

上述代码中，通过 HibernateUtils 类获取 Session 对象，然后利用 Session 对象创建 Query 对象，并通过 HQL 进行查询并返回唯一的 User 对象，如果无查询结果，则返回 null。

LoginAction 的修改代码如下所示。

【任务 5-1】 LoginAction. java

```
/**
 * 用户登录
 */
public String login() {
    HttpServletRequest request = ActionContextUtils.getRequest();
    HttpSession session = request.getSession();
    //回传字符串
    String result = LOGIN;
    //处理刷新系统的情况
    if (LoginManager.isOnline()) {
        result = INDEX;
        ActionContextUtils.setAtrributeToRequest("msg","登录成功!");
        return result;
    }
    try {
        if (StringUtils.isEmpty(this.userName)) {
            //如果用户名为空,返回提示信息
            ActionContextUtils.setAtrributeToRequest("msg","登录成功!");
        } else if (StringUtils.isEmpty(this.password)) {
            //如果密码为空,返回提示信息
            ActionContextUtils.setAtrributeToRequest("msg","密码不能为空!");
        } else {
            //根据姓名或电话进行查询
            User user =
                    userService.findUserByNameOrMobile(userName.trim());
            //判读用户是否已经注册
            if (user != null) {
                String pass2 = new MD5().getMD5ofStr(this.password);
                if (!user.getPassword().equals(pass2)) {
                    //判断密码是否正确
                    ActionContextUtils.setAtrributeToRequest("msg","密码错误!");
                } else if (user.getStatus() != 0) {
                    //用户冻结
                    ActionContextUtils.setAtrributeToRequest("msg","用户已经被冻结!");
                } else {
                    User user2 = LoginManager.getLoggedinUser(user.getId());
                    if (user2 != null) {
                        session.setAttribute("anotherUser", user2);
                        LoginManager.logout(user2.getId());
                    }
                    //登录时间
                    user.setLastLoginTime(new Date(System
                            .currentTimeMillis()));
                    LoginManager.login(user);
                    //保存信息
                    ActionContextUtils.setAtrributeToRequest("user", user);
                    //返回成功界面
                    result = INDEX;
            ActionContextUtils.setAtrributeToRequest("msg","登录成功!");
                }
            } else {
```

```
                ActionContextUtils.setAtrributeToRequest("msg","用户不存在!");
            }
        }
    } catch (Exception ex) {
            String event = ExceptionUtils.formatStackTrace(ex);
            logger.error(event);
        }
        return result;
    }
```

上述代码的加粗部分首先使用 UserService 对象获取 User 对象，然后使用了 MD5 算法对用户传递上来的密码进行加密，并和数据库中获取的已经加密的密码进行比较。如果用户名、密码都相同，则登录成功，否则登录失败，并返回到登录页面，同时以相应错误消息进行提示。

