

arena teaching system

Hibernate 核心



Java企业应用及互联网
高级工程师培训课程

达内集团教学研发部 编著

目 录

Unit01	1
1. Hibernate 简介	3
1.1. 什么是 Hibernate	3
1.1.1. Hibernate 的概念	3
1.1.2. Hibernate 的作用	3
1.2. 为什么要用 Hibernate	3
1.2.1. Hibernate 与 JDBC 对比	3
1.2.2. Hibernate 与 MyBatis 对比	4
2. 使用 Hibernate	4
2.1. Hibernate 框架设计原理	4
2.1.1. 设计原理	4
2.1.2. ORM 思想	4
2.2. Hibernate 框架体系结构	5
2.2.1. 主配置文件	5
2.2.2. 实体类	5
2.2.3. 映射关系文件	5
2.2.4. 底层 API	6
2.3. 如何使用 Hibernate	6
2.3.1. Hibernate 常用 API	6
2.3.2. Hibernate 使用步骤	6
2.4. Hibernate 映射类型	7
2.4.1. Java 类型	7
2.4.2. Hibernate 预定义类型	7
3. 在项目中使用 Hibernate	8
3.1. 在 NETCTOSS 中引入 Hibernate	8
3.1.1. 导包	8
3.1.2. 引入主配置文件	8
3.1.3. 创建 HibernateUtil	8
3.2. 使用 Hibernate 重构资费 DAO	9
3.2.1. 创建资费映射关系文件	9
3.2.2. 声明映射关系文件	9
3.2.3. 重构资费 DAO 实现类	9
4. Hibernate 主键生成方式	10
4.1. 常用方式	10
4.1.1. sequence	10

4.1.2. identity	10
4.1.3. native	10
4.2. 其他方式	11
4.2.1. increment	11
4.2.2. assigned	11
4.2.3. uuid/hilo	11
经典案例	12
1. Hibernate 基本使用	12
2. Hibernate 映射类型	24
3. 在 NETCTOSS 中引入 Hibernate	25
4. 使用 Hibernate 重构资费 DAO	29
课后作业	35
Unit02	36
1. 一级缓存	38
1.1. 一级缓存介绍	38
1.1.1. 什么是一级缓存	38
1.1.2. 为什么使用一级缓存	38
1.2. 使用一级缓存	38
1.2.1. 如何使用一级缓存	38
1.2.2. 一级缓存规则	39
1.2.3. 一级缓存管理	39
2. 对象持久性	40
2.1. Hibernate 中对象的状态	40
2.1.1. 对象的 3 种状态	40
2.1.2. 3 种状态的转换	40
2.2. 各种状态的规则	40
2.2.1. 临时态	40
2.2.2. 持久态	41
2.2.3. 游离态	41
2.3. 思考	41
2.3.1. Hibernate 中批量插入数据该如何处理	41
3. 延迟加载	42
3.1. 延迟加载介绍	42
3.1.1. 什么是延迟加载	42
3.1.2. 为什么使用延迟加载	42
3.2. 使用延迟加载	42
3.2.1. 采用了延迟加载的方法	42
3.2.2. 使用延迟加载需要注意的问题	43
3.2.3. Open session in view	43
3.2.4. 延迟加载实现原理	43

4. 关联映射	44
4.1. 关联映射介绍	44
4.1.1. 什么是关联映射	44
4.1.2. 关联映射的作用	44
4.1.3. 关联映射的类型	45
4.2. 如何使用关联映射	45
4.2.1. 明确 2 张表的关系及关系字段	45
4.2.2. 在实体类中添加关联属性	45
4.2.3. 在映射关系文件中配置关联关系	46
5. 一对多关联	46
5.1. 一对多关联介绍	46
5.1.1. 什么是一对多关联	46
5.1.2. 一对多关联的作用	46
5.2. 实现步骤	47
5.2.1. 明确关系字段	47
5.2.2. 在 “一” 方实体类中添加集合属性	47
5.2.3. 在 “一” 方 hbm 文件中配置关联关系	47
经典案例	48
1. 验证一级缓存	48
2. 管理一级缓存	51
3. 验证持久态对象的特性	55
4. 验证延迟加载	60
5. 在 NETCTOSS 中使用延迟加载	63
6. 使用一对多关联映射	77
课后作业	97
Unit03	98
1. 多对一关联	100
1.1. 多对一关联介绍	100
1.1.1. 什么是多对一关联	100
1.1.2. 多对一关联的作用	100
1.2. 实现步骤	100
1.2.1. 明确关系字段	100
1.2.2. 在 “多” 方实体类中添加实体属性	101
1.2.3. 在 “多” 方 hbm 文件中配置关联关系	101
2. 关联操作	101
2.1. 关联查询	101
2.1.1. 延迟加载	101
2.1.2. 抓取策略	102

2.2. 级联操作	102
2.2.1. 什么是级联操作	102
2.2.2. 级联添加/修改	102
2.2.3. 级联删除	103
2.2.4. 控制反转	103
3. Hibernate 查询	103
3.1. HQL 查询	103
3.1.1. 按条件查询	103
3.1.2. 查询一部分字段	104
3.1.3. 分页查询	104
3.1.4. 多表联合查询	105
3.2. 其他查询	106
3.2.1. 直接使用 SQL 查询	106
3.2.2. 使用 Criteria 查询	107
4. Hibernate 高级特性	107
4.1. 二级缓存	107
4.1.1. 什么是二级缓存	107
4.1.2. 如何使用二级缓存	107
4.2. 查询缓存	108
4.2.1. 什么是查询缓存	108
4.2.2. 如何使用查询缓存	108
经典案例	109
1. 使用多对一关联映射	109
2. 关联查询的一些特性	115
3. 级联添加/修改	122
4. 级联删除	131
5. HQL 查询, 按条件查询	139
6. HQL 查询, 查询一部分字段	140
7. HQL 查询, 分页查询	143
8. HQL 查询, 多表联合查询	147
9. Hibernate 中的 SQL 查询	153
10. 使用二级缓存	155
11. 使用查询缓存	163
课后作业	171

Hibernate 核心

Unit01

知识体系.....Page 3

Hibernate 简介	什么是 Hibernate	Hibernate 的概念
		Hibernate 的作用
	为什么要用 Hibernate	Hibernate 与 JDBC 对比
		Hibernate 与 MyBatis 对比
使用 Hibernate	Hibernate 框架设计原理	设计原理
		ORM 思想
	Hibernate 框架体系结构	主配置文件
		实体类
		映射关系文件
		底层 API
	如何使用 Hibernate	Hibernate 常用 API
		Hibernate 使用步骤
	Hibernate 映射类型	Java 类型
		Hibernate 预定义类型
在项目中 使用 Hibernate	在 NETCTOSS 中引入 Hibernate	导包
		引入主配置文件
		创建 HibernateUtil
	使用 Hibernate 重构资费 DAO	创建资费映射关系文件
		声明映射关系文件
		重构资费 DAO 实现类
Hibernate 主键生成方式	常用方式	sequence
		identity
		native
	其他方式	increment
		assigned
		uuid/hilo

经典案例.....Page 12

Hibernate 基本使用	Hibernate 使用步骤
----------------	----------------

Hibernate 映射类型	Hibernate 预定义类型
在 NETCTOSS 中引入 Hibernate	创建 HibernateUtil
使用 Hibernate 重构资费 DAO	重构资费 DAO 实现类

课后作业.....Page 35

1. Hibernate 简介

1.1. 什么是 Hibernate

1.1.1. 【什么是 Hibernate】Hibernate 的概念

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">Technology Tarena 达内科技</div> <h3 style="text-align: center;">Hibernate的概念</h3> <p>Hibernate是数据访问层的框架，对JDBC进行了封装，是针对数据库访问提出的面向对象的解决方案。</p> <div style="text-align: right;">+ +</div>
---	--

1.1.2. 【什么是 Hibernate】Hibernate 的作用

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">Technology Tarena 达内科技</div> <h3 style="text-align: center;">Hibernate的作用</h3> <p>使用Hibernate可以直接访问对象，Hibernate自动将此访问转换成SQL执行，从而达到间接访问数据库的目的，简化了数据访问层的代码开发。</p> <div style="text-align: right;">+ +</div>
---	--

1.2. 为什么要用 Hibernate

1.2.1. 【为什么要用 Hibernate】Hibernate 与 JDBC 对比

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">Technology Tarena 达内科技</div> <h3 style="text-align: center;">Hibernate与JDBC对比</h3> <ul style="list-style-type: none"> • 使用JDBC具有以下缺点： <ul style="list-style-type: none"> - 需要编写大量的SQL语句 - 需要给大量的？参数赋值 - 需要将ResultSet结果集转换成实体对象 - SQL中包含特有函数，无法移植 • 使用Hibernate可以解决上述问题： <ul style="list-style-type: none"> - 自动生成SQL语句 - 自动给？参数赋值 - 自动将ResultSet结果集转换成实体对象 - 采用一致的方法对数据库操作，移植性好 <div style="text-align: right;">+ +</div>
---	--

1.2.2. 【为什么要用 Hibernate】Hibernate 与 MyBatis 对比

<div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div>	<div><div>Hibernate与MyBatis对比</div><div><div>Tarena 达内科技</div></div></div> <div><ul style="list-style-type: none">• 共性<ul style="list-style-type: none">– 对JDBC进行了封装– 采用ORM思想解决了Entity和数据库的映射问题• MyBatis<ul style="list-style-type: none">– MyBatis采用SQL与Entity映射，对JDBC封装程度较轻– MyBatis自己写SQL，更具有灵活性• Hibernate<ul style="list-style-type: none">– Hibernate采用数据库与Entity映射，对JDBC封装程度较重– Hibernate自动生成SQL，对于基本的操作，开发效率高</div> <div><div>知识拓展</div><div>+</div></div>
---	--

2. 使用 Hibernate

2.1. Hibernate 框架设计原理

2.1.1. 【Hibernate 框架设计原理】设计原理


<div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div>	<div><div>设计原理</div><div><div>Tarena 达内科技</div></div></div> <div><ul style="list-style-type: none">• Hibernate采用了ORM思想对JDBC进行了封装。• Hibernate框架是ORM思想的一种实现，解决了对对象和数据库数据映射问题。• Hibernate提供一系列API，允许我们直接访问实体对象，然后其根据ORM映射关系，转换成SQL并且去执行，从而达到访问数据库的目的。</div> <div><div>知识拓展</div><div>+</div></div>
---	--

2.1.2. 【Hibernate 框架设计原理】ORM 思想

<div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div>	<div><div>ORM思想</div><div><div>Tarena 达内科技</div></div></div> <div><ul style="list-style-type: none">• ORM : Object Relation Mapping，即对象关系映射，指的是Java对象和关系数据库之间的映射。• ORM思想，就是将对象与数据库数据进行相互转换的思想，不同的框架/技术实现ORM的手段不同，但更多的是采用配置+反射的方式来实现ORM。</div> <div><div>知识拓展</div><div>+</div></div>
---	--

2.2. Hibernate 框架体系结构

2.2.1. 【Hibernate 框架体系结构】主配置文件




主配置文件

- Hibernate的主配置文件是一个XML文件，通常命名为hibernate.cfg.xml。
- 该文件中可以配置数据库连接参数、Hibernate框架参数，以及映射关系文件。

设计并实现

+

2.2.2. 【Hibernate 框架体系结构】实体类




实体类

- 实体类是与数据库表对应的Java类型，它是用于封装数据库记录的对象类型。

设计并实现

+

2.2.3. 【Hibernate 框架体系结构】映射关系文件




映射关系文件

- 映射关系文件指定了实体类和数据库表的对应关系，以及类中属性和表中字段之间的对应关系。
- Hibernate中使用XML文件来描述映射关系，文件通常命名为“实体类.hbm.xml”，并放于实体类相同的路径下。

设计并实现


+

2.2.4. 【Hibernate 框架体系结构】底层 API


<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>底层API</h3> <ul style="list-style-type: none"> • Hibernate提供了一系列的底层API，基于ORM思想，对数据库进行访问。 • 这些API主要是对映射关系文件的解析，根据解析出来的内容，动态生成SQL语句，自动将属性和字段映射。 <div style="text-align: right;">+</div>
---	--

2.3. 如何使用 Hibernate

2.3.1. 【如何使用 Hibernate】Hibernate 常用 API

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>Hibernate常用API</h3> <ul style="list-style-type: none"> • Configuration <ul style="list-style-type: none"> – 负责加载主配置文件信息，同时也加载映射关系文件信息。 • SessionFactory <ul style="list-style-type: none"> – 负责创建Session对象。 • Session <ul style="list-style-type: none"> – 数据库连接会话，负责执行增删改操作。 • Transaction <ul style="list-style-type: none"> – 负责事务控制。 • Query <ul style="list-style-type: none"> – 负责执行特殊查询。 <div style="text-align: right;">+</div>
---	--

2.3.2. 【如何使用 Hibernate】Hibernate 使用步骤

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>Hibernate使用步骤</h3> <ol style="list-style-type: none"> 1、导入Hibernate包，以及数据库驱动包。 2、引入Hibernate主配置文件hibernate.cfg.xml。 3、创建实体类。 4、创建映射关系文件。 5、使用Hibernate常用API执行增删改查操作。 <div style="text-align: right;">+</div>
---	---

2.4. Hibernate 映射类型

2.4.1. 【Hibernate 映射类型】Java 类型

Tarena
达内科技

Java类型

- Hibernate映射关系文件中，配置属性和字段关系时，可以在type属性上指定Java类型，用于做Java属性和数据库字段的转换。
- 指定Java类型时，需要写出完整的类型名，如java.lang.String。
- 自定义类型
 - 某些特殊的类型，Java预置类型无法支持，需要自定义一个类来实现，这个类要求实现接口UserType。
 - 比如布尔类型，数据库中一般存的是char(1)，一般存入y/n或者t/f。Java预置类型无法支持对布尔类型的配置，因此需要自定义UserType接口的实现类来实现。

+

2.4.2. 【Hibernate 映射类型】Hibernate 预定义类型

Tarena
达内科技

Hibernate预定义类型

- Hibernate也提供了一些类型来支持这些映射，这些类型在写法上比较简单，另外也支持所有的数据类型。
- Hibernate提供了7种映射类型，这些类型在书写时是全小写的。
- 这些映射类型以及它能够进行哪些Java类型和数据库类型的转换如下图所示。

+

Tarena
达内科技

Hibernate预定义类型（续1）

数据库类型	Java类型	数据库类型
byte	java.lang.Byte	number(m)
short	java.lang.Short	number(m)
integer	java.lang.Integer	number(m)
long	java.lang.Long	number(m)
float	java.lang.Float	number(m,n)
double	java.lang.Double	number(m,n)
string	java.lang.String	varchar
日期 (DATE)	java.util.Date	date
时间 (TIME)	java.sql.Time	date
时间戳 (TIMESTAMP)	java.sql.Timestamp	date
布尔	java.lang.Boolean	char(1)

+

3.1.1. 【在 NETCTOSS 中引入 Hibernate】导包


3.1.2. 【在 NETCTOSS 中引入 Hibernate】引入主配置文件

3.1.3. 【在 NETCTOSS 中引入 Hibernate】创建 HibernateUtil

[illegible]


3.2. 使用 Hibernate 重构资费 DAO

3.2.1. 【使用 Hibernate 重构资费 DAO】创建资费映射关系文件




创建资费映射关系文件

- 创建资费的映射关系文件Cost.hbm.xml。
- 在该文件中使用Hibernate映射类型配置实体对象属性和表中字段的关系。




3.2.2. 【使用 Hibernate 重构资费 DAO】声明映射关系文件




声明映射关系文件

- 在hibernate.cfg.xml中，声明资费的映射关系文件。




3.2.3. 【使用 Hibernate 重构资费 DAO】重构资费 DAO 实现类



重构资费DAO实现类

- 使用Hibernate的API，重构资费DAO的实现类。
- 部署并启动tomcat，访问资费模块，进行测试。



4. Hibernate 主键生成方式

4.1. 常用方式

4.1.1. 【常用方式】sequence

<div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div>	<div><div>sequence</div><div><ul style="list-style-type: none">• sequence是采用序列方式生成主键，适用于Oracle数据库。• 其配置语法为<pre><generator class="sequence"> <param name="sequence">序列名</param> </generator></pre></div><div><div>可以拖拽</div><div>+</div></div></div>
---	---