下述内容是用户在登录过程中的校验规则内容如下：

(1) 账号不能为空，登录账号可以为手机号码。

(2) 密码不能为空，在 6～15 位之间。

在实际应用中，通常通过 JS 来实现上述校验规则，本项目中通过 jQuery 框架实现了上述规则，代码如下：

【任务 5-1】 Login.jsp 中的 JS 校验

```
$(function(){
    //提示信息
    var userNameInfoStr = "账号长度在 3～20 位之间";
    var passwordInfoStr = "密码长度在 6～15 位之间";
    //输入框
    var userNameInput = $("input#userName");
    var passwordInput = $("input#password");
    //提示区域
    var userNameSpan = $("span#userName-info-span");
    var passwordSpan = $("span#password-info-span");
    //当键盘按键被按下时,会发生该事件.它发生在当前获得焦点的元素上,
    //每插入一个字符,就会发生 keypress 事件.
    userNameInput.bind("keyup", function(event) {
        var userName = userNameInput.val();
        //删除按钮  backspace
        if (event.which == 8) {
            if ( $.trim(userName) == "") {
                userNameSpan.text(userNameInfoStr);
            }else{
                userNameSpan.text("");
            }
            return;
        }
    });
    passwordInput.bind("keyup", function(event) {
        var password = passwordInput.val();
        //删除按钮 backspace
        if (event.which == 8) {
            if ( $.trim(password) == "") {
                passwordSpan.text(passwordInfoStr);
            }else{
                passwordSpan.text("");
```

```
            }
            return;
        }
    });
    //提交表单
    function submitLoginForm(event){
        //分别获取用户名和密码的值
        var userName = userNameInput.val();
        var password = passwordInput.val();
        //进行用户名位数校验
        if ($.trim(userName) == ""
        || $.trim(userName).length<3||$.trim(userName).length>20) {
            userNameSpan.text(userNameInfoStr);
            userNameInput.focus();
            return;
        }
        userNameSpan.text("");
        //进行密码位数校验
        if ($.trim(password) == ""
        || $.trim(password).length<6 || $.trim(password).length>15) {
            passwordSpan.text(passwordInfoStr);
            passwordInput.focus();
            return;
        }
        userNameSpan.text("");
        $("#loginForm").submit();
    }
    //提交表单
    $("#login-button").click(submitLoginForm);
    $(".close").click(function(){
        history.back(-1);
    });
});
```

当用户没有输入密码并单击“登录”按钮时，效果图如图 5-16 所示。

图 5-16 登录时的校验

5.8.2 实现任务 5-2

下述内容实现用户注册功能，并完成客户端的 JS 校验功能。步骤如下：

(1) 编写注册界面 register.jsp，并完成基本的 JS 校验功能。

(2) 在 LoginAction 中添加 register()方法，实现注册成功或失败的请求转发功能。

(3) 分别在 UserService 和 UserDao 中添加相应的业务方法完成注册功能。

(4) 在 struts.xml 中配置请求转发。

(5) 为了避免暴力注册，添加验证码功能。

1. 创建 register.jsp 文件

在 WebContent/login 文件夹中创建 register.jsp 文件，实现基本的用户注册功能。核心代码如下。

【任务 5-2】 register.jsp

```
<form id="register-form" name="register-form"
              action="${ctx}/user/l.action?method=register" method="post">
            <dl class="reg_text">
                <dt>账号：</dt>
                <dd>
                    <input type="text" class="input1" name="userName"
            id="userName" autocomplete="off"
placeholder="至少 3 位以上" onblur="javascript:validateUserName()"/><span class="red"id=
"userName-info-span"></span><span id="name_message"></span>
                </dd>
                <dt>密码：</dt>
                <dd>
                    <input type="password" name="password" class="input1"
                        placeholder="长度在 6-15 位之间" /><span class="red"
                        id="password-info-span"></span>
                </dd>
                <dt>确认密码：</dt>
                <dd>
                    <input name="confirm_password" type="password" class="input1"
                        placeholder="密码确认" />
                </dd>
                <dt>手机号码：</dt>
                <dd>
                    <input name="mobile" type="text" id="mobile" class="input1"
autocomplete="off"
                        placeholder="请输入手机号"/>
                </dd>
                <c:if test="${not empty info}">
                  <dd>   提示：<font style="font-size: 12;color: red">
${info}</font></dd>
                </c:if>
                <dd><input name="" type="checkbox" value="" checked="checked" style=
"vertical-align: middle;margin-right:5px;" />
        我已阅读并同意<a href="${ctx}/about.action?method=copyright" target="_blank"
class="blue">«骆礼用户注册协议»</a></dd>
            </dl>
```

```
        <div align = "center">
            < a href = "javascript:void(0)" id = "register">< span
                class = "reg_button">注册</span></a>     
            < a href = " ${ctx}/user/l.action?method = login" id = "login">< span
                class = "reg_button">直接登录</span></a>
        </div>
    </form>
```

效果图如图 5-17 所示。

图 5-17 register.jsp 界面效果图

JS 校验代码如下所示。

【任务 5-2】 register.jsp 中的 JS 校验

```
var isRegister = false;
$(function(){
        //注册账号
        $('#register').live('click',function(){
            var userName = $('#userName').val();
            //用户名长度
            if( $.trim(userName).length < 3 || userName.length > 20){
                alert("用户名长度在 3～20 位之间");
                return;
            }
            //不能连续单击多次“注册”按钮
            if(isRegister){
                return;
            }
            //判断密码长度
```

```
            var password = $('input[name=password]').val();
            if(password.length<6 || password.length>15){
                alert("密码长度在6～15位之间");
                return;
            }
            //确认密码
            var confirm_password = $('input[name=confirm_password]').val();
            if($.trim(confirm_password) == ''){
                alert("请输入确认密码!");
                return;
            }
            //前后密码是否一致
            if(confirm_password != password){
                alert("前后密码输入不一致,请确认!");
                return;
            }
            var confirm_password = $('input[name=confirm_password]').val();
            if($.trim(confirm_password) == ''){
                alert("请输入确认密码!");
                return;
            }
            if(confirm_password != password){
                alert("前后密码输入不一致,请确认!");
                return;
            }
            //验证手机格式
            var mobile = $.trim($('#mobile').val());
            var mobileReg_ = /^1\d{10}$/;
            if(mobile == '' || !mobileReg_.test(mobile)){
               alert("手机格式不正确!");
               return;
             }
            $('#register-form').submit();
        });    //end
        $(".close").click(function(){
            history.back(-1);
        });
    });
```

上述代码中对用户名、密码的长度进行了校验,同时对手机的格式进行了校验,读者可以进行验证。

在实际应用中,通常前端使用JS进行数据格式的校验,后台使用Java代码再进行同样的校验,从而保证数据的格式安全性。

2. 实现 register()方法

在 LoginAction 中添加 register()方法,实现注册功能,代码如下:

【任务 5-2】 LoginAction.java

```
package com.qst.giftems.login;
import java.util.Date;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import com.qst.core.utils.ActionContextUtils;
import com.qst.core.utils.CommonUtils;
import com.qst.core.utils.ExceptionUtils;
import com.qst.core.utils.MD5;
import com.qst.core.utils.StringUtils;
import com.qst.core.web.action.BaseAction;
import com.qst.giftems.user.pojos.User;
import com.qst.giftems.user.service.UserService;
@SuppressWarnings("serial")
public class LoginAction extends BaseAction {
    …省略内容
    /***
     * 注册功能：注册成功回到主页
     */
     @NotNecessaryLogin
     public String register(){
         //回传字符串
         String result = REGISTER;

         //获取验证码
     if(StringUtils.isEmpty(this.userName)||this.userName.length()<3 || this.userName.length()>
20){
             //如果用户名为空,返回提示信息
         ActionContextUtils.setAtrributeToRequest("msg","用户名长度在3～20之间!");
     }else if(StringUtils.isEmpty(this.password)||this.password.length()<6 || this.password
.length()>15){
             //如果密码为空,返回提示信息
         ActionContextUtils.setAtrributeToRequest("msg","密码长度在6～15之间!");
     }else if(StringUtils.isEmpty(this.mobile)){
         //验证手机号是否为空
         ActionContextUtils.setAtrributeToRequest("msg","手机号不能为空!");
     }else {
             //查询是否存在同名的用户对象并返回
             User u = userService.findUserByUserName(this.userName);
             if(u!=null && StringUtils.isNotEmpty(u.getUserName())){
                 //该用户已经注册过
                 ActionContextUtils.setAtrributeToRequest("msg","该用户已经注册过!");
             }else{
                 u = userService.findUserByNameOrMobile(this.mobile);
                 if(u!=null){
                     ActionContextUtils.setAtrributeToRequest("msg","该手机号已经注册过!");
                 }else{
                     //可以注册了
                 User user = new User();
                 user.setUserName(this.userName);
                 user.setMobile(mobile);
                 //密码进行MD5加密
                 user.setPassword(new MD5().getMD5ofStr(this.password));
                 user.setCreateTime(new Date());
                 user.setBindStatus(1);
```

```
                    user.setLastLoginIp(CommonUtils.getIP());
                    user.setDeleted(0);
                    user.setStatus(0);
                    userService.doRegister(user);
                    //处于登录状态
                    LoginManager.login(user);
                    ActionContextUtils.setAtrributeToRequest("user", user);
                    result = "regSuccess";
                    }
                }
        }
        return result;
        }
        …省略 getter/setter 方法
```