4.1.2. 【常用方式】identity



<div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div>	<div><div>identity</div><div><ul style="list-style-type: none">• identity是采用数据库自增长机制生成主键，适用于Oracle之外的其他数据库。• 其配置语法为<pre><generator class="identity"> </generator></pre></div><div><div>可以拖拽</div><div>+</div></div></div>
---	---

4.1.3. 【常用方式】native



<div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div></div>	<div><div>native</div><div><ul style="list-style-type: none">• native是根据当前配置的数据库方言，自动选择sequence或者identity。• 其配置语法为<pre><generator class="native"> <param name="sequence">序列名</param> </generator></pre></div><div><div>可以拖拽</div><div>+</div></div></div>
---	---

4.2. 其他方式



4.2.1. 【其他方式】increment

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>increment</h4> <ul style="list-style-type: none"> • increment不是采用数据库自身的机制来生成主键，而是Hibernate提供的一种生成主键的方式。它会获取当前表中主键的最大值，然后加1作为新的主键。 • 其配置语法为 <pre><generator class="increment"> </generator></pre> • 注意 <p>这种方式在并发量高时存在问题，可能会生成重复的主键，因此不推荐使用。</p> <div style="text-align: right;">  </div>
---	--

4.2.2. 【其他方式】assigned

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>assigned</h4> <ul style="list-style-type: none"> • assigned是Hibernate不负责生成主键，需要程序员自己处理主键的生成。 • 其配置语法为 <pre><generator class="assigned"> </generator></pre> <div style="text-align: right;">  </div>
---	---

4.2.3. 【其他方式】uuid/hilo

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>uuid/hilo</h4> <ul style="list-style-type: none"> • uuid/hilo是采用uuid或hilo算法生成一个主键值，这个主键值是一个不规则的长数字。 • 其配置语法为 <pre><generator class="uuid"> </generator></pre> • 注意 <p>这种方式生成的主键可以保证不重复，但是没有规律，因此不能按主键排序。</p> <div style="text-align: right;">  </div>
---	---

经典案例

1. Hibernate 基本使用

- **问题**

使用 Hibernate 实现对员工表的增、删、改、查。

- **方案**

Hibernate 使用步骤：

- 1) 导入 Hibernate 包，以及数据库驱动包。
- 2) 引入 Hibernate 主配置文件 hibernate.cfg.xml。
- 3) 创建实体类。
- 4) 创建映射关系文件。
- 5) 使用 Hibernate 常用 API 执行增删改查操作。

- **步骤**

实现此案例需要按照如下步骤进行。

步骤一：准备工作

创建员工表 EMP，建表语句如下：

```
CREATE TABLE EMP (
  ID          NUMBER(4) CONSTRAINT EMP_ID_PK PRIMARY KEY,
  NAME        VARCHAR(50) NOT NULL,
  AGE         NUMBER(11),
  SALARY      NUMBER(7,2),
  MARRY       CHAR(1),
  BIRTHDAY    DATE,
  LAST_LOGIN TIME    DATE
);

CREATE SEQUENCE emp_seq;
```

创建一个 WEB 项目，名为 HibernateDay01。

步骤二：导包

导入 Hibernate 开发包，以及数据库驱动包，完成后项目的包结构如下图：



图-1

步骤三：引入 Hibernate 主配置文件

将 Hibernate 主配置文件 hibernate.cfg.xml 复制到项目中，放在 src 根路径下。并在主配置文件中配置好数据库连接信息，以及 Hibernate 框架参数，代码如下：

```
<?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>
    <!-- 数据库连接信息，根据自己的数据库进行配置 -->
    <property name="connection.url">
      jdbc:oracle:thin:@localhost:1521:xe
    </property>
    <property name="connection.username">lhh</property>
    <property name="connection.password">123456</property>
    <property name="connection.driver class">
      oracle.jdbc.OracleDriver
    </property>

    <!-- Hibernate 配置信息 -->
    <!-- dialect 方言，用于配置生成针对哪个数据库的 SQL 语句 -->
    <property name="dialect">
      <!-- 方言类，Hibernate 提供的，用于封装某种特定数据库的方言 -->
      org.hibernate.dialect.OracleDialect
    </property>

    <!-- Hibernate 生成的 SQL 是否输出到控制台 -->
    <property name="show sql">true</property>

    <!-- 将 SQL 输出时是否格式化 -->
    <property name="format_sql">true</property>

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

步骤四：创建实体类

创建包 com.tarena.entity，并在该包下创建员工表对应的实体类 Emp.java，用于

封装员工表的数据，代码如下：

```
package com.tarena.entity;

import java.sql.Date;
import java.sql.Timestamp;

public class Emp {

    private Integer id;
    private String name;
    private Integer age;
    private Double salary;
    private Boolean marry;
    private Date birthday;
    private Timestamp lastLoginTime;

    public Integer getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

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

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    public Double getSalary() {
        return salary;
    }

    public void setSalary(Double salary) {
        this.salary = salary;
    }

    public Boolean getMarry() {
        return marry;
    }

    public void setMarry(Boolean marry) {
        this.marry = marry;
    }

    public Date getBirthday() {
        return birthday;
    }

    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }

    public Timestamp getLastLoginTime() {
```

```
        return lastLoginTime;
    }

    public void setLastLoginTime(Timestamp lastLoginTime) {
        this.lastLoginTime = lastLoginTime;
    }
}
```

步骤五：创建映射关系文件

在 `com.tarena.entity` 包下,创建员工实体类的映射关系文件,名为 `Emp.hbm.xml`,并在该文件中配置实体类和表的关系,以及类中属性和表中字段的关系,代码如下:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <!-- 配置实体类和表的关系 -->
    <class name="com.tarena.entity.Emp" table="emp">

        <!-- 配置主键属性和字段的关系 -->
        <id name="id" type="java.lang.Integer" column="id">
            <!-- 用来指明主键的生成方式 -->
            <generator class="sequence">
                <!-- 指定用于生成主键的 sequence -->
                <param name="sequence">emp_seq</param>
            </generator>
        </id>

        <!-- 配置实体类中属性与表中字段的关系 -->
        <property name="name"
            type="java.lang.String" column="name"/>
        <property name="age"
            type="java.lang.Integer" column="age"/>
        <property name="salary"
            type="java.lang.Double" column="salary"/>
        <property name="birthday"
            type="java.sql.Date" column="birthday"/>
        <property name="lastLoginTime"
            type="java.sql.Timestamp" column="last_login_time"/>

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

注意：先不要配置布尔型属性 `marry`, 后面会单独讲解布尔型属性的配置及使用。

步骤六：声明映射关系文件

在主配置文件 `hibernate.cfg.xml` 中,声明映射关系文件 `Emp.hbm.xml`,代码如下:

```
<?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>
        <!-- 数据库连接信息, 根据自己的数据库进行配置 -->
        <property name="connection.url">
```

```

        jdbc:oracle:thin:@localhost:1521:xe
    </property>
    <property name="connection.username">lhh</property>
    <property name="connection.password">123456</property>
    <property name="connection.driver_class">
        oracle.jdbc.OracleDriver
    </property>

    <!-- Hibernate 配置信息 -->
    <!-- dialect 方言, 用于配置生成针对哪个数据库的 SQL 语句 -->
    <property name="dialect">
        <!-- 方言类, Hibernate 提供的, 用于封装某种特定数据库的方言 -->
        org.hibernate.dialect.OracleDialect
    </property>
    <!-- Hibernate 生成的 SQL 是否输出到控制台 -->
    <property name="show_sql">true</property>
    <!-- 将 SQL 输出时是否格式化 -->
    <property name="format_sql">true</property>

    <!-- 声明映射关系文件 -->
    <mapping resource="com/tarena/entity/Emp.hbm.xml" />

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

```

步骤七：创建 Session 工具类

创建 HibernateUtil 工具类，用于创建 Session 对象，代码如下：

```

package com.tarena.util;

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

public class HibernateUtil {

    private static SessionFactory sessionFactory;

    static {
        // 加载 Hibernate 主配置文件
        Configuration conf = new Configuration();
        conf.configure("/hibernate.cfg.xml");
        sessionFactory = conf.buildSessionFactory();
    }

    /**
     * 创建 session
     */
    public static Session getSession() {
        return sessionFactory.openSession();
    }

    public static void main(String[] args) {
        System.out.println(getSession());
    }

}

```

步骤八：练习使用 Hibernate 对员工表进行增删改查

创建包 `com.tarena.test`，并在该包下创建一个 `JUNIT` 测试类，并在类中使用 `Hibernate` 写出对 `EMP` 表的增删改查的方法，代码如下：

```
package com.tarena.test;

import java.sql.Date;
import java.sql.Timestamp;
import java.util.List;
import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.junit.Test;
import com.tarena.entity.Emp;
import com.tarena.util.HibernateUtil;

/**
 * 演示 Hibernate 的基本使用
 */
public class TestEmp {

    /**
     * 新增 emp
     */
    @Test
    public void add() {
        // 模拟要新增的 emp
        Emp e = new Emp();
        e.setName("ggg");
        e.setAge(29);
        e.setMarry(false);
        e.setSalary(8000.00);
        e.setBirthday(Date.valueOf("1983-10-20"));
        e.setLastLoginTime(
            new Timestamp(System.currentTimeMillis()));

        // 获取 session
        Session session = HibernateUtil.getSession();
        // 开启事务
        Transaction ts = session.beginTransaction();
        try {
            // 执行新增
            session.save(e);
            // 提交事务
            ts.commit();
        } catch (HibernateException e1) {
            e1.printStackTrace();
            // 回滚事务
            ts.rollback();
        } finally {
            session.close();
        }
    }

    /**
     * 根据 ID 查询 emp
     */
    @Test
    public void findById() {
```

```
Session session = HibernateUtil.getSession();
Emp emp = (Emp) session.get(Emp.class, 201);
System.out.println(emp.getName());
System.out.println(emp.getBirthDay());
System.out.println(emp.getLastLoginTime());
session.close();
}

/**
 * 修改 emp
 */
@Test
public void update() {
    Session session = HibernateUtil.getSession();
    // 查询要修改的数据
    Emp emp = (Emp) session.get(Emp.class, 42);
    // 开启事务
    Transaction ts = session.beginTransaction();
    try {
        // 模拟修改数据
        emp.setName("ee");
        emp.setAge(20);
        // 执行修改
        session.update(emp);
        // 提交事务
        ts.commit();
    } catch (HibernateException e) {
        e.printStackTrace();
        // 回滚事务
        ts.rollback();
    } finally {
        // 关闭连接
        session.close();
    }
}

/**
 * 删除 emp
 */
@Test
public void delete() {
    Session session = HibernateUtil.getSession();
    Emp emp = (Emp) session.get(Emp.class, 148);
    Transaction ts = session.beginTransaction();
    try {
        session.delete(emp);
        ts.commit();
    } catch (HibernateException e) {
        e.printStackTrace();
        ts.rollback();
    } finally {
        session.close();
    }
}

/**
 * 查询全部 emp
 */
@Test
public void findAll() {
    String hql = "from Emp";
    Session session = HibernateUtil.getSession();
    Query query = session.createQuery(hql);
```

```
List<Emp> emps = query.list();
for (Emp e : emps) {
    System.out.println(e.getName());
}
session.close();
}
}
```

步骤九：测试

以 JUNIT 方式运行上面的每一个测试方法，并观察控制台输出的 SQL。

• 完整代码

以下是本案例的完整代码。

其中建表语句完整代码如下：

```
CREATE TABLE EMP(
    ID          NUMBER(4) CONSTRAINT EMP ID PK PRIMARY KEY,
    NAME        VARCHAR(50) NOT NULL,
    AGE         NUMBER(11),
    SALARY       NUMBER(7,2),
    MARRY       CHAR(1),
    BIRTHDAY     DATE,
    LAST_LOGIN_TIME DATE
);

CREATE SEQUENCE emp_seq;
```

主配置文件 hibernate.cfg.xml 完整代码如下：

```
<?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>
        <!-- 数据库连接信息，根据自己的数据库进行配置 -->
        <property name="connection.url">
            jdbc:oracle:thin:@localhost:1521:xe
        </property>
        <property name="connection.username">lhh</property>
        <property name="connection.password">123456</property>
        <property name="connection.driver class">
            oracle.jdbc.OracleDriver
        </property>

        <!-- Hibernate 配置信息 -->
        <!-- dialect 方言，用于配置生成针对哪个数据库的 SQL 语句 -->
        <property name="dialect">
            <!-- 方言类，Hibernate 提供的，用于封装某种特定数据库的方言 -->
            org.hibernate.dialect.OracleDialect
        </property>

        <!-- Hibernate 生成的 SQL 是否输出到控制台 -->
        <property name="show_sql">true</property>
        <!-- 将 SQL 输出时是否格式化 -->
        <property name="format_sql">true</property>

        <!-- 声明映射关系文件 -->
        <mapping resource="com/tarena/entity/Emp.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```



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

实体类 Emp.java 完整代码如下：

```
package com.tarena.entity;

import java.sql.Date;
import java.sql.Timestamp;

public class Emp {

    private Integer id;
    private String name;
    private Integer age;
    private Double salary;
    private Boolean marry;
    private Date birthday;
    private Timestamp lastLoginTime;

    public Integer getId() {
        return id;
    }

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

    public String getName() {
        return name;
    }

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

    public Integer getAge() {
        return age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

    public Double getSalary() {
        return salary;
    }

    public void setSalary(Double salary) {
        this.salary = salary;
    }

    public Boolean getMarry() {
        return marry;
    }

    public void setMarry(Boolean marry) {
        this.marry = marry;
    }

    public Date getBirthday() {
        return birthday;
    }

    public void setBirthday(Date birthday) {
        this.birthday = birthday;
    }
}
```

```

    }

    public Timestamp getLastLoginTime() {
        return lastLoginTime;
    }

    public void setLastLoginTime(Timestamp lastLoginTime) {
        this.lastLoginTime = lastLoginTime;
    }
}

```

映射关系文件 Emp.hbm.xml 完整代码如下：

```

<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <!-- 配置实体类和表的关系 -->
    <class name="com.tarena.entity.Emp" table="emp">
        <!-- 配置主键属性和字段的关系 -->
        <id name="id" type="java.lang.Integer" column="id">
            <!-- 用来指明主键的生成方式 -->
            <generator class="sequence">
                <!-- 指定用于生成主键的 sequence -->
                <param name="sequence">emp_seq</param>
            </generator>
        </id>

        <!-- 配置实体类中属性与表中字段的关系 -->
        <property name="name"
            type="java.lang.String" column="name"/>
        <property name="age"
            type="java.lang.Integer" column="age"/>
        <property name="salary"
            type="java.lang.Double" column="salary"/>
        <property name="birthday"
            type="java.sql.Date" column="birthday"/>
        <property name="lastLoginTime"
            type="java.sql.Timestamp" column="last_login_time"/>
    </class>
</hibernate-mapping>

```

HibernateUtil 完整代码如下：

```

package com.tarena.util;

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

public class HibernateUtil {
    private static SessionFactory sessionFactory;

    static {
        // 加载 Hibernate 主配置文件
        Configuration conf = new Configuration();
        conf.configure("/hibernate.cfg.xml");
        sessionFactory = conf.buildSessionFactory();
    }

    /**

```

```
* 创建 session
*/
public static Session getSession() {
    return sessionFactory.openSession();
}

public static void main(String[] args) {
    System.out.println(getSession());
}
}
```