上述代码首先对用户上传的用户名、密码及手机号进行格式校验；然后查询数据库中是否已存在该用户名的用户，如果符合注册条件，把密码进行加密并设置其他默认值，然后把数据保存到数据库中从而实现了用户注册功能，最后成功跳转到首页。

3. 在 UserService 中添加注册方法

【任务 5-2】 UserService.java

```
package com.qst.giftems.user.service;
import com.qst.giftems.user.daos.UserDao;
import com.qst.giftems.user.pojos.User;
public class UserService {
    /** UserDao **/
    private UserDao userDao = new UserDao();
    …
    /**
     * 根据用户名查找用户对象
     * @param userName
     * @return
     */
    public User findUserByUserName(String userName) {
        return userDao.findUserByUserName(userName);
    }
    /**
     * 注册用户
     */
    public void doRegister(User user) {
        userDao.save(user);
    }
    …省略代码
}
```

上述代码分别定义并实现了 findUserByUserName()和 doRegister()方法，用于实现按用户名查询用户和用户的注册功能。

注意

在实际应用中，通常在 Action 层调用 Service 层的方法；在 Service 层实现具体的业务逻辑并调用 DAO 层的方法；在 DAO 层实现具体的数据库处理。本例中业务逻辑较为简单，对于复杂的业务逻辑，通常在 Service 方法中可能会调用多个 DAO 来完成一项事务处理。

4. 在 UserDao 实现注册业务方法

在 UserDao 中定义并实现 doRegister()和 findUserByUserName()方法，用于实现注册的功能，核心代码如下所示。

【任务 5-2】 UserDao.java

```
package com.qst.giftems.user.daos;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;
import com.qst.core.utils.HibernateUtils;
import com.qst.giftems.user.pojos.User;
public class UserDao {
    /**
     * 根据用户名查找
     * @param userName
     * @return
     */
    public User findUserByUserName(String userName) {
        Session session = null;
        try {
            //获取 Session 对象
            session = HibernateUtils.getSession();
            //创建 Query 对象，并使用 HQL 进行查询
            Query query = session
                    .createQuery("from User u where u.userName = ?");
            //设置参数
            query.setString(0, userName);
            //返回唯一 User 对象
            User user = (User) query.uniqueResult();
            return user;
        } finally {
            //释放 Session 对象
            HibernateUtils.closeSession();
        }
    }
    /**
     * 保存用户对象
     * @param user
     */
    public void save(User user) {
        Session session = null;
        Transaction trans = null;
        try {
            //获取 Session 对象
            session = HibernateUtils.getSession();
            //提交事务
            trans = session.beginTransaction();
          session.save(user);
          trans.commit();
        } catch(Exception ex){
            //回滚事务
            trans.rollback();
```

```
            }finally {
                //释放 Session 对象
                HibernateUtils.closeSession();
            }
        }

    }
```

上述代码分别实现了 findUserByUserName ( ) 方法和 save ( ) 方法。其中，findUserByUserName()方法用于实现根据用户名查询用户对象的功能；save()方法用于保存对象。

在调用完 Session 后，通过 HibernateUtils.closeSession()方法进行 session 的关闭来释放资源，对于初学者而言，调用了资源要及时释放，以避免造成内存泄露。

5. 配置请求转发

下述代码用于配置注册功能的请求转发，核心代码如下：

【任务 5-2】 struts.xml

```
<!-- 用户中心 -->
<package name = "user" extends = "qrsx - default" namespace = "/user">
    <!-- 登录 Action -->
    <action name = "l" class = "com.qst.giftems.login.LoginAction">
        <!-- 用户登录成功后跳转 -->
        <result name = "main">/jsp/main.jsp</result>
        <!-- 用户登录界面 -->
        <result name = "toLogin">/jsp/login/login.jsp</result>
        <!-- 用户注册界面 -->
        <result name = "toRegister">/jsp/login/register.jsp</result>
        <!-- 用户注册成功跳转 -->
        <result name = "regSuccess">/jsp/login/register_suc.jsp</result>
    </action>
</package>
```