JUNIT 测试类代码如下：

```
package com.tarena.test;

import java.sql.Date;
import java.sql.Timestamp;
import java.util.List;
import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.junit.Test;
import com.tarena.entity.Emp;
import com.tarena.util.HibernateUtil;

/**
 * 演示 Hibernate 的基本使用
 */
public class TestEmp {

    /**
     * 新增 emp
     */
    @Test
    public void add() {
        // 模拟要新增的 emp
        Emp e = new Emp();
        e.setName("ggg");
        e.setAge(29);
        e.setMarry(false);
        e.setSalary(8000.00);
        e.setBirthday(Date.valueOf("1983-10-20"));
        e.setLastLoginTime(
            new Timestamp(System.currentTimeMillis()));

        // 获取 session
        Session session = HibernateUtil.getSession();
        // 开启事务
        Transaction ts = session.beginTransaction();
        try {
            // 执行新增
            session.save(e);
            // 提交事务
            ts.commit();
        } catch (HibernateException e1) {
            e1.printStackTrace();
            // 回滚事务
            ts.rollback();
        } finally {
            session.close();
        }
    }
}
```

```

/**
 * 根据 ID 查询 emp
 */
@Test
public void findById() {
    Session session = HibernateUtil.getSession();
    Emp emp = (Emp) session.get(Emp.class, 301);
    System.out.println(emp.getName());
    System.out.println(emp.getBirthDay());
    System.out.println(emp.getLastLoginTime());
    session.close();
}

/**
 * 修改 emp
 */
@Test
public void update() {
    Session session = HibernateUtil.getSession();
    // 查询要修改的数据
    Emp emp = (Emp) session.get(Emp.class, 301);
    // 开启事务
    Transaction ts = session.beginTransaction();
    try {
        // 模拟修改数据
        emp.setName("eee");
        emp.setAge(20);
        // 执行修改
        session.update(emp);
        // 提交事务
        ts.commit();
    } catch (HibernateException e) {
        e.printStackTrace();
        // 回滚事务
        ts.rollback();
    } finally {
        // 关闭连接
        session.close();
    }
}

/**
 * 删除 emp
 */
@Test
public void delete() {
    Session session = HibernateUtil.getSession();
    Emp emp = (Emp) session.get(Emp.class, 301);
    Transaction ts = session.beginTransaction();
    try {
        session.delete(emp);
        ts.commit();
    } catch (HibernateException e) {
        e.printStackTrace();
        ts.rollback();
    } finally {
        session.close();
    }
}

/**
 * 查询全部 emp
 */
@Test
public void findAll() {
    String hql = "from Emp";

```

```
Session session = HibernateUtil.getSession();
Query query = session.createQuery(hql);
List<Emp> emps = query.list();
for (Emp e : emps) {
    System.out.println(e.getName());
}
session.close();
}
}
```

2. Hibernate 映射类型

- 问题

使用 Hibernate 预定义类型配置属性和字段的映射关系。

- 方案

将项目 HibernateDay01 中，Emp.hbm.xml 里面每个属性的配置，其 type 用 Hibernate 预定义类型来重构。由于 Hibernate 预定义类型支持布尔值，因此在该配置文件中追加 marry 属性的配置。

- 步骤

实现此案例需要按照如下步骤进行。

步骤一：重构 Emp.hbm.xml

重构 Emp.hbm.xml，将各个属性的 type 由 Java 类型改为 Hibernate 预定义类型，同时追加 marry 属性，并使用布尔型来配置该属性，代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <!-- 配置实体类和表的关系 -->
    <class name="com.tarena.entity.Emp" table="emp">
        <!-- 配置主键属性和字段的关系 -->

        <id name="id" type="integer" column="id">

            <!-- 用来指明主键的生成方式 -->
            <generator class="sequence">
                <!-- 指定用于生成主键的 sequence -->
                <param name="sequence">emp seq</param>
            </generator>

        </id>

        <!-- 配置实体类中属性与表中字段的关系 -->
```

```
<property name="name" type="string" column="name"/>
<property name="age" type="integer" column="age"/>
<property name="salary" type="double" column="salary"/>
<property name="birthday" type="date" column="birthday"/>
<property name="lastLoginTime"
    type="timestamp" column="last_login_time"/>
<property name="marry" type="yes_no" column="marry"/>

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

步骤二：测试

再次以 JUNIT 方式执行测试代码中的每个方法，看控制台输出结果是否正确。

• 完整代码

以下是本案例的完整代码。

其中 Emp.hbm.xml 完整代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<!-- 配置实体类和表的关系 -->
<class name="com.tarena.entity.Emp" table="emp">
    <!-- 配置主键属性和字段的关系 -->
    <id name="id" type="integer" column="id">
        <!-- 用来指明主键的生成方式 -->
        <generator class="sequence">
            <!-- 指定用于生成主键的 sequence -->
            <param name="sequence">emp_seq</param>
        </generator>
    </id>

    <!-- 配置实体类中属性与表中字段的关系 -->
    <property name="name" type="string" column="name"/>
    <property name="age" type="integer" column="age"/>
    <property name="salary" type="double" column="salary"/>
    <property name="birthday" type="date" column="birthday"/>
    <property name="lastLoginTime"
        type="timestamp" column="last_login_time"/>
    <property name="marry" type="yes no" column="marry"/>
</class>
</hibernate-mapping>
```

其他代码不变，此处不再重复。

3. 在 NETCTOSS 中引入 Hibernate

• 问题

将 Hibernate 框架引入到 NETCTOSS。

• 方案

将 Hibernate 框架引入到 NETCTOSS，以保证在项目可以使用 Hibernate 来进行数据库访问。根据 Hibernate 的使用步骤，我们需要在项目中引入 Hibernate 开发包、主配置文件，并创建 HibernateUtil 工具类。

• 步骤

实现此案例需要按照如下步骤进行。

步骤一：导包

在 NETCTOSS 项目中，导入 Hibernate 开发包，导入后项目的包结构如下图：

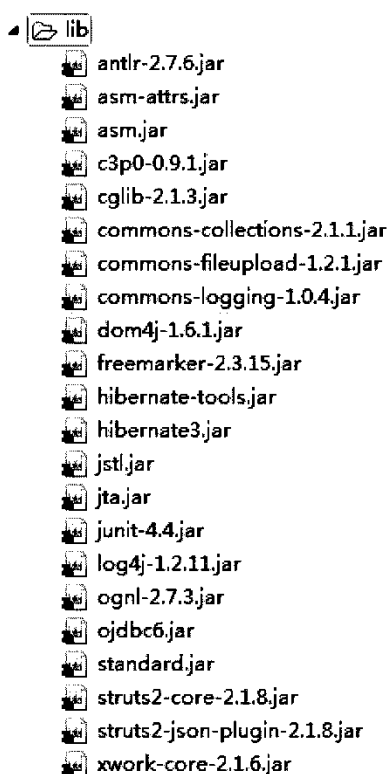


图-2

步骤二：引入主配置文件

将 Hibernate 主配置文件引入到项目中，放于 src 根路径下，该文件可以从 HibernateDay01 项目中直接复制过来，并将声明映射关系文件部分代码去掉，代码如下：

```
<?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>
    <!-- 数据库连接信息，根据自己的数据库进行配置 -->
    <property name="connection.url">
      jdbc:oracle:thin:@localhost:1521:xe
    </property>
```

```
<property name="connection.username">lhh</property>
<property name="connection.password">123456</property>
<property name="connection.driver_class">
    oracle.jdbc.OracleDriver
</property>

<!-- Hibernate 配置信息 -->
<!-- dialect 方言，用于配置生成针对哪个数据库的 SQL 语句 -->
<property name="dialect">
    <!-- 方言类，Hibernate 提供的，用于封装某种特定数据库的方言 -->
    org.hibernate.dialect.OracleDialect
</property>
<!-- Hibernate 生成的 SQL 是否输出到控制台 -->
<property name="show_sql">true</property>
<!-- 将 SQL 输出时是否格式化 -->
<property name="format_sql">true</property>

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

步骤三：创建 HibernateUtil 工具类

在 com.netctoss.util 包下，创建 HibernateUtil 工具类，该类可以直接从 HibernateDay01 项目中复制过来，代码如下：

```
package com.netctoss.util;

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

public class HibernateUtil {
    private static SessionFactory sessionFactory;
    static {
        // 加载 Hibernate 主配置文件
        Configuration conf = new Configuration();
        conf.configure("/hibernate.cfg.xml");
        sessionFactory = conf.buildSessionFactory();
    }

    /**
     * 创建 session
     */
    public static Session getSession() {
        return sessionFactory.openSession();
    }

    public static void main(String[] args) {
        System.out.println(getSession());
    }
}
```

• 完整代码

下面是本案例的完整代码。

其中主配置文件完整代码如下：

```
<?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>
    <!-- 数据库连接信息，根据自己的数据库进行配置 -->
    <property name="connection.url">
        jdbc:oracle:thin:@localhost:1521:xe
    </property>
    <property name="connection.username">lhh</property>
    <property name="connection.password">123456</property>
    <property name="connection.driver_class">
        oracle.jdbc.OracleDriver
    </property>

    <!-- Hibernate 配置信息 -->
    <!-- dialect 方言，用于配置生成针对哪个数据库的 SQL 语句 -->
    <property name="dialect">
        <!-- 方言类，Hibernate 提供的，用于封装某种特定数据库的方言 -->
        org.hibernate.dialect.OracleDialect
    </property>

    <!-- Hibernate 生成的 SQL 是否输出到控制台 -->
    <property name="show_sql">true</property>
    <!-- 将 SQL 输出时是否格式化 -->
    <property name="format_sql">true</property>

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

HibernateUtil 工具类完整代码如下：

```
package com.netctoss.util;

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

public class HibernateUtil {
    private static SessionFactory sessionFactory;

    static {
        // 加载 Hibernate 主配置文件
        Configuration conf = new Configuration();
        conf.configure("/hibernate.cfg.xml");
        sessionFactory = conf.buildSessionFactory();
    }

    /**
     * 创建 session
     */
    public static Session getSession() {
        return sessionFactory.openSession();
    }

    public static void main(String[] args) {
        System.out.println(getSession());
    }
}
```

4. 使用 Hibernate 重构资费 DAO

• 问题

使用 Hibernate 重构资费 DAO 的实现类。

• 方案

使用 Hibernate 的增删改查 API，重构资费 DAO 的实现类。

• 步骤

实现此案例需要按照如下步骤进行。

步骤一：创建资费的映射关系文件

在 com.netctoss.entity 包下，创建资费的映射关系文件，名为 Cost.hbm.xml，代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<!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.netctoss.entity.Cost" table="cost">
    <id name="id" type="integer" column="id">
        <!-- 用来指明主键的生成方式 -->
        <generator class="sequence">
            <param name="sequence">cost_seq</param>
        </generator>
    </id>

    <property name="name"
        type="string" column="name" />
    <property name="baseDuration"
        type="integer" column="base duration" />
    <property name="baseCost"
        type="double" column="base_cost" />
    <property name="unitCost"
        type="double" column="unit cost" />
    <property name="status"
        type="string" column="status" />
    <property name="descr"
        type="string" column="descr" />
    <property name="createTime"
        type="date" column="create time" />
    <property name="startTime"
        type="date" column="start time" />
    <property name="costType"
        type="string" column="cost_type" />
</class>
</hibernate-mapping>
```

步骤二：声明映射关系文件

在 hibernate.cfg.xml 中声明映射关系文件，代码如下：

```
<?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>
    <!-- 数据库连接信息，根据自己的数据库进行配置 -->
    <property name="connection.url">
        jdbc:oracle:thin:@localhost:1521:xe
    </property>
    <property name="connection.username">lhh</property>
    <property name="connection.password">123456</property>
    <property name="connection.driver class">
        oracle.jdbc.OracleDriver
    </property>

    <!-- Hibernate 配置信息 -->
    <!-- dialect 方言，用于配置生成针对哪个数据库的 SQL 语句 -->
    <property name="dialect">
        <!-- 方言类，Hibernate 提供的，用于封装某种特定数据库的方言 -->
        org.hibernate.dialect.OracleDialect
    </property>

    <!-- Hibernate 生成的 SQL 是否输出到控制台 -->
    <property name="show sql">true</property>
    <!-- 将 SQL 输出时是否格式化 -->
    <property name="format sql">true</property>

    <!-- 在配置文件中关联映射文件 -->
    <mapping resource="com/netctoss/entity/Cost.hbm.xml" />

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

步骤三：使用 Hibernate 重构 CostDaoImpl

使用 HibernateUtil 的增删改查 API，来重构 CostDaoImpl，代码如下：

```
package com.netctoss.dao;

import java.util.List;
import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;
import com.netctoss.entity.Cost;
import com.netctoss.util.HibernateUtil;

/**
 * 当前阶段学习重点是 Struts2，对于 DAO 的实现就模拟实现了。
 * 同学们可以使用 JDBC/MyBatis 自行实现该 DAO。
 */
public class CostDaoImpl implements ICostDao {

    @Override
    public List<Cost> findAll() {

        String hql = "from Cost";
        Session session = HibernateUtil.getSession();
```

```

        Query query = session.createQuery(hql);
        List<Cost> list = query.list();
        session.close();
        return list;
    }

    @Override
    public void delete(int id) {

        Cost cost = new Cost();
        cost.setId(id);

        Session session = HibernateUtil.getSession();
        Transaction ts = session.beginTransaction();
        try {
            session.delete(cost);
            ts.commit();
        } catch (HibernateException e) {
            e.printStackTrace();
            ts.rollback();
        } finally {
            session.close();
        }
    }

    @Override
    public Cost findByName(String name) {
        // 模拟根据名称查询资费数据，假设资费表中只有一条名为 tarena 的数据
        if("tarena".equals(name)) {
            Cost c = new Cost();
            c.setId(97);
            c.setName("tarena");
            c.setBaseDuration(99);
            c.setBaseCost(9.9);
            c.setUnitCost(0.9);
            c.setDescr("tarena 套餐");
            c.setStatus("0");
            c.setCostType("2");
            return c;
        }
        return null;
    }

    @Override
    public Cost findById(int id) {

        Session session = HibernateUtil.getSession();
        Cost cost = (Cost) session.get(Cost.class, id);
        session.close();
        return cost;
    }
}

```

注意：目前我们还没学习有条件的查询，因此先不重构 `findByName` 方法。

步骤四：测试

重新部署项目，并重启 tomcat，访问资费模块，对其各功能进行测试。

• 完整代码

下面是本案例的完整代码。

其中映射关系文件完整代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<!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.netctoss.entity.Cost" table="cost">
<id name="id" type="integer" column="id">
<!-- 用来指明主键的生成方式 -->
<generator class="sequence">
<param name="sequence">cost seq</param>
</generator>
</id>

<property name="name"
type="string" column="name" />
<property name="baseDuration"
type="integer" column="base duration" />
<property name="baseCost"
type="double" column="base_cost" />
<property name="unitCost"
type="double" column="unit_cost" />
<property name="status"
type="string" column="status" />
<property name="descr"
type="string" column="descr" />
<property name="createTime"
type="date" column="create time" />
<property name="startTime"
type="date" column="start time" />
<property name="costType"
type="string" column="cost_type" />
</class>
</hibernate-mapping>
```

主配置文件完整代码如下：

```
<?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>
<!-- 数据库连接信息，根据自己的数据库进行配置 -->
<property name="connection.url">
jdbc:oracle:thin:@localhost:1521:xe
</property>
<property name="connection.username">lh</property>
<property name="connection.password">123456</property>
<property name="connection.driver_class">
oracle.jdbc.OracleDriver
</property>

<!-- Hibernate 配置信息 -->
<!-- dialect 方言，用于配置生成针对哪个数据库的 SQL 语句 -->
```

```
<property name="dialect">
    <!-- 方言类, Hibernate 提供的, 用于封装某种特定数据库的方言 -->
    org.hibernate.dialect.OracleDialect
</property>
<!-- Hibernate 生成的 SQL 是否输出到控制台 -->
<property name="show_sql">true</property>
<!-- 将 SQL 输出时是否格式化 -->
<property name="format_sql">true</property>

<!-- 在配置文件中关联映射文件 -->
<mapping resource="com/netctoss/entity/Cost.hbm.xml" />
</session-factory>
</hibernate-configuration>
```

CostDaoImpl 完整代码如下：

```
package com.netctoss.dao;

import java.util.List;
import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;
import com.netctoss.entity.Cost;
import com.netctoss.util.HibernateUtil;

/**
 * 当前阶段学习重点是 Struts2, 对于 DAO 的实现就模拟实现了。
 * 同学们可以使用 JDBC/MyBatis 自行实现该 DAO。
 */
public class CostDaoImpl implements ICostDao {

    @Override
    public List<Cost> findAll() {
        String hql = "from Cost";
        Session session = HibernateUtil.getSession();
        Query query = session.createQuery(hql);
        List<Cost> list = query.list();
        session.close();
        return list;
    }

    @Override
    public void delete(int id) {
        Cost cost = new Cost();
        cost.setId(id);

        Session session = HibernateUtil.getSession();
        Transaction ts = session.beginTransaction();
        try {
            session.delete(cost);
            ts.commit();
        } catch (HibernateException e) {
            e.printStackTrace();
            ts.rollback();
        } finally {
            session.close();
        }
    }

    @Override
    public Cost findByName(String name) {
        // 模拟根据名称查询资费数据, 假设资费表中只有一条名为 tarena 的数据
        if("tarena".equals(name)) {
            Cost c = new Cost();