6. 验证码功能

通常对于网站的注册功能而言，为了避免非法用户的暴力注册，一般采用验证码的方式来提高网站的安全性。创建步骤如下：

创建 ImageCodeServlet 并在 web.xml 中进行配置，核心代码如下所示。

【任务 5-2】 ImageCodeServlet.java

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
    //设置浏览器不要缓存此图片
    response.setHeader("Pragma", "No - cache");
    response.setHeader("Cache - Control", "no - cache");
```

```
        response.setDateHeader("Expires", 0);
        Map < String,Object > resultMap = ImageCodeUtils.createImageCode();
        //写到浏览器中、同时保存到 Session 中
        HttpSession session = request.getSession();
        String rand = String.valueOf(resultMap.get(ValidateCode.RAND));
        session.setAttribute("validateCode", rand);
        byte[] buf = (byte[])resultMap.get(ValidateCode.BUFFER);
        response.setContentLength(buf.length);
        //获取输出流对象
        ServletOutputStream out = response.getOutputStream();
        out.write(buf);
        //关闭流
        out.close();
    }
```

在上述代码中，在界面中如果访问上述 Servlet，将动态生成临时验证码图片并把相应图片的字节流存放到内存中，其中验证码内容存放到 Session 中用于服务器端和终端的对比。

在 web.xml 中对 ImageCodeServlet 进行配置，配置代码如下所示。

【任务 5-2】 web.xml

```
<!-- 验证码 -->
<servlet>
    <display-name>ValidateCode</display-name>
    <servlet-name>ValidateCodeServlet</servlet-name>
    <servlet-class>
        com.qst.core.web.servlet.ImageCodeServlet
    </servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>ValidateCodeServlet</servlet-name>
    <url-pattern>/imgcode</url-pattern>
</servlet-mapping>
```

限于篇幅，对 ImageCodeUtils 类不做讲解，请读者下载源代码后自行分析。

在 register.jsp 中添加验证码引用，核心代码如下所示。

【任务 5-2】 register.jsp

```
<dt>验证码：</dt>
<dd>
    <input name="validateCode" type="text" class="input2" id="validateCode"/>
        <span><a href="javascript:" onclick
           ="document.getElementById('img_code').src='${ctx}/imgcode?r='+Math.random();">
            <imgheight="30px" id="img_code"
            width="70px"
            src='${ctx}/imgcode' align="absmiddle"
            onclick="document.getElementById('img_code').src='${ctx}/imgcode?r='+Math
.random();" />
            <span class="blue">换一张</span>
                                    </a>
        </span>
</dd>
```

在上述代码中，当单击验证码图片或单击“换一张”链接时，都会访问“${ctx}/imgcode”链接对应的 Servlet，即 ImageServlet，并重新生成新的验证码图片，效果如图 5-18 所示。

图 5-18　验证码界面效果图

在 LoginAction 中的 register()方法中添加对应的验证码处理逻辑，核心代码如下：

【任务 5-2】　LogingAction.java