```

```
        c.setId(97);
        c.setName("tarena");
        c.setBaseDuration(99);
        c.setBaseCost(9.9);
        c.setUnitCost(0.9);
        c.setDescr("tarena 套餐");
        c.setStatus("0");
        c.setCostType("2");
        return c;
    }
    return null;
}

@Override
public Cost findById(int id) {
    Session session = HibernateUtil.getSession();
    Cost cost = (Cost) session.get(Cost.class, id);
    session.close();
    return cost;
}
}
```

课后作业

1. Hibernate 有哪些缺点
2. 请简述 Hibernate 框架设计原理
3. 请简述如何使用 Hibernate 访问数据库

Hibernate 核心

Unit02

知识体系.....Page 38

一级缓存	一级缓存介绍	什么是一级缓存
		为什么使用一级缓存
	使用一级缓存	如何使用一级缓存
		一级缓存规则
		一级缓存管理
对象持久性	Hibernate 中对象的状态	对象的 3 种状态
		3 种状态的转换
	各种状态的规则	临时态
		持久态
		游离态
	思考	Hibernate 中批量插入数据该如何处理
延迟加载	延迟加载介绍	什么是延迟加载
		为什么使用延迟加载
	使用延迟加载	采用了延迟加载的方法
		使用延迟加载需要注意的问题
		Open session in view
关联映射	关联映射介绍	什么是关联映射
		关联映射的作用
		关联映射的类型
	如何使用关联映射	明确 2 张表的关系及关系字段
		在实体类中添加关联属性
一对多关联	一对多关联介绍	什么是一对多关联
		一对多关联的作用
	实现步骤	明确关系字段
		在 “一” 方实体类中添加集合属性
		在 “一” 方 hbm 文件中配置关联关系

经典案例.....Page 48

验证一级缓存	一级缓存规则
管理一级缓存	一级缓存管理
验证持久态对象的特性	游离态
验证延迟加载	采用了延迟加载的方法
在 NETCTOSS 中使用延迟加载	Open session in view
使用一对多关联映射	在 “一” 方映射关系文件中配置关联关系

课后作业.....Page 97

1. 一级缓存

1.1. 一级缓存介绍

1.1.1. 【一级缓存介绍】什么是一级缓存

知识讲解

什么是一级缓存

- Hibernate创建每个Session对象时，都会给该Session分配一块独立的缓存区，用于存放该Session查询出来的对象，这个分配给Session的缓存区称之为一级缓存，也叫Session级缓存。

+

+

1.1.2. 【一级缓存介绍】为什么使用一级缓存

知识讲解

为什么使用一级缓存

- Session取数据时，会优先向缓存区取数据，如果存在数据则直接返回，不存在才会去数据库查询，从而降低了数据库访问次数，提升了代码运行效率。

+

+

1.2. 使用一级缓存

1.2.1. 【使用一级缓存】如何使用一级缓存

知识讲解

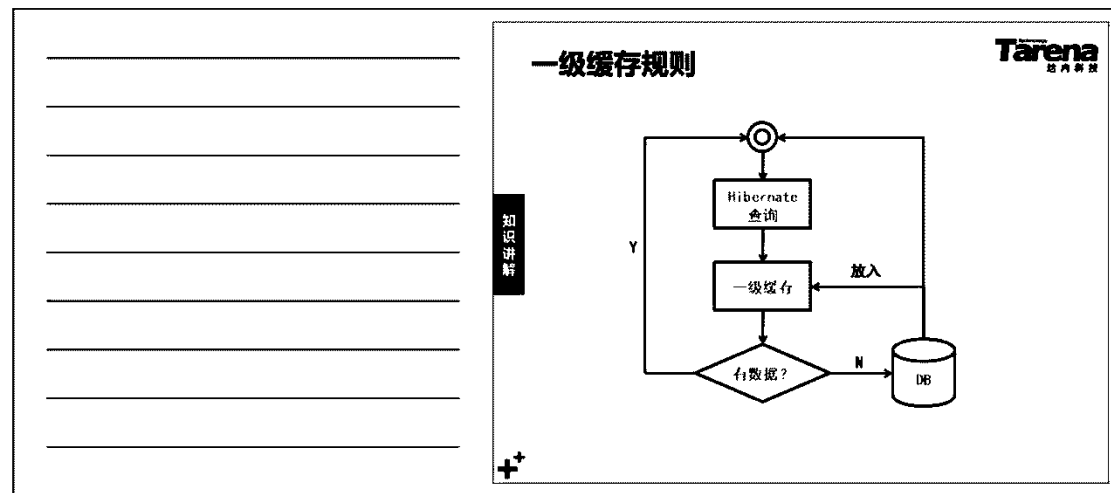
如何使用一级缓存

- 一级缓存是默认开启的，在使用Hibernate的API进行查询时会自动使用。

+

+

1.2.2. 【使用一级缓存】一级缓存规则



Tarena
达内科技

一级缓存规则（续1）

- 一级缓存是Session独享的，每个Session不能访问其他Session的缓存区。
- Session的save、update、delete操作会触发缓存更新。

知识讲解

++

1.2.3. 【使用一级缓存】一级缓存管理

Tarena
达内科技

一级缓存管理

- session.evict(obj);
将obj从一级缓存中移除。
- session.clear();
清除一级缓存中所有的obj。
- session.close();
关闭Session，释放缓存空间。

知识讲解

++

2. 对象持久性

2.1. Hibernate 中对象的状态

2.1.1. 【Hibernate 中对象的状态】对象的 3 种状态

Tarena
达内科技

对象的3种状态

- 在Hibernate中，可以把实体对象看成有3种状态，分别是临时态、持久态、游离态。

知识点

+ +

2.1.2. 【Hibernate 中对象的状态】3 种状态的转换

Tarena
达内科技

3种状态的转换

```

graph TD
    Start(( )) -- new --> Transient[临时态]
    Transient -- save() --> Persistent[持久态]
    Persistent -- update() --> Persistent
    Persistent -- delete() --> Transient
    Persistent -- evict() --> Detached[游离态]
    Persistent -- clear() --> Detached
    Persistent -- close() --> Detached
    Detached -- save() --> Persistent
    Detached -- update() --> Persistent
    Transient -.->|garbage| End((( )))
    Detached -.->|garbage| End
    
```

知识点

+ +

2.2. 各种状态的规则

2.2.1. 【各种状态的规则】临时态

Tarena
达内科技

临时态

- 转换
 - 通过new创建的对象为临时态
 - 通过delete方法操作的对象将转变为临时态
- 特征
 - 临时态的对象可以被垃圾回收
 - 临时态的对象未进行过持久化，未与session关联

知识点

+ +

2.2.2. 【各种状态的规则】持久态

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">Tarena 达内科技</div> <h4>持久态</h4> <ul style="list-style-type: none"> • 转换 <ul style="list-style-type: none"> – 通过get、load、list、iterate方法查询到的对象为持久态 – 通过save、update方法操作的对象转变为持久态 • 特征 <ul style="list-style-type: none"> – 持久态对象垃圾回收器不能回收 – 持久态的对象进行了持久化，与session相关联。实际上持久态对象存在于session缓存中，由session负责管理。 – 持久态对象的数据可以自动更新到数据库中，时机是在调用session.flush()时执行。而提交事务时会调用session.flush()，因此提交事务时也会触发同步，可以理解为ts.commit=session.flush()+commit <div style="text-align: right;">++</div>
---	--

2.2.3. 【各种状态的规则】游离态

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">Tarena 达内科技</div> <h4>游离态</h4> <ul style="list-style-type: none"> • 转换 <ul style="list-style-type: none"> – 通过session的evict、clear、close方法操作的对象会转变为游离态 • 特征 <ul style="list-style-type: none"> – 游离态的对象可以被垃圾回收 – 游离态的对象进行过持久化，但已与Session解除了关联 <div style="text-align: right;">++</div>
---	--

2.3. 思考

2.3.1. 【思考】Hibernate 中批量插入数据该如何处理

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">Tarena 达内科技</div> <h4>Hibernate中批量插入数据该如何处理</h4> <ul style="list-style-type: none"> • 如果存在下面的批量插入数据的方法，假设数据超过10万行，应该如何避免一级缓存溢出的情况？ <pre> public void batchAdd(List<Emp> emps) { Session session = HibernateUtil.getSession(); Transaction ts = session.beginTransaction(); for(Emp e : emps) { session.save(e); } ts.commit(); HibernateUtil.close(); } </pre> <div style="text-align: right;">++</div>
---	--

3. 延迟加载

3.1. 延迟加载介绍

3.1.1. 【延迟加载介绍】什么是延迟加载

知识讲解

什么是延迟加载

- 在使用某些Hibernate方法查询数据时，Hibernate返回的只是一个空对象(除id外属性都为null)，并没有真正查询数据库。而在用这个对象时才会触发查询数据库，并将查询到的数据注入到这个空对象中。这种将查询时机推迟到对象访问时的机制称之为延迟加载。

3.1.2. 【延迟加载介绍】为什么使用延迟加载

知识讲解

为什么使用延迟加载

- 可以提升内存资源的使用率
- 可以降低对数据库的访问次数

3.2. 使用延迟加载



3.2.1. 【使用延迟加载】采用了延迟加载的方法

知识讲解



采用了延迟加载的方法

- session.load()
- query.iterate()
- 关联映射中对关联属性的加载



3.2.2. 【使用延迟加载】使用延迟加载需要注意的问题


<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4 style="text-align: center;">使用延迟加载需要注意的问题</h4> <ul style="list-style-type: none"> • 采用具有延迟加载机制的操作，需要避免session提前关闭，避免在使用对象之前关闭session。 • 可以采用以下2种方案解决此问题 <ul style="list-style-type: none"> – 采用非延迟加载的查询方法，如query.get()、session.list()等。 – 在使用对象之后再关闭session。 <div style="text-align: right;">  </div>
---	---

3.2.3. 【使用延迟加载】Open session in view

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4 style="text-align: center;">Open session in view</h4> <ul style="list-style-type: none"> • 在项目中，DAO只是负责查询出数据，而使用数据的时机是在JSP解析的过程中，因此要避免在DAO中关闭session，或者说要在视图层保持session的开启。 • 项目中解决这个问题的手段称之为Open session in view，即在视图层保持session的开启。 • 在不同的技术框架下，实现Open session in view的手段不同： <ul style="list-style-type: none"> – 在Servlet中使用过滤器实现。 – 在Struts2中使用拦截器实现。 – 在Spring中使用AOP实现。 <div style="text-align: right;">  </div>
---	---

3.2.4. 【使用延迟加载】延迟加载实现原理


<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4 style="text-align: center;">延迟加载实现原理</h4> <ul style="list-style-type: none"> • 采用延迟加载方法，返回的对象类型是Hibernate采用CGLIB技术在内存中动态生成的类型，该类型是原实体类的子类，并在子类中重写了属性的get方法。 <div style="text-align: right;">  </div>
---	--

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>延迟加载实现原理（续1）</h3> <ul style="list-style-type: none"> • 生成子类的伪代码如下 <pre> public class Emp\$\$EnhancerByCGLIB\$\$abfb9a7f extends Emp { public String getName() { if(未加载) { //加载并注入数据 } return name; } 其他get方法... } </pre> <div style="position: absolute; left: 455px; top: 175px; writing-mode: vertical-rl; background-color: black; color: white; padding: 2px;">代码续篇</div> <div style="position: absolute; bottom: 10px; left: 455px;">++</div>
---	---


4. 关联映射

4.1. 关联映射介绍

4.1.1. 【关联映射介绍】什么是关联映射

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>什么是关联映射</h3> <ul style="list-style-type: none"> • 若两张表具有关联关系，我们可以在实体对象和映射关系文件中配置这种关系，然后使用Hibernate操作其中一张表时，它可以通过配置关系自动的帮我们操作到另一张表，这种通过配置自动操作另一张表的手段称之为关联映射。 <div style="position: absolute; left: 455px; top: 485px; writing-mode: vertical-rl; background-color: black; color: white; padding: 2px;">代码续篇</div> <div style="position: absolute; bottom: 10px; left: 455px;">++</div>
---	--

4.1.2. 【关联映射介绍】关联映射的作用

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>关联映射的作用</h3> <ul style="list-style-type: none"> • 当我们操作一张表的时候，Hibernate可以通过关联映射自动帮助我们操作其关系表。 • 这种关联操作包括 <ul style="list-style-type: none"> - 关联查询出关系表的数据 - 关联新增、修改关系表的数据 - 关联删除关系表的数据 <div style="position: absolute; left: 455px; top: 720px; writing-mode: vertical-rl; background-color: black; color: white; padding: 2px;">代码续篇</div> <div style="position: absolute; bottom: 10px; left: 455px;">++</div>
---	---

4.1.3. 【关联映射介绍】关联映射的类型

Tarena
达内科技

关联映射的类型

- 一对多关联
- 多对一关联
- 多对多关联
- 一对一关联
- 继承关联

+

4.2. 如何使用关联映射

4.2.1. 【如何使用关联映射】明确 2 张表的关系及关系字段

Tarena
达内科技

明确2张表的关系及关系字段

- 首先需要明确2张关系表的关系，以及他们的关系字段。

+

4.2.2. 【如何使用关联映射】在实体类中添加关联属性

Tarena
达内科技

在实体类中添加关联属性

- 在实体类上体现2张表的关系，一般情况下是在当前的实体类中增加一个属性，用于封装其关系表的数据。

+

4.2.3. 【如何使用关联映射】在映射关系文件中配置关联关系

知识讲解

在映射关系文件中配置关联关系

Tarena 达内科技

- 在映射关系文件中配置出2个对象的关系，以及他们的关系类型。

5. 一对多关联

5.1. 一对多关联介绍

5.1.1. 【一对多关联介绍】什么是一对多关联

知识讲解

什么是一对多关联

Tarena 达内科技

- 如果2张表具有一对多的关系，希望在使用Hibernate操作“一”方数据时，可以自动关联操作“多”方数据，那么这种关联映射称之为**一对多关联**。

5.1.2. 【一对多关联介绍】一对多关联的作用

知识讲解

一对多关联的作用

Tarena 达内科技

- 可以通过“一”来操作“多”，包括
 - 通过查询“一”，自动查询“多”。
 - 通过新增/修改“一”，自动新增/修改“多”。
 - 通过删除“一”，自动删除“多”。

5.2. 实现步骤

5.2.1. 【实现步骤】明确关系字段

Tarena
达内科技

明确关系字段

- 以account与服务为例
- account与服务具有一对多关系，其关系字段是 service.account_id

+

5.2.2. 【实现步骤】在“一”方实体类中添加集合属性

Tarena
达内科技

在“一”方实体类中添加集合属性

- 在实体类Account中添加集合属性，以及 get、set方法

```
private Set<Service> services;
public Set<Service> getServices() {
    return this.services;
}
public void setServices(Set<Service> services) {
    this.services = services;
}
```

+

5.2.3. 【实现步骤】在“一”方 hbm 文件中配置关联关系

Tarena
达内科技

在“一”方hbm文件中配置关联关系

- 在Account.hbm.xml中配置与服务关联关系

```
<!-- set指定了属性类型为Set集合，
      name指定了属性名。 -->
<set name= "services" >
  <!-- column指定了关联字段名 -->
  <key column= "account_id" />
  <!-- one-to-many指定了关联关系，
      class指定了另一方类型。 -->
  <one-to-many class= "com.tarena.entity.Service" />
</set>
```

+

经典案例

1. 验证一级缓存

- 问题

设计几个测试案例，以验证一级缓存的存在及特性。

- 方案

- 1) 用同一个 Session 查询同一条数据 2 次，如果只查询一次数据库，则验证了一级缓存的存在。
- 2) 用 2 个不同的 Session，分别查询同一条数据，如果查询 2 次数据库，则验证了一级缓存是 Session 独享的。

- 步骤

实现此案例需要按照如下步骤进行。

步骤一：创建项目

复制项目 HibernateDay01，粘贴并将项目名改为 HibernateDay02。

步骤二：写测试案例代码

在 com.tarena.test 包下创建测试类 TestFirstCache，在这个测试类中分别写出方案中提到的 2 个测试方法，用以验证一级缓存的存在及特性，代码如下：

```
package com.tarena.test;

import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Emp;
import com.tarena.util.HibernateUtil;

public class TestFirstCache {

    /**
     * 用同一个 Session 查询同一条数据 2 次，
     * 如果只查询一次数据库，则验证了一级缓存的存在。
     */
    @Test
    public void test1() {
        Session session = HibernateUtil.getSession();
        Emp e1 = (Emp) session.get(Emp.class, 321);
        System.out.println(e1.getName());
        System.out.println("-----");
        Emp e2 = (Emp) session.get(Emp.class, 321);
        System.out.println(e2.getName());
        session.close();
    }

    /**
```

- * 用 2 个不同的 Session，分别查询同一条数据，
- * 如果查询 2 次数据库，则验证了一级缓存是 Session 独享的。

```
*/
@Test
public void test2() {
    Session session1 = HibernateUtil.getSession();
    Emp e1 = (Emp) session1.get(Emp.class, 321);
    System.out.println(e1.getName());

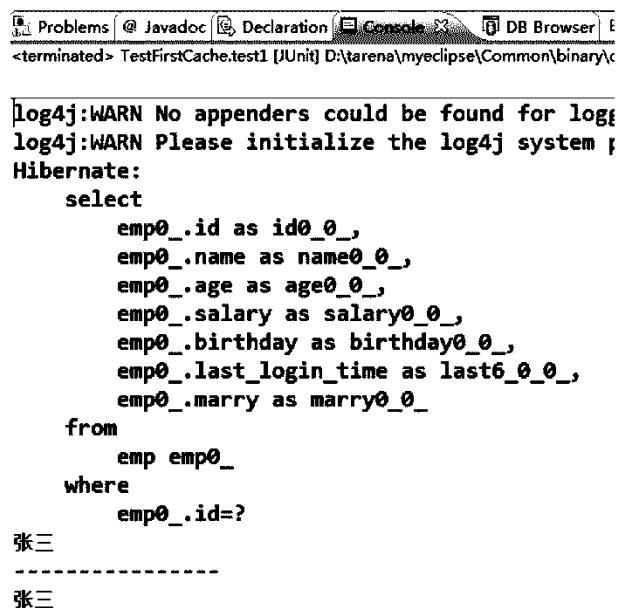
    Session session2 = HibernateUtil.getSession();
    Emp e2 = (Emp) session2.get(Emp.class, 321);
    System.out.println(e2.getName());

    session1.close();
    session2.close();
}
}
```

步骤三：测试

分别执行以上 2 个方法，根据控制台输出的结果，根据其 SQL 语句数量可以判断出查询执行的次数，进而验证一级缓存的存在及特性。

test1() 执行后，控制台输出结果如下图，可以看出第二次查询并没有真正访问数据库，验证了一级缓存的存在：



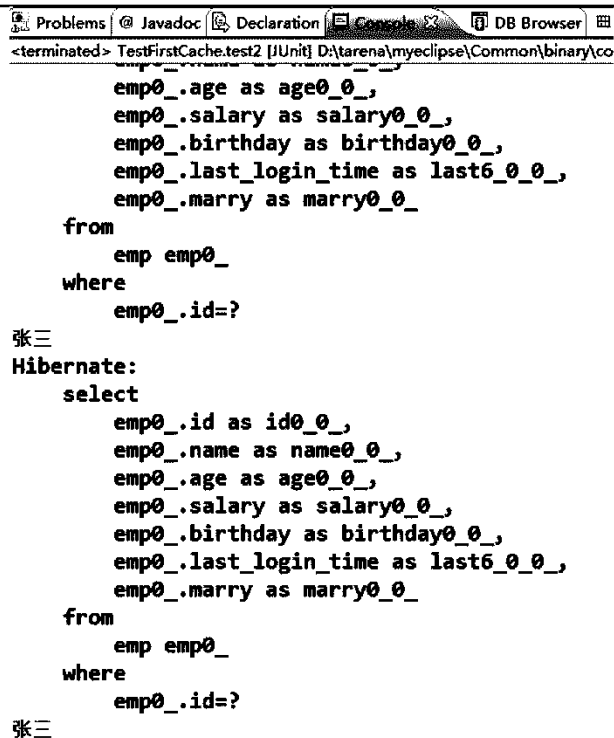
```

<terminated> TestFirstCache.test1 [JUnit] D:\tarena\myeclipse\Common\binary\c
log4j:WARN No appenders could be found for logger log4j
log4j:WARN Please initialize the log4j system ;
Hibernate:
select
    emp0_.id as id0_0_,
    emp0_.name as name0_0_,
    emp0_.age as age0_0_,
    emp0_.salary as salary0_0_,
    emp0_.birthday as birthday0_0_,
    emp0_.last_login_time as last6_0_0_,
    emp0_.marry as marry0_0_
from
    emp emp0_
where
    emp0_.id=?
张三
-----
张三

```

图-1

test2() 执行后，控制台输出结果如下图，可以看出第二次查询访问了数据库，验证了一级缓存是 Session 独享的。



```

<terminated> TestFirstCache.test2 [JUnit] D:\tarena\myeclipse\Common\binary\co
    emp0_.age as age0_0_,
    emp0_.salary as salary0_0_,
    emp0_.birthday as birthday0_0_,
    emp0_.last_login_time as last6_0_0_,
    emp0_.marry as marry0_0_
from
    emp emp0_
where
    emp0_.id=?
张三
Hibernate:
select
    emp0_.id as id0_0_,
    emp0_.name as name0_0_,
    emp0_.age as age0_0_,
    emp0_.salary as salary0_0_,
    emp0_.birthday as birthday0_0_,
    emp0_.last_login_time as last6_0_0_,
    emp0_.marry as marry0_0_
from
    emp emp0_
where
    emp0_.id=?
张三

```

图-2

• 完整代码

下面是本案例的完整代码。

其中 TestFirstCache 完整代码如下：

```

package com.tarena.test;

import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Emp;
import com.tarena.util.HibernateUtil;

public class TestFirstCache {

    /**
     * 用同一个 Session 查询同一条数据 2 次，
     * 如果只查询一次数据库，则验证了一级缓存的存在。
     */
    @Test
    public void test1() {
        Session session = HibernateUtil.getSession();
        Emp e1 = (Emp) session.get(Emp.class, 321);
        System.out.println(e1.getName());
        System.out.println("-----");
        Emp e2 = (Emp) session.get(Emp.class, 321);
        System.out.println(e2.getName());
        session.close();
    }

    /**
     * 用 2 个不同的 Session，分别查询同一条数据，
     * 如果查询 2 次数据库，则验证了一级缓存是 Session 独享的。
     */
}

```

```
@Test
public void test2() {
    Session session1 = HibernateUtil.getSession();
    Emp e1 = (Emp) session1.get(Emp.class, 321);
    System.out.println(e1.getName());

    Session session2 = HibernateUtil.getSession();
    Emp e2 = (Emp) session2.get(Emp.class, 321);
    System.out.println(e2.getName());

    session1.close();
    session2.close();
}
}
```

2. 管理一级缓存

- **问题**

掌握一级缓存管理的 2 种方式：

- 1) 使用 evict 方法，从一级缓存中移除一个对象。
- 2) 使用 clear 方法，将一级缓存中的对象全部移除。

设计出案例，来使用并验证一级缓存管理方法。

- **方案**

设计 2 个案例，使用同一个 Session 查询同一条数据 2 次，由于一级缓存的存在，第二次查询时将从一级缓存中取数，而不会查询数据库。

那么，如果在第二次查询之前将数据从缓存中移除，第二次查询时就会访问数据库。在这两个案例中，我们分别使用 evict 和 clear 方法将数据从缓存中移除。

- **步骤**

实现此案例需要按照如下步骤进行。

步骤一：在 TestFirstCache 中增加测试案例代码

在 TestFirstCache 中增加 2 个方法，均使用同一个 Session 查询同一条数据 2 次，在第二次查询之前，分别使用 evice 和 clear 方法移除缓存数据，代码如下：

```
package com.tarena.test;

import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Emp;
import com.tarena.util.HibernateUtil;

public class TestFirstCache {
    /**
     * 用同一个 Session 查询同一条数据 2 次，
     * 如果只查询一次数据库，则验证了一级缓存的存在。
     */
}
```



```
@Test
public void test1() {
    Session session = HibernateUtil.getSession();
    Emp e1 = (Emp) session.get(Emp.class, 321);
    System.out.println(e1.getName());
    System.out.println("-----");
    Emp e2 = (Emp) session.get(Emp.class, 321);
    System.out.println(e2.getName());
    session.close();
}

/**
 * 用 2 个不同的 Session，分别查询同一条数据，
 * 如果查询 2 次数据库，则验证了一级缓存是 Session 独享的。
 */
@Test
public void test2() {
    Session session1 = HibernateUtil.getSession();
    Emp e1 = (Emp) session1.get(Emp.class, 321);
    System.out.println(e1.getName());

    Session session2 = HibernateUtil.getSession();
    Emp e2 = (Emp) session2.get(Emp.class, 321);
    System.out.println(e2.getName());

    session1.close();
    session2.close();
}

/**
 * 验证缓存管理的方法 evict
 */
@Test
public void test3() {
    Session session = HibernateUtil.getSession();
    Emp e1 = (Emp) session.get(Emp.class, 321);
    System.out.println(e1.getName());
    session.evict(e1);
    Emp e2 = (Emp) session.get(Emp.class, 321);
    System.out.println(e2.getName());
    session.close();
}

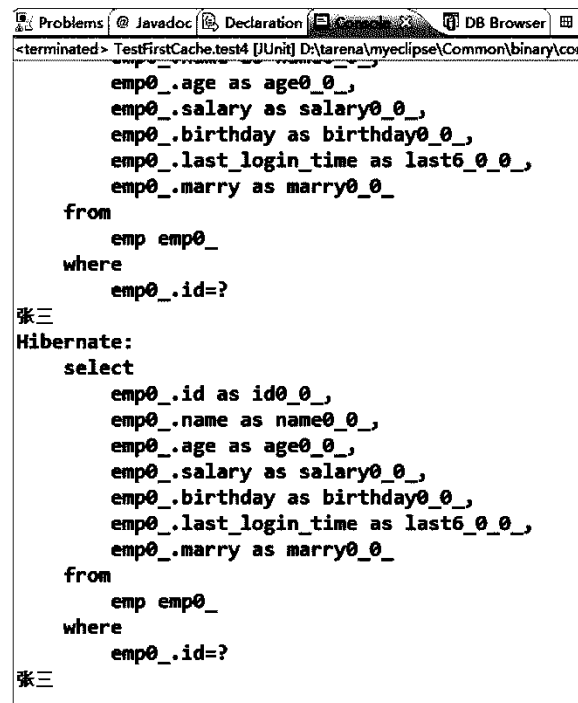
/**
 * 验证缓存管理的方法 clear
 */
@Test
public void test4() {
    Session session = HibernateUtil.getSession();
    Emp e1 = (Emp) session.get(Emp.class, 321);
    System.out.println(e1.getName());
    session.clear();
    Emp e2 = (Emp) session.get(Emp.class, 321);
    System.out.println(e2.getName());
    session.close();
}
}
```

步骤二：测试

分别执行这两个方法，并观察控制台，会发现都只输出了一次 SQL，验证了这 2 个方法

可以有效的管理一级缓存。

test3()执行后，控制台输出结果如下图，可以看出第二次查询访问了数据库，验证了evict 管理一级缓存是有效的。



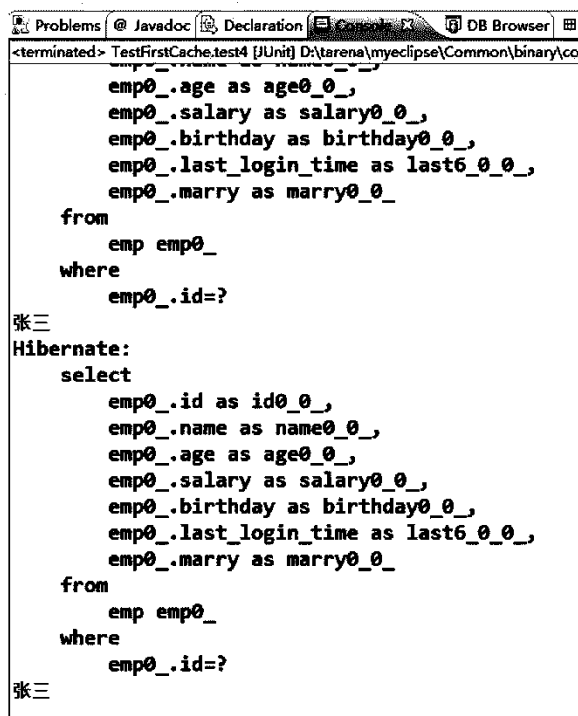
```

<terminated> TestFirstCache.test3 [JUnit] D:\tarena\myeclipse\Common\binary\cor
emp0_.age as age0_0_,
emp0_.salary as salary0_0_,
emp0_.birthday as birthday0_0_,
emp0_.last_login_time as last6_0_0_,
emp0_.marry as marry0_0_
from
emp emp0_
where
emp0_.id=?
张三
Hibernate:
select
emp0_.id as id0_0_,
emp0_.name as name0_0_,
emp0_.age as age0_0_,
emp0_.salary as salary0_0_,
emp0_.birthday as birthday0_0_,
emp0_.last_login_time as last6_0_0_,
emp0_.marry as marry0_0_
from
emp emp0_
where
emp0_.id=?
张三

```

图-3

test4()执行后，控制台输出结果如下图，可以看出第二次查询访问了数据库，验证了clear 管理一级缓存是有效的。



```

<terminated> TestFirstCache.test4 [JUnit] D:\tarena\myeclipse\Common\binary\cor
emp0_.age as age0_0_,
emp0_.salary as salary0_0_,
emp0_.birthday as birthday0_0_,
emp0_.last_login_time as last6_0_0_,
emp0_.marry as marry0_0_
from
emp emp0_
where
emp0_.id=?
张三
Hibernate:
select
emp0_.id as id0_0_,
emp0_.name as name0_0_,
emp0_.age as age0_0_,
emp0_.salary as salary0_0_,
emp0_.birthday as birthday0_0_,
emp0_.last_login_time as last6_0_0_,
emp0_.marry as marry0_0_
from
emp emp0_
where
emp0_.id=?
张三

```

图-4

- 完整代码

下面是本案例的完整代码。

其中 TestFirstCache 完整代码如下：

```
package com.tarena.test;

import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Emp;
import com.tarena.util.HibernateUtil;

public class TestFirstCache {

    /**
     * 用同一个 Session 查询同一条数据 2 次，
     * 如果只查询一次数据库，则验证了一级缓存的存在。
     */
    @Test
    public void test1() {
        Session session = HibernateUtil.getSession();
        Emp e1 = (Emp) session.get(Emp.class, 321);
        System.out.println(e1.getName());
        System.out.println("-----");
        Emp e2 = (Emp) session.get(Emp.class, 321);
        System.out.println(e2.getName());
        session.close();
    }

    /**
     * 用 2 个不同的 Session，分别查询同一条数据，
     * 如果查询 2 次数据库，则验证了一级缓存是 Session 独享的。
     */
    @Test
    public void test2() {
        Session session1 = HibernateUtil.getSession();
        Emp e1 = (Emp) session1.get(Emp.class, 321);
        System.out.println(e1.getName());

        Session session2 = HibernateUtil.getSession();
        Emp e2 = (Emp) session2.get(Emp.class, 321);
        System.out.println(e2.getName());

        session1.close();
        session2.close();
    }

    /**
     * 验证缓存管理的方法 evict
     */
    @Test
    public void test3() {
        Session session = HibernateUtil.getSession();
        Emp e1 = (Emp) session.get(Emp.class, 321);
        System.out.println(e1.getName());
        session.evict(e1);
        Emp e2 = (Emp) session.get(Emp.class, 321);
        System.out.println(e2.getName());
        session.close();
    }

    /**
     * 验证缓存管理的方法 clear
     */
}
```

```
@Test
public void test4() {
    Session session = HibernateUtil.getSession();
    Emp e1 = (Emp) session.get(Emp.class, 321);
    System.out.println(e1.getName());
    session.clear();
    Emp e2 = (Emp) session.get(Emp.class, 321);
    System.out.println(e2.getName());
    session.close();
}
}
```