```
public String register(){
 …省略代码

        String validateCode =
          (String)ActionContextUtils.getAttribute("validateCode", "session");
    if(StringUtils.isEmpty(validateCode) ||
                StringUtils.isEmpty(this.validateCode)){
          //检查验证码是否为空
          ActionContextUtils.setAtrributeToRequest("msg","验证码不能为空!");
    }else if(!validateCode.trim().equalsIgnoreCase(
                    this.validateCode.trim())){
          //验证码错误
          ActionContextUtils.setAtrributeToRequest("msg","验证码错误!");
    }else{
          …省略代码
    }
```

上述代码添加了验证码为空和正确与否的判断，同时验证码不区分大小写。

5.8.3　实现任务 5-3

下述内容实现“GIFT-EMS 礼记”系统中实现登录验证功能。

前面章节的任务驱动中所有的功能，例如登录、注册、礼品中心、礼品详情等，都没有要求

在用户必须登录后才能操作，在后续的任务驱动中，例如购物车功能、个人中心、用户下单、付款等等功能要求用户必须登录后才能操作，这就需要采用 Session 登录权限验证的方式来实现上述要求。实现步骤如下：

（1）创建 NotNecessaryLogin，实现对特定方法或对类中所有方法的非 Session 权限验证。

（2）创建 Action 类，在该类中实现 Session 登录校验功能，并结合 NotNecessaryLogin 实现特定方法或类的非 Session 权限验证。

（3）针对 GiftAction 和 LoginAction 类设置不需要进行登录验证的方法。

1. 创建 NotNecessaryLogin

创建 NotNecessaryLogin 类型的注解并放到 com. qst. giftems. login 包中，代码如下所示。

【任务 5-3】 NotNecessaryLogin. java

```
package com.qst.giftems.login;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.TYPE;
import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
/**
 * 不需要登录即可执行的 Action 方法.
 *
 * 在 Action 中处理请求的方法上加上此注解后,不会判断必须登录.
 *
 * @see {@link com.yunrui.core.web.action.Action#execute()}
 */
@Documented
@Target({TYPE, METHOD})
@Retention(RetentionPolicy.RUNTIME)
public @interface NotNecessaryLogin {}
```

上述代码创建了 NotNecessaryLogin 注解，由于该注解的 Target 值为 TYPE 和 METHOD，所以该注解既可以用到类前面，也可以用到具体的方法前面。

以 GiftActioin 为例，注解用到该类前面的语法如下所示。

```
@NotNecessaryLogin
public class GiftAction extends BaseAction {}
```

这意味着 GiftAction 中的所有方法在前端调用时，都不需要登录。

注解用到类中特定方法的语法如下：

```
public class GiftAction extends Action {
    @NotNecessaryLogin
    public String giftList() {}
}
```

上述代码中，在前端调用 giftList()方法时，不需要登录就可以调用，这就意味着对于 giftList()方法之外的其他方法调用，则必须登录后才能调用。

2. 创建 Action

创建 Action 类，实现 Session 登录权限的验证，并结合 NotNecesssaryLogin 实现特定方法的非登录操作，核心代码如下所示。

【任务 5-3】 LoginAction.java 中的核心代码

```
public String execute() {
        //检查是否需要登录
        String className = getClass().getName();
        String methodName = getMethod();
        String classMethodName = className + '#' + methodName;
        Boolean[] methodFlag = METHODS.get(classMethodName);
        HttpServletRequest request = ActionContextUtils.getRequest();
        Method method = null;
        try {
            method = getClass().getMethod(methodName);
        } catch (Exception e) {
            if (e instanceof NoSuchMethodException) {
                throw new AppException("抱歉,你访问的页面地址有误,或者该页面不存在!", e);
            }
            throw new AppException(ERROR_MSG, e);
        }
        if (methodFlag == null) {
            methodFlag = new Boolean[3];
            //1.在方法上添加注解, 此方法是否是不需要登录就可访问的
            methodFlag[0]
                    = method.isAnnotationPresent(NotNecessaryLogin.class);
            //2.是否是 AJAX 请求(通过 x-requested-with 头判断)
            String ajaxHeader = request.getHeader("x-requested-with");
            methodFlag[1] = ajaxHeader != null
                    && ajaxHeader.equalsIgnoreCase("XMLHttpRequest");
            //3.如果在类头部添加 NotNecessaryLogin 注解,
            //则该类的全部方法都不需要进行登录验证
            methodFlag[2] =
                    getClass().isAnnotationPresent(NotNecessaryLogin.class);
            METHODS.put(classMethodName, methodFlag);
        }
        if (!methodFlag[2]&&!methodFlag[0]) { //需要登录
            //没有登录或也没有记住登录状态
            if (!LoginManager.isOnline()) {
                String msg = "您还没有登录,请立即登录!";
                if (methodFlag[1]) {
                    ActionContextUtils.getResponse().setHeader("sessionstatus",
                            "timeout");
                    //Ajax 形式的未登录在 JQuery.js 中已经判断
                    this.putRootJson("msg", msg);
                    this.putRootJson("login", false);
                    writeToResponse(this.getJsonString());
                    return RESULT_NULL;
                } else {
                    request.setAttribute(MSG, msg);
                    //返回登录界面
                    return LOGIN;
```