3. 验证持久态对象的特性

• 问题

设计出案例，验证持久态对象的特性：

- 1) 持久态对象存在于一级缓存中。
- 2) 持久态对象可以自动更新至数据库。
- 3) 持久态对象自动更新数据库的时机是 session.flush()。

• 方案

设计 3 个案例，分别验证持久态对象的 3 个特性：

- 1) 新增一条数据，在新增后该数据对象为持久态的，然后根据对象的 ID 再次查询数据。执行时如果控制台不重新输出 SQL 则验证了持久态对象存在于一级缓存。
- 2) 新增一条数据，在新增后该对象为持久态的，然后修改这个对象的任意属性值，并提交事务。在执行时，如果发现数据库中的数据是修改后的内容，则验证了持久态对象可以自动更新至数据库。
- 3) 查询一条数据，该数据对象为持久态的，然后修改对象的任意属性值，再调用 session.flush()方法，并且不提交事务。如果执行时控制台输出更新的 SQL，则验证了一级缓存对象更新至数据库的时机为 session.flush()。

• 步骤

实现此案例需要按照如下步骤进行。

步骤一：创建验证对象持久性测试类

在 com.tarena.test 包下，创建一个测试类 TestPersistent，在其中写出验证对象持久性的 3 个测试方法，用以验证对象持久性的 3 个特性，代码如下：

```
package com.tarena.test;

import java.sql.Date;
import java.sql.Timestamp;
```

```
import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.junit.Test;
import com.tarena.entity.Emp;
import com.tarena.util.HibernateUtil;

public class TestPersistent {

    /**
     * 持久态对象存在于一级缓存中
     */
    @Test
    public void test1() {
        Emp e = new Emp();
        e.setName("唐僧");
        e.setAge(29);
        e.setMarry(false);
        e.setSalary(12000.00);
        e.setBirthday(
            Date.valueOf("1983-10-20"));
        e.setLastLoginTime(
            new Timestamp(System.currentTimeMillis()));

        Session session = HibernateUtil.getSession();
        Transaction ts = session.beginTransaction();
        try {
            session.save(e);
            ts.commit();
        } catch (HibernateException e1) {
            e1.printStackTrace();
            ts.rollback();
        }

        Emp emp = (Emp) session.get(Emp.class, e.getId());
        System.out.println(emp.getId() + " " + emp.getName());

        session.close();
    }

    /**
     * 持久态对象可以自动更新至数据库
     */
    @Test
    public void test2() {
        Emp e = new Emp();
        e.setName("孙悟空");
        e.setAge(29);
        e.setMarry(false);
        e.setSalary(12000.00);
        e.setBirthday(
            Date.valueOf("1983-10-20"));
        e.setLastLoginTime(
            new Timestamp(System.currentTimeMillis()));

        Session session = HibernateUtil.getSession();
        Transaction ts = session.beginTransaction();
        try {
            session.save(e);
            e.setName("猪八戒");
            ts.commit();
        } catch (HibernateException e1) {
            e1.printStackTrace();
            ts.rollback();
        }
    }
}
```

```

        session.close();
    }
    /**
     * 持久态对象自动更新数据库的时机
     */
    @Test
    public void test3() {
        Session session = HibernateUtil.getSession();
        Emp e = (Emp) session.load(Emp.class, 201);
        e.setName("太上老君");
        session.flush(); //同步但未提交事务
        session.close();
    }
}

```

步骤二：测试

分别执行上述 3 个测试方法，观察控制台输出的 SQL，看是否与方案中描述的一致。

test1() 执行后，控制台输出结果如下图，可以看出后面的查询并没有再次访问数据库，验证了持久态对象是存在于一级缓存中的。

```

<terminated> TestPersistent.test1 [JUnit] D:\tarena\myeclipse\Common\binary\com.sun.java.jdk.win32.x86_1.6.0.01
log4j:WARN No appenders could be found for logger (org.hibernate.c
log4j:WARN Please initialize the log4j system properly.
Hibernate:
    select
        emp_seq.nextval
    from
        dual
Hibernate:
    insert
    into
        emp
        (name, age, salary, birthday, last_login_time, marry, id)
    values
        (?, ?, ?, ?, ?, ?, ?)
323 唐僧

```

图-5

test2() 执行后，控制台输出结果如下图，可以看出 Hibernate 自动执行了一次更新，验证了持久态对象会自动更新至数据库。

```

Problems @ Javadoc Declaration DB Browser SQL Results
<terminated> TestPersistent.test2 [JUnit] D:\tarena\myeclipse\Common\binary\com.sun.java.jdk.win32.x86_1.6.0.013
Hibernate:
select
  emp_seq.nextval
from
  dual
Hibernate:
insert
into
  emp
  (name, age, salary, birthday, last_login_time, marry, id)
values
  (?, ?, ?, ?, ?, ?, ?)
Hibernate:
update
  emp
set
  name=?,
  age=?,
  salary=?,
  birthday=?,
  last_login_time=?,
  marry=?
where
  id=?

```

图-6

EMP 表查询结果如下图，可以看出最后一条数据的名称的确被更新为猪八戒。

ID	NAME	AGE	SALARY	MARRY	BIRTHDAY	LAST_LOGIN_TIME
321	张三	30	10000	Y	<NULL>	<NULL>
322	李四	31	12000	Y	<NULL>	<NULL>
323	唐僧	29	12000	N	1983-10-20 00:00:00.0	2014-07-11 09:47:14.0
324	猪八戒	29	12000	N	1983-10-20 00:00:00.0	2014-07-11 09:48:03.0

图-7

test3() 执行后，控制台输出结果如下图，可以看出 Hibernate 自动执行了一次更新，验证了自动更新的时机是 session.flush()。

```

Problems @ Javadoc Declaration DB Browser
<terminated> TestPersistent.test3 [JUnit] D:\tarena\myeclipse\Common\binary\
select
  emp0_.id as id0_0_,
  emp0_.name as name0_0_,
  emp0_.age as age0_0_,
  emp0_.salary as salary0_0_,
  emp0_.birthday as birthday0_0_,
  emp0_.last_login_time as last6_0_0_,
  emp0_.marry as marry0_0_
from
  emp emp0_
where
  emp0_.id=?
Hibernate:
update
  emp
set
  name=?,
  age=?,
  salary=?,
  birthday=?,
  last_login_time=?,
  marry=?
where
  id=?

```

图-8

EMP 表查询结果如下图，可以看出最后一条数据的名称并没有更新为太上老君，说明 session.flush 只是触发更新，并没有提交事务。

ID	NAME	AGE	SALARY	MARRY	BIRTHDAY	LAST_LOGIN_TIME
321	张三	30	10000	Y	<NULL>	<NULL>
322	李四	31	12000	Y	<NULL>	<NULL>
323	唐僧	29	12000	N	1983-10-20 00:00:00.0	2014-07-11 09:47:14.0
324	猪八戒	29	12000	N	1983-10-20 00:00:00.0	2014-07-11 09:48:03.0

图-9

• 完整代码

本案例的完整代码如下所示：

TestPersistent 完整代码如下：

```
package com.tarena.test;

import java.sql.Date;
import java.sql.Timestamp;
import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.junit.Test;
import com.tarena.entity.Emp;
import com.tarena.util.HibernateUtil;

public class TestPersistent {
    /**
     * 持久态对象存在于一级缓存中
     */
    @Test
    public void test1() {
        Emp e = new Emp();
        e.setName("唐僧");
        e.setAge(29);
        e.setMarry(false);
        e.setSalary(12000.00);
        e.setBirthday(Date.valueOf("1983-10-20"));
        e.setLastLoginTime(
            new Timestamp(System.currentTimeMillis()));

        Session session = HibernateUtil.getSession();
        Transaction ts = session.beginTransaction();
        try {
            session.save(e);
            ts.commit();
        } catch (HibernateException e1) {
            e1.printStackTrace();
            ts.rollback();
        }

        Emp emp = (Emp) session.get(Emp.class, e.getId());
        System.out.println(emp.getId() + " " + emp.getName());

        session.close();
    }

    /**
     * 持久态对象可以自动更新至数据库
     */
}
```



```
@Test
public void test2() {
    Emp e = new Emp();
    e.setName("孙悟空");
    e.setAge(29);
    e.setMarry(false);
    e.setSalary(12000.00);
    e.setBirthday(Date.valueOf("1983-10-20"));
    e.setLastLoginTime(
        new Timestamp(System.currentTimeMillis()));

    Session session = HibernateUtil.getSession();
    Transaction ts = session.beginTransaction();
    try {
        session.save(e);
        e.setName("猪八戒");
        ts.commit();
    } catch (HibernateException e1) {
        e1.printStackTrace();
        ts.rollback();
    }
    session.close();
}
/**
 * 持久态对象自动更新数据库的时机
 */
@Test
public void test3() {
    Session session = HibernateUtil.getSession();
    Emp e = (Emp) session.load(Emp.class, 201);
    e.setName("太上老君");
    session.flush(); //同步但未提交事务
    session.close();
}
}
```

4. 验证延迟加载

- 问题

设计案例，验证 session.load()方法和 query.iterate()方法在查询时是采用延迟加载机制的。

- 方案

- 1) 验证 session.load()的延迟加载，可以先查询某条 EMP 数据，然后输出一个分割线，在分割线的后面再使用这个对象，输出它的一些属性值。在运行时，如果查询 EMP 的 SQL 输出在分割线的后面，则验证了该方法是采用延迟加载机制的。
- 2) 验证 query.iterate()的延迟加载，可以先查询出全部的 EMP 数据，然后输出一个分割线，在分割线的后面再遍历查询结果并输出每个对象的一些属性值。在运行时，如果查询 EMP 的 SQL 输出在分割线的后面，则验证了该方法是采用延迟加载机制的。

• 步骤

实现此案例需要按照如下步骤进行。

步骤一：创建验证延迟加载的测试类

在 `com.tarena.test` 包下，创建一个测试类 `TestLazy`，并在这个类中写 2 个测试方法，分别验证 `session.load()` 和 `query.iterate()` 方法是采用延迟加载机制的，代码如下：

```
package com.tarena.test;

import java.util.Iterator;
import org.hibernate.Query;
import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Emp;
import com.tarena.util.HibernateUtil;

public class TestLazy {

    /**
     * 验证 load 方法是延迟加载的
     */
    @Test
    public void test1() {
        Session session = HibernateUtil.getSession();
        // load 方法并没有触发访问数据库
        Emp emp = (Emp) session.load(Emp.class, 321);
        System.out.println("-----");
        // 使用 emp 对象时才真正访问数据库
        System.out.println(emp.getName());
        session.close();
    }

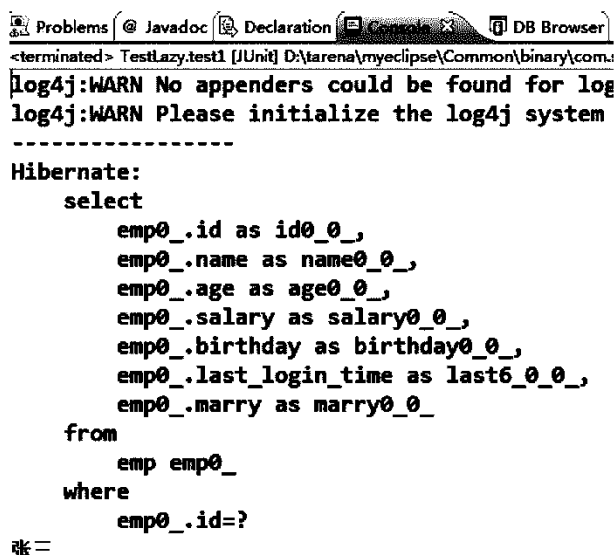
    /**
     * 验证 iterate 方法是延迟加载的
     */
    @Test
    public void test2() {
        String hql = "from Emp";
        Session session = HibernateUtil.getSession();
        Query query = session.createQuery(hql);
        // iterate 方法访问了数据库，但只查询了 ID 列
        Iterator<Emp> it = query.iterate();
        System.out.println("-----");
        while (it.hasNext()) {
            Emp emp = it.next();
            // 使用 emp 对象时才将其他列全部加载
            System.out.println(emp.getName());
        }
        session.close();
    }
}
```

步骤二：测试

分别执行上面的 2 个测试方法，根据控制台输出的结果，判断 `session.load()` 和

query.iterate()方法是否采用延迟加载机制。

test1()方法执行后,控制台输出结果如下图,可以看出查询 SQL 是输出在分割线后面的,也就是在使用 emp 对象时才触发了数据库的访问,验证了延迟加载的存在。



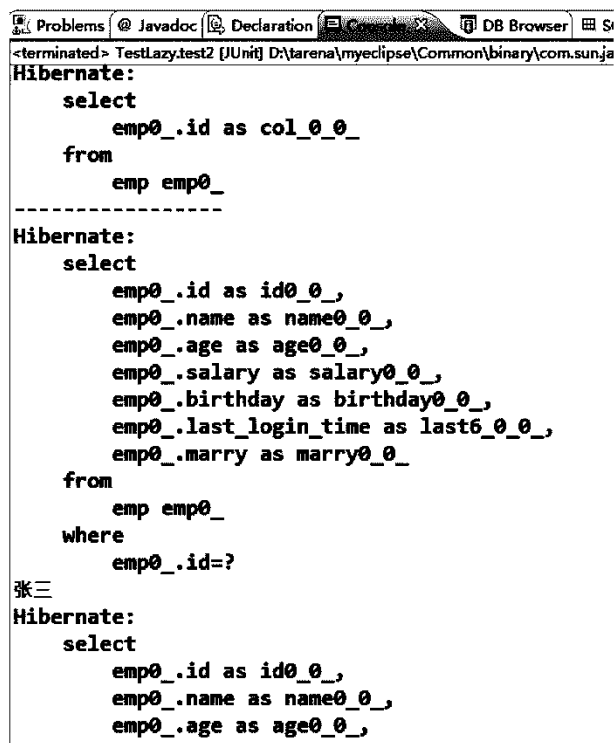
```

Problems | @ Javadoc | Declaration | Console | DB Browser
<terminated> TestLazy.test1 [JUnit] D:\tarena\myeclipse\Common\binary\com.
log4j:WARN No appenders could be found for log
log4j:WARN Please initialize the log4j system
-----
Hibernate:
select
    emp0_.id as id0_0_,
    emp0_.name as name0_0_,
    emp0_.age as age0_0_,
    emp0_.salary as salary0_0_,
    emp0_.birthday as birthday0_0_,
    emp0_.last_login_time as last6_0_0_,
    emp0_.marry as marry0_0_
from
    emp emp0_
where
    emp0_.id=?
张三

```

图-10

test2()方法执行后,控制台输出结果如下图,可以看出查询 EMP 表全部内容的 SQL 输出在分割线的后面,也就是在使用 emp 对象时才触发的数据库访问,验证了延迟加载的存在。



```

Problems | @ Javadoc | Declaration | Console | DB Browser | Si
<terminated> TestLazy.test2 [JUnit] D:\tarena\myeclipse\Common\binary\com.sun.ja
Hibernate:
select
    emp0_.id as col_0_0_
from
    emp emp0_
-----
Hibernate:
select
    emp0_.id as id0_0_,
    emp0_.name as name0_0_,
    emp0_.age as age0_0_,
    emp0_.salary as salary0_0_,
    emp0_.birthday as birthday0_0_,
    emp0_.last_login_time as last6_0_0_,
    emp0_.marry as marry0_0_
from
    emp emp0_
where
    emp0_.id=?
张三
Hibernate:
select
    emp0_.id as id0_0_,
    emp0_.name as name0_0_,
    emp0_.age as age0_0_,