```
                }
            }
        }

        //是否是 AJAX。super.execute()出异常时无法通过结果是否是 RESULT_AJAXJSON 判断
        boolean ajax = methodFlag[1];
        //获取处理结果
        try {
            String result = super.execute();
            String msg = "unseted";
            //如果是 Ajax 的形式
            if (RESULT_AJAXJSON.equals(result)) {
                //国际化信息返回,一般用于成功或失败的信息
                if (StringUtils.isNotEmpty(this.getInfo())) {
                    msg = getMessage(this.getInfo());
                    putRootJson(MSG, msg);   //存为根对象
                }
                if (json.entrySet().size() != 0) {
                    //如果用户使用了 json 对象
                    writeToResponse(this.getJsonString());
                } else if (!StringUtils.isEmpty(getJsonData())) {
                    //直接写到 Response 对象中
                    writeToResponse(getJsonData());
                }
                //对于 Ajax 的形式,一律返回 Ajax 的 Json 串形式
                result = RESULT_NULL;
            } else {
                //非获取信息
                if (StringUtils.isNotEmpty(this.getInfo())) {
                    msg = this.getMessage(this.getInfo());
                    //传到页面进行显示
                    this.setInfo(msg);
                }
            }
            return result;
        } catch (Exception ex) {
            …省略代码
        }
    }
```

上述代码实现了 Session 登录校验功能,并实现了基于 Ajax 响应和普通 Response 响应的两种方式,解释如下:

(1) 通过 method.isAnnotationPresent()方法查询在当前 method 方法上是否存在 NotNecessaryLogin 注解,如果存在则把 methodFlag[0]设置为 true,否则设置为 false;

(2) 通过 request.getHeader("x-requested-with")方式判断,当前请求是否是 Ajax 请求,如果是 Ajax 请求,则把 methodFlag[1]赋值为 true,否则赋值为 false;

(3) 通过 getClass().isAnnotationPresent()方式判断,当前类上是否存在 NotNecessaryLogin 注解,如果存在则把 methodFlag[2]设置为 true,否则设置为 false;

(4) 接着通过 methodFlag[2]和 methodFlag[0]的值判断当前被调用的方法是否需要进行 Session 登录校验,代码如下:

```
if (!methodFlag[2]&&!methodFlag[0]) {…}
```

如果满足以上条件，则当前被调用的方法需要进行 Session 登录校验，否则不需要进行登录验证。

注意

> 在实际应用中，可以根据具体情况通过 NotNecessaryLogin 注解来设置哪些方法不需要登录就可以操作，对于没有设置 NotNecessaryLogin 注解的方法默认情况下必须进行 Session 登录校验。

本章总结

小结

- Hibernate 是一个开放源码的 ORM(Object Relational Mapping，对象关系映射)框架
- ORM 框架是面向对象编程语言和关系型数据库之间的桥梁
- Hibernate 框架是企业 JavaEE 应用中持久层的解决方案，是一个面向对象/关系数据库映射工具，用来将对象模型映射到基于 SQL 的关系模型数据结构中
- Hibernate 中的 PO 非常简单，采用低侵入设计，完全使用 POJO(Plain Old Java Object，普通传统的 Java 对象)作为持久化对象
- Hibernate 配置文件可以是 hibernate. properties 或 hibernate. cfg. xml 这两种配置文件任选其一，或两者结合使用
- 每个配置文件对应一个 Configuration 对象，代表一个应用程序到数据库的映射配置
- Configuration 对象提供一个 buildSessionFactory( )方法，该方法可以创建一个 SessionFactory 对象
- 持久化对象的生命周期、事务的管理、对象的查询、更新和删除都是通过 Session 对象完成
- HQL(Hibernate Query Language)是完全面向对象查询语言，所操作的对象是类、实例、属性等

Q&A

1. 问题：简述 Hibernate 的优点。

回答：Hibernate 具有的优点是：开源、免费，便于研究源代码，或修改源代码进行功能定制等；轻量级封装，避免引入过多复杂问题，易调试；具有可扩展性，API 开放，根据研发需要可以自行扩展；性能稳定，具有保障。

2. 问题：简述在应用中使用 Hibernate 进行开发的 3 种方式。

回答：(1) 自底向上从数据库表到持久化类；采用手动或者开发工具根据数据库中表的结构生成对应的映射文件和持久化类。

(2) 自顶向下从持久化类到数据库表；先编写持久化类，然后手动或采用工具编写映射文件，进而生成数据库表结构。

(3) 从中间出发向上与向下同时发展；先编写映射文件，然后根据映射文件向上生成持久化类，向下生成数据库表结构。

章节练习

习题

1. 下面关于 ORM 框架的优点的说法中，错误的是_____。
   A. 贯彻面向对象的编程思想对数据库进行访问
   B. 简单易用，提高开发效率
   C. 提高访问数据库的性能，增加访问数据库的频率
   D. 具有相对独立性，发生变化时不会影响上层的实现
2. 下面对接口或类描述错误的一项是_____。
   A. Configuration 类用于配置、启动 Hibernate，创建 SessionFactory 实例对象
   B. SessionFactory 接口用于初始化 Hibernate，创建 Session 实例，充当数据源代理
   C. Session 接口用于保存、更新、删除、加载和查询持久化对象，充当持久化管理器
   D. Query 接口和 Criteria 接口都可以充当 Hibernate 查询器，其中 Criteria 用于执行 HQL 查询语句
3. 下述对持久化对象的状态描述正确的是_____。(多选)
   A. 对象由 new 关键字创建，且尚未与 Hibernate Session 关联，这时对象的状态为瞬时状态
   B. 持久化状态(Persistent)的对象与数据库中表的一条记录对应，并拥有一个持久化标识，这时该对象可以不与 Session 对象进行关联
   C. 曾经处于持久化状态，但随之与之关联的 Session 被关闭，这时对象的状态为脱管状态
   D. 处于脱管状态下的对象和瞬时状态的对象的区别是，脱管状态的对象具有一个持久化标识
4. 关于 ORM 描述错误的一项是_____。
   A. ORM 表示对象关系映射，用于在面向对象的编程语言和关系型数据库中间简化互操作的过程
   B. 常见的 ORM 技术有 Hibernate、Toplink、JPA 等
   C. ORM 框架特别适合于存在大量复杂查询，但是数据修改较少的系统
   D. 使用 ORM 框架后，基本不必再编写 SQL 语句
5. 关于 Hibernate 框架描述正确的一项是_____。
   A. Hibernate 只支持 Oracle 和 MySql 数据库
   B. Hibernate 是开源并且免费的
   C. Hibernate 框架只能用于 Web 项目
   D. Hibernate 框架可用于 ASP.NET
6. 使用 Hibernate 框架后，持久化对象具有三种状态_____、_____和_____。

上机

训练目标：Hibernate 框架应用。

培养能力	熟练使用 Hibernate 框架进行数据操作		
掌握程度	★★★★★	难度	中
代码行数	400	实施方式	编码强化
结束条件	运行不出错误		

参考训练内容

使用 Oracle 数据库和 Hibernate 框架，完成下列功能：

(1) 编写用户实体类(id，name，age，birthday)、创建数据库表；

(2) 编写 Hibernate 配置文件和映射文件；

(3) 在 main()方法中添加 10 个用户；

(4) 在 main()方法中删除 id 为偶数的用户。