```

图-11

- **完整代码**

本案例的完整代码如下所示：

```
package com.tarena.test;

import java.util.Iterator;
import org.hibernate.Query;
import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Emp;
import com.tarena.util.HibernateUtil;

public class TestLazy {

    /**
     * 验证 load 方法是延迟加载的
     */
    @Test
    public void test1() {
        Session session = HibernateUtil.getSession();
        // load 方法并没有触发访问数据库
        Emp emp = (Emp) session.load(Emp.class, 321);
        System.out.println("-----");
        // 使用 emp 对象时才真正访问数据库
        System.out.println(emp.getName());
        session.close();
    }

    /**
     * 验证 iterate 方法是延迟加载的
     */
    @Test
    public void test2() {
        String hql = "from Emp";
        Session session = HibernateUtil.getSession();
        Query query = session.createQuery(hql);
        // iterate 方法访问了数据库，但只查询了 ID 列
        Iterator<Emp> it = query.iterate();
        System.out.println("-----");
        while (it.hasNext()) {
            Emp emp = it.next();
            // 使用 emp 对象时才将其他列全部加载
            System.out.println(emp.getName());
        }
        session.close();
    }
}
```

5. 在 NETCTOSS 中使用延迟加载

- **问题**

请将 NETCTOSS 中资费模块 DAO 的实现，改用 Hibernate 中延迟加载的方法。

- **方案**

将 CostDaoImpl 中的 findById 方法的实现，改为 session.load()。

为了避免 session 提前关闭导致延迟加载出现问题，需要在 findById 方法中不关闭 session，而是自定义拦截器，在拦截器中调用完 action 之后关闭 session。自然地，资费模块的 action 都应该引用这个拦截器。

- **步骤**

实现此案例需要按照如下步骤进行。

步骤一：使用延迟加载方法

将 CostDaoImpl 中的 findById 方法中，session.get()改为 session.load()，代码如下：

```
package com.netctoss.dao;

import java.util.List;
import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;
import com.netctoss.entity.Cost;
import com.netctoss.util.HibernateUtil;

/**
 * 当前阶段学习重点是 Struts2，对于 DAO 的实现就模拟实现了。
 * 同学们可以使用 JDBC/MyBatis 自行实现该 DAO。
 */
public class CostDaoImpl implements ICostDao {

    @Override
    public List<Cost> findAll() {
        String hql = "from Cost";
        Session session = HibernateUtil.getSession();
        Query query = session.createQuery(hql);
        List<Cost> list = query.list();
        session.close();
        return list;
    }

    @Override
    public void delete(int id) {
        Cost cost = new Cost();
        cost.setId(id);

        Session session = HibernateUtil.getSession();
        Transaction ts = session.beginTransaction();
        try {
            session.delete(cost);
            ts.commit();
        } catch (HibernateException e) {
            e.printStackTrace();
            ts.rollback();
        } finally {
            session.close();
        }
    }

    @Override
    public Cost findByName(String name) {
        // 模拟根据名称查询资费数据，假设资费表中只有一条名为 tarena 的数据
    }
}
```

```

        if("tarena".equals(name)) {
            Cost c = new Cost();
            c.setId(97);
            c.setName("tarena");
            c.setBaseDuration(99);
            c.setBaseCost(9.9);
            c.setUnitCost(0.9);

            c.setDescr("tarena 套餐");
            c.setStatus("0");
            c.setCostType("2");
            return c;
        }
        return null;
    }

    @Override
    public Cost findById(int id) {
        Session session = HibernateUtil.getSession();

        Cost cost = (Cost) session.load(Cost.class, id);

        session.close();
        return cost;
    }
}

```

步骤二：测试

重新部署项目，并重启 tomcat，访问资费修改页面，效果如下图：

图-12

可以看出要修改的数据，只有 ID 显示正确，其他字段都为空。原因是我们的 findById 方法中直接关闭了 session，而该方法返回的对象是在 JSP 中使用的，在使用时 session 已经关闭，由于延迟加载机制的存在，导致了这个问题的发生。要想解决这个问题，我们需要继续如下的步骤。

步骤三：重构 HibernateUtil，使用 ThreadLocal 管理 Session

重构 HibernateUtil，引入 ThreadLocal 来管理 Session。ThreadLocal 对象是与线程有关的工具类，它的目的是将管理的对象按照线程进行隔离，以保证一个线程只对应一个对象。

由于后面我们要在拦截器中关闭连接，因此需要准确的取出 DAO 中使用的连接对象，为了便于实现在一个线程（一次客户端请求，就是一个线程）中，不同的代码位置获取同一个连接，那么使用 ThreadLocal 来管理 session 就再合适不过了。

请注意，引入 ThreadLocal 管理 session，不仅仅是在创建 session 时将其加入到 ThreadLocal 中，在关闭 session 时也需要将其从 ThreadLocal 中移除，因此 HibernateUtil 中还需要提供一个关闭 session 的方法。

重构以后，HibernateUtil 代码如下：

```
package com.netctoss.util;

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

public class HibernateUtil {

    private static SessionFactory sessionFactory;

    /**
     * 使用 ThreadLocal 管理 Session，可以保证一个线程中只有唯一的一个连接。
     * 并且我们在获取连接时，它会自动的给我们返回当前线程对应的连接。
     */
    private static ThreadLocal<Session> tl =
        new ThreadLocal<Session>();

    static {
        // 加载 Hibernate 主配置文件
        Configuration conf = new Configuration();
        conf.configure("/hibernate.cfg.xml");
        sessionFactory = conf.buildSessionFactory();
    }

    /**
     * 创建 session
     */
    public static Session getSession() {

        // ThreadLocal 会以当前线程名为 key 获取连接
        Session session = tl.get();
        // 如果取到的当前线程的连接为空
        if(session == null) {
            // 使用工厂创建连接
            session = sessionFactory.openSession();
            // ThreadLocal 会以当前线程名为 key 保存 session
            tl.set(session);
        }
        return session;
    }
}
```

```
/**
 * 关闭 session
 */
public static void close() {
    // ThreadLocal 会以当前线程名为 key 获取连接
    Session session = tl.get();
    // 如果取到的当前线程的连接不为空
    if(session != null) {
        // 关闭 session
        session.close();
        // 将当前线程对应的连接从 ThreadLocal 中移除
        tl.remove();
    }
}

public static void main(String[] args) {
    System.out.println(getSession());
    close();
}
}
```

步骤四：创建保持 session 在视图层开启的拦截器

在 `com.netctoss.interceptor` 包下，创建一个拦截器 `OpenSessionInViewInterceptor`，在拦截方法中，先调用 `action` 和 `result`，之后再关闭本次访问线程对应的 `session`，代码如下：

```
package com.netctoss.interceptor;

import com.netctoss.util.HibernateUtil;
import com.opensymphony.xwork2.ActionInvocation;
import com.opensymphony.xwork2.interceptor.Interceptor;

/**
 * 保持 Session 在视图层开启的拦截器，主要作用
 * 是在执行完 JSP 之后再统一关闭 session。
 */
public class OpenSessionInViewInterceptor
    implements Interceptor {

    @Override
    public void destroy() {
    }

    @Override
    public void init() {
    }

    @Override
    public String intercept(ActionInvocation ai)
        throws Exception {
        // 调用 action 和 result
        ai.invoke();
        /*
         * result 会把请求转发到页面，因此调用 result，

```



```

    * 就相当于调用 JSP，因此此处的代码是在 JSP 之后执行。
    * */
    HibernateUtil.close();
    return null;
}

}

```

步骤五：注册并引用拦截器

在 struts.xml 中，注册并引用这个拦截器，代码如下：

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.1.7//EN"
    "http://struts.apache.org/dtds/struts-2.1.7.dtd">
<struts>

    <!-- 公共的包，封装了通用的拦截器、通用的 result -->
    <package name="netctoss" extends="json-default">
        <interceptors>
            <!-- 登录检查拦截器 -->
            <interceptor name="loginInterceptor"
                class="com.netctoss.interceptor.LoginInterceptor"/>

            <!-- 保持 session 开启拦截器 -->
            <interceptor name="openSessionInterceptor"
                class="com.netctoss.interceptor.OpenSessionInViewInterceptor"/>

            <!-- 登录检查拦截器栈 -->
            <interceptor-stack name="loginStack">
                <interceptor-ref name="loginInterceptor"/>

                <interceptor-ref name="openSessionInterceptor"/>

            <!-- 不要丢掉默认的拦截器栈，里面有很多 Struts2 依赖的拦截器 -->
            <interceptor-ref name="defaultStack"/>
        </interceptor-stack>
    </interceptors>

    <!-- 设置 action 默认引用的拦截器 -->
    <default-interceptor-ref name="loginStack"/>

    <!-- 全局的 result，包下所有的 action 都可以共用 -->
    <global-results>
        <!-- 跳转到登录页面的 result -->
        <result name="login" type="redirectAction">
            <param name="namespace">/login</param>
            <param name="actionName">toLogin</param>
        </result>
    </global-results>
</package>

<!--
    资费模块配置信息：
    一般情况下，一个模块的配置单独封装在一个 package 下，
    并且以模块名来命名 package 的 name 和 namespace。
-->
<package name="cost" namespace="/cost" extends="netctoss">

```

```

<!-- 查询资费数据 -->
<action name="findCost" class="com.netctoss.action.FindCostAction">
    <!--
        正常情况下跳转到资费列表页面。
        一般一个模块的页面要打包在一个文件夹下，并且文件夹以模块名命名。
    -->
    <result name="success">
        /WEB-INF/cost/find_cost.jsp
    </result>
    <!--
        错误情况下，跳转到错误页面。
        错误页面可以被所有模块复用，因此放在 main 下，
        该文件夹用于存放公用的页面。
    -->
    <result name="error">
        /WEB-INF/main/error.jsp
    </result>
</action>
<!-- 删除资费 -->
<action name="deleteCost"
    class="com.netctoss.action.DeleteCostAction">
    <!-- 删除完之后，重定向到查询 action -->
    <result name="success" type="redirectAction">
        findCost
    </result>
    <result name="error">
        /WEB-INF/main/error.jsp
    </result>
</action>
<!-- 打开资费新增页 -->
<action name="toAddCost">
    <result name="success">
        /WEB-INF/cost/add_cost.jsp
    </result>
</action>
<!-- 资费名唯一性校验 -->
<action name="checkCostName"
    class="com.netctoss.action.CheckCostNameAction">
    <!-- 使用 json 类型的 result 把结果输出给回调函数 -->
    <result name="success" type="json">
        <param name="root">info</param>
    </result>
</action>
<!-- 打开修改页面 -->
<action name="toUpdateCost"
    class="com.netctoss.action.ToUpdateCostAction">
    <result name="success">
        /WEB-INF/cost/update_cost.jsp
    </result>
    <result name="error">
        /WEB-INF/main/error.jsp
    </result>
</action>
</package>

<!-- 登录模块 -->
<package name="login" namespace="/login" extends="struts-default">
    <!--
        打开登录页面：
        1、action 的 class 属性可以省略，省略时 Struts2
    -->

```

会自动实例化默认的 Action 类 ActionSupport,

该类中有默认业务方法 execute, 返回 success。

2、action 的 method 属性可以省略, 省略时 Struts2

会自动调用 execute 方法。

```
-->
<action name="toLogin">
    <result name="success">
        /WEB-INF/main/login.jsp
    </result>
</action>
<!-- 登录校验 -->
<action name="login" class="com.netctoss.action.LoginAction">
    <!-- 校验成功, 跳转到系统首页 -->
    <result name="success">
        /WEB-INF/main/index.jsp
    </result>
    <!-- 登录失败, 跳转回登录页面 -->
    <result name="fail">
        /WEB-INF/main/login.jsp
    </result>
    <!-- 报错, 跳转到错误页面 -->
    <result name="error">
        /WEB-INF/main/error.jsp
    </result>
</action>
<!-- 生成验证码 -->
<action name="createImage"
class="com.netctoss.action.CreateImageAction">
    <!-- 使用 stream 类型的 result -->
    <result name="success" type="stream">
        <!-- 指定输出的内容 -->
        <param name="inputName">imageStream</param>
    </result>
</action>
</package>

</struts>
```

步骤六：重构资费 DAO 实现类，去掉关闭 session

重构 CostDaoImpl, 将 session 关闭的代码注释掉, 统一由拦截器关闭。重构后代码如下：

```
package com.netctoss.dao;

import java.util.List;
import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;
import com.netctoss.entity.Cost;
import com.netctoss.util.HibernateUtil;

/**
 * 当前阶段学习重点是 Struts2, 对于 DAO 的实现就模拟实现了。
 * 同学们可以使用 JDBC/MyBatis 自行实现该 DAO。
 */
public class CostDaoImpl implements ICostDao {
```

```

@Override
public List<Cost> findAll() {
    String hql = "from Cost";
    Session session = HibernateUtil.getSession();
    Query query = session.createQuery(hql);
    List<Cost> list = query.list();

    //      session.close();

    return list;
}

@Override
public void delete(int id) {
    Cost cost = new Cost();
    cost.setId(id);

    Session session = HibernateUtil.getSession();
    Transaction ts = session.beginTransaction();
    try {
        session.delete(cost);
        ts.commit();
    } catch (HibernateException e) {
        e.printStackTrace();
        ts.rollback();
    } finally {

    //      session.close();

    }
}

@Override
public Cost findByName(String name) {
    // 模拟根据名称查询资费数据，假设资费表中只有一条名为 tarena 的数据
    if("tarena".equals(name)) {
        Cost c = new Cost();
        c.setId(97);
        c.setName("tarena");
        c.setBaseDuration(99);
        c.setBaseCost(9.9);
        c.setUnitCost(0.9);
        c.setDescr("tarena 套餐");
        c.setStatus("0");
        c.setCostType("2");
        return c;
    }
    return null;
}

@Override
public Cost findById(int id) {
    Session session = HibernateUtil.getSession();
    Cost cost = (Cost) session.load(Cost.class, id);

    //      session.close();

    return cost;
}
}

```

步骤七：测试

重新部署项目，并启动 tomcat，访问资费修改功能，效果如下图，可以看出使用了 OpenSessionInViewInterceptor 之后，的确解决了 session 提前关闭引发的延迟加载问题：

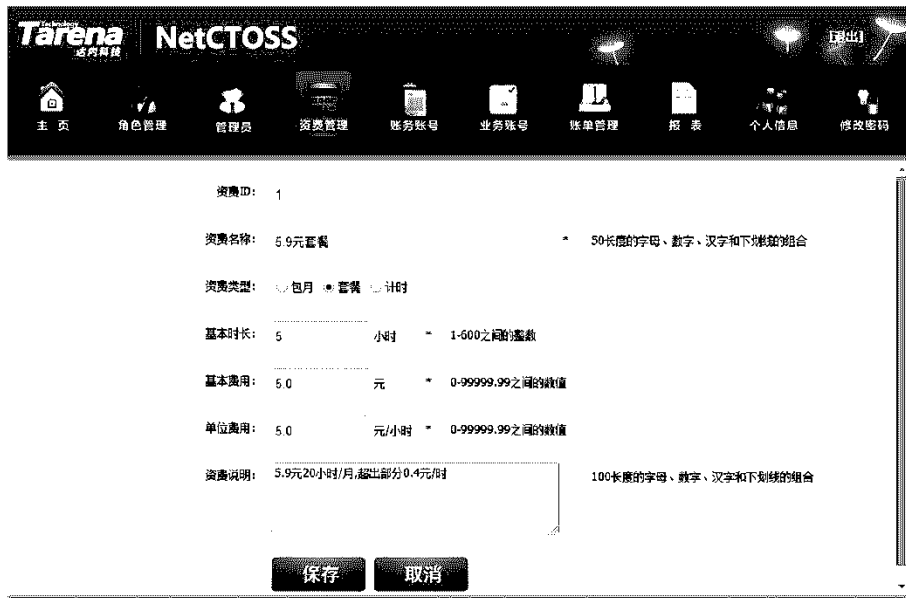


图-13

• 完整代码

以下是本案例的完整代码。

其中 CostDaoImpl 完整代码如下：

```
package com.netctoss.dao;

import java.util.List;
import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;
import com.netctoss.entity.Cost;
import com.netctoss.util.HibernateUtil;

/**
 * 当前阶段学习重点是 Struts2，对于 DAO 的实现就模拟实现了。
 * 同学们可以使用 JDBC/MyBatis 自行实现该 DAO。
 */
public class CostDaoImpl implements ICostDao {

    @Override
    public List<Cost> findAll() {
        String hql = "from Cost";
        Session session = HibernateUtil.getSession();
        Query query = session.createQuery(hql);
        List<Cost> list = query.list();
        // session.close();
        return list;
    }
}
```

```

@Override
public void delete(int id) {
    Cost cost = new Cost();
    cost.setId(id);

    Session session = HibernateUtil.getSession();
    Transaction ts = session.beginTransaction();
    try {
        session.delete(cost);
        ts.commit();
    } catch (HibernateException e) {
        e.printStackTrace();
        ts.rollback();
    } finally {
        session.close();
    }
}

@Override
public Cost findByName(String name) {
    // 模拟根据名称查询资费数据, 假设资费表中只有一条名为 tarena 的数据
    if("tarena".equals(name)) {
        Cost c = new Cost();
        c.setId(97);
        c.setName("tarena");
        c.setBaseDuration(99);
        c.setBaseCost(9.9);
        c.setUnitCost(0.9);
        c.setDescr("tarena 套餐");
        c.setStatus("0");
        c.setCostType("2");
        return c;
    }
    return null;
}

@Override
public Cost findById(int id) {
    Session session = HibernateUtil.getSession();
    Cost cost = (Cost) session.load(Cost.class, id);
    // session.close();
    return cost;
}
}

```

HibernateUtil 完整代码如下：

```

package com.netctoss.util;

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

public class HibernateUtil {

    private static SessionFactory sessionFactory;
    /**
     * 使用 ThreadLocal 管理 Session, 可以保证一个线程中只有唯一的一个连接。
     * 并且我们在获取连接时, 它会自动的给我们返回当前线程对应的连接。
     */
    private static ThreadLocal<Session> tl =
        new ThreadLocal<Session>();

    static {

```

```
// 加载 Hibernate 主配置文件
Configuration conf = new Configuration();
conf.configure("/hibernate.cfg.xml");
sessionFactory = conf.buildSessionFactory();
}

/**
 * 创建 session
 */
public static Session getSession() {
    // ThreadLocal 会以当前线程名为 key 获取连接
    Session session = tl.get();
    // 如果取到的当前线程的连接为空
    if(session == null) {
        // 使用工厂创建连接
        session = sessionFactory.openSession();
        // ThreadLocal 会以当前线程名为 key 保存 session
        tl.set(session);
    }
    return session;
}

/**
 * 关闭 session
 */
public static void close() {
    // ThreadLocal 会以当前线程名为 key 获取连接
    Session session = tl.get();
    // 如果取到的当前线程的连接不为空
    if(session != null) {
        // 关闭 session
        session.close();
        // 将当前线程对应的连接从 ThreadLocal 中移除
        tl.remove();
    }
}

public static void main(String[] args) {
    System.out.println(getSession());
    close();
}
}
```

OpenSessionInViewInterceptor 完整代码如下：

```
package com.netctoss.interceptor;

import com.netctoss.util.HibernateUtil;
import com.opensymphony.xwork2.ActionInvocation;
import com.opensymphony.xwork2.interceptor.Interceptor;

/**
 * 保持 Session 在视图层开启的拦截器，主要作用
 * 是在执行完 JSP 之后再统一关闭 session。
 */
public class OpenSessionInViewInterceptor
    implements Interceptor {

    @Override
    public void destroy() {
    }
}
```

```
@Override
public void init() {

}

@Override
public String intercept(ActionInvocation ai)
    throws Exception {
    // 调用 action 和 result
    ai.invoke();
    /*
     * result 会把请求转发到页面，因此调用 result，
     * 就相当于调用 JSP，因此此处的代码是在 JSP 之后执行。
     */
    HibernateUtil.close();
    return null;
}
}
```

struts.xml 完整代码如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.1.7//EN"
    "http://struts.apache.org/dtds/struts-2.1.7.dtd">
<struts>

    <!-- 公共的包，封装了通用的拦截器、通用的 result -->
    <package name="netctoss" extends="json-default">
        <interceptors>
            <!-- 登录检查拦截器 -->
            <interceptor name="loginInterceptor"
                class="com.netctoss.interceptor.LoginInterceptor"/>
            <!-- 保持 session 开启拦截器 -->
            <interceptor name="openSessionInterceptor"
                class="com.netctoss.interceptor.OpenSessionInViewInterceptor"/>
            <!-- 登录检查拦截器栈 -->
            <interceptor-stack name="loginStack">
                <interceptor-ref name="loginInterceptor"/>
                <interceptor-ref name="openSessionInterceptor"/>
                <!-- 不要丢掉默认的拦截器栈，里面有很多 Struts2 依赖的拦截器 -->
                <interceptor-ref name="defaultStack"/>
            </interceptor-stack>
        </interceptors>
        <!-- 设置 action 默认引用的拦截器 -->
        <default-interceptor-ref name="loginStack"/>
        <!-- 全局的 result，包下所有的 action 都可以共用 -->
        <global-results>
            <!-- 跳转到登录页面的 result -->
            <result name="login" type="redirectAction">
                <param name="namespace">/login</param>
                <param name="actionName">toLogin</param>
            </result>
        </global-results>
    </package>

    <!--
    资费模块配置信息：
    一般情况下，一个模块的配置单独封装在一个 package 下，
    并且以模块名来命名 package 的 name 和 namespace。
    -->
    <package name="cost" namespace="/cost" extends="netctoss">
```



```
<!-- 查询资费数据 -->
<action name="findCost" class="com.netctoss.action.FindCostAction">
    <!--
        正常情况下跳转到资费列表页面。
        一般一个模块的页面要打包在一个文件夹下，并且文件夹以模块名命名。
    -->
    <result name="success">
        /WEB-INF/cost/find_cost.jsp
    </result>
    <!--
        错误情况下，跳转到错误页面。
        错误页面可以被所有模块复用，因此放在 main 下，
        该文件夹用于存放公用的页面。
    -->
    <result name="error">
        /WEB-INF/main/error.jsp
    </result>
</action>
<!-- 删除资费 -->
<action name="deleteCost"
    class="com.netctoss.action.DeleteCostAction">
    <!-- 删除完之后，重定向到查询 action -->
    <result name="success" type="redirectAction">
        findCost
    </result>
    <result name="error">
        /WEB-INF/main/error.jsp
    </result>
</action>
<!-- 打开资费新增页 -->
<action name="toAddCost">
    <result name="success">
        /WEB-INF/cost/add_cost.jsp
    </result>
</action>
<!-- 资费名唯一性校验 -->
<action name="checkCostName"
    class="com.netctoss.action.CheckCostNameAction">
    <!-- 使用 json 类型的 result 把结果输出给回调函数 -->
    <result name="success" type="json">
        <param name="root">info</param>
    </result>
</action>
<!-- 打开修改页面 -->
<action name="toUpdateCost"
    class="com.netctoss.action.ToUpdateCostAction">
    <result name="success">
        /WEB-INF/cost/update_cost.jsp
    </result>
    <result name="error">
        /WEB-INF/main/error.jsp
    </result>
</action>
</package>

<!-- 登录模块 -->
<package name="login" namespace="/login" extends="struts-default">
    <!--
        打开登录页面：
        1、action 的 class 属性可以省略，省略时 Struts2
           会自动实例化默认的 Action 类 ActionSupport，
           该类中有默认业务方法 execute，返回 success。
        2、action 的 method 属性可以省略，省略时 Struts2
           会自动调用 execute 方法。
    -->
    <!--
```

```
<action name="toLogin">
    <result name="success">
        /WEB-INF/main/login.jsp
    </result>
</action>
<!-- 登录校验 -->
<action name="login" class="com.netctoss.action.LoginAction">
    <!-- 校验成功, 跳转到系统首页 -->
    <result name="success">
        /WEB-INF/main/index.jsp
    </result>
    <!-- 登录失败, 跳转回登录页面 -->
    <result name="fail">
        /WEB-INF/main/login.jsp
    </result>
    <!-- 报错, 跳转到错误页面 -->
    <result name="error">
        /WEB-INF/main/error.jsp
    </result>
</action>
<!-- 生成验证码 -->
<action name="createImage"
class="com.netctoss.action.CreateImageAction">
    <!-- 使用 stream 类型的 result -->
    <result name="success" type="stream">
        <!-- 指定输出的内容 -->
        <param name="inputName">imageStream</param>
    </result>
</action>
</package>

</struts>
```

6. 使用一对多关联映射

• 问题

使用一对多关联映射，在查询账务账号时，自动查询出它对应的全部业务账号。

• 方案

一对多关联映射开发步骤：

- 1) 账务账号与业务账号具有一对多关系，他们的关系字段是 service.account_id。
- 2) 在账务账号中追加集合属性，用于封装它对应的一组业务账号。
- 3) 在账务账号映射关系文件中配置此集合属性。

• 步骤

实现此案例需要按照如下步骤进行。

步骤一：创建账务账号实体类

创建账务账号实体类 Account，代码如下：

```
package com.tarena.entity;
```

```
import java.sql.Date;

public class Account {

    private Integer id;
    private Integer recommenderId;
    private String loginName;
    private String loginPassword;
    private String status;
    private Date createDate;
    private Date pauseDate;
    private Date closeDate;
    private String realName;
    private String idcardNo;
    private Date birthdate;
    private String gender;
    private String occupation;
    private String telephone;
    private String email;
    private String mailaddress;
    private String zipcode;
    private String qq;
    private Date lastLoginTime;
    private String lastLoginIp;

    public Integer getId() {
        return id;
    }

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

    public Integer getRecommenderId() {
        return recommenderId;
    }

    public void setRecommenderId(Integer recommenderId) {
        this.recommenderId = recommenderId;
    }

    public String getLoginName() {
        return loginName;
    }

    public void setLoginName(String loginName) {
        this.loginName = loginName;
    }

    public String getLoginPassword() {
        return loginPassword;
    }

    public void setLoginPassword(String loginPassword) {
        this.loginPassword = loginPassword;
    }

    public String getStatus() {
        return status;
    }

    public void setStatus(String status) {
        this.status = status;
    }

    public Date getCreateDate() {
        return createDate;
    }
}
```

```
public void setCreateDate(Date createDate) {
    this.createDate = createDate;
}

public Date getPauseDate() {
    return pauseDate;
}

public void setPauseDate(Date pauseDate) {
    this.pauseDate = pauseDate;
}

public Date getCloseDate() {
    return closeDate;
}

public void setCloseDate(Date closeDate) {
    this.closeDate = closeDate;
}

public String getRealName() {
    return realName;
}

public void setRealName(String realName) {
    this.realName = realName;
}

public String getIdcardNo() {
    return idcardNo;
}

public void setIdcardNo(String idcardNo) {
    this.idcardNo = idcardNo;
}

public Date getBirthdate() {
    return birthdate;
}

public void setBirthdate(Date birthdate) {
    this.birthdate = birthdate;
}

public String getGender() {
    return gender;
}

public void setGender(String gender) {
    this.gender = gender;
}

public String getOccupation() {
    return occupation;
}

public void setOccupation(String occupation) {
    this.occupation = occupation;
}

public String getTelephone() {
    return telephone;
}

public void setTelephone(String telephone) {
    this.telephone = telephone;
}
```

```
public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}

public String getMailaddress() {
    return mailaddress;
}

public void setMailaddress(String mailaddress) {
    this.mailaddress = mailaddress;
}

public String getZipcode() {
    return zipcode;
}

public void setZipcode(String zipcode) {
    this.zipcode = zipcode;
}

public String getQq() {
    return qq;
}

public void setQq(String qq) {
    this.qq = qq;
}

public Date getLastLoginTime() {
    return lastLoginTime;
}

public void setLastLoginTime(Date lastLoginTime) {
    this.lastLoginTime = lastLoginTime;
}

public String getLastLoginIp() {
    return lastLoginIp;
}

public void setLastLoginIp(String lastLoginIp) {
    this.lastLoginIp = lastLoginIp;
}
}
```

步骤二：创建账务账号映射关系文件

创建账务账号映射关系文件 Account.hbm.xml，代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<!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.tarena.entity.Account" table="ACCOUNT">
    <id name="id" type="integer" column="id">
        <generator class="sequence">
            <param name="sequence">ACCOUNT_SEQ</param>
        </generator>
    </id>
    <property name="recommenderId"
```

```

        type="integer" column="RECOMMENDER ID"/>
    <property name="loginName"
        type="string" column="LOGIN_NAME"/>
    <property name="loginPassword"
        type="string" column="LOGIN_PASSWD"/>
    <property name="status"
        type="string" column="STATUS"/>
    <property name="createDate"
        type="date" column="CREATE_DATE"/>
    <property name="pauseDate"
        type="date" column="PAUSE_DATE"/>
    <property name="closeDate"
        type="date" column="CLOSE DATE"/>
    <property name="realName"
        type="string" column="REAL NAME"/>
    <property name="idcardNo"
        type="string" column="IDCARD_NO"/>
    <property name="birthdate"
        type="date" column="BIRTHDATE"/>
    <property name="gender"
        type="string" column="GENDER"/>
    <property name="occupation"
        type="string" column="OCCUPATION"/>
    <property name="telephone"
        type="string" column="TELEPHONE"/>
    <property name="email"
        type="string" column="EMAIL"/>
    <property name="mailaddress"
        type="string" column="MAILADDRESS"/>
    <property name="zipcode"
        type="string" column="ZIPCODE"/>
    <property name="qq"
        type="string" column="QQ"/>
    <property name="lastLoginTime"
        type="date" column="LAST_LOGIN_TIME"/>
    <property name="lastLoginIp"
        type="string" column="LAST LOGIN IP"/>

</class>
</hibernate-mapping>

```

步骤三：创建业务账号实体类

创建业务账号实体类 Service，代码如下：

```

package com.tarena.entity;

import java.sql.Date;

public class Service {

    private Integer id;
    private Integer accountId;
    private String unixHost;
    private String osUserName;
    private String loginPassword;
    private String status;
    private Date createDate;
    private Date pauseDate;
    private Date closeDate;
    private Integer costId;

    public Integer getId() {
        return id;
    }
}

```

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

public Integer getAccountId() {
    return accountId;
}

public void setAccountId(Integer accountId) {
    this.accountId = accountId;
}

public String getUnixHost() {
    return unixHost;
}

public void setUnixHost(String unixHost) {
    this.unixHost = unixHost;
}

public String getOsUserName() {
    return osUserName;
}

public void setOsUserName(String osUserName) {
    this.osUserName = osUserName;
}

public String getLoginPassword() {
    return loginPassword;
}

public void setLoginPassword(String loginPassword) {
    this.loginPassword = loginPassword;
}

public String getStatus() {
    return status;
}

public void setStatus(String status) {
    this.status = status;
}

public Date getCreateDate() {
    return createDate;
}

public void setCreateDate(Date createDate) {
    this.createDate = createDate;
}

public Date getPauseDate() {
    return pauseDate;
}

public void setPauseDate(Date pauseDate) {
    this.pauseDate = pauseDate;
}

public Date getCloseDate() {
    return closeDate;
}

public void setCloseDate(Date closeDate) {
    this.closeDate = closeDate;
}
```

```
public Integer getCostId() {
    return costId;
}

public void setCostId(Integer costId) {
    this.costId = costId;
}

}
```

步骤四：创建业务账号映射关系文件

创建业务账号映射关系文件 Service.hbm.xml，代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<!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.tarena.entity.Service" table="SERVICE">
    <id name="id" type="integer" column="id">
        <generator class="sequence">
            <param name="sequence">SERVICE_SEQ</param>
        </generator>
    </id>
    <property name="accountId"
        type="integer" column="ACCOUNT ID"/>
    <property name="unixHost"
        type="string" column="UNIX_HOST"/>
    <property name="osUserName"
        type="string" column="OS_USERNAME"/>
    <property name="loginPassword"
        type="string" column="LOGIN PASSWD"/>
    <property name="status"
        type="string" column="STATUS"/>
    <property name="createDate"
        type="date" column="CREATE DATE"/>
    <property name="pauseDate"
        type="date" column="PAUSE DATE"/>
    <property name="closeDate"
        type="date" column="CLOSE DATE"/>
    <property name="costId"
        type="integer" column="COST_ID"/>
</class>
</hibernate-mapping>
```

步骤五：声明映射关系文件

在 hibernate.cfg.xml 中，声明账务账号和业务账号的映射关系文件，代码如下：

```
<?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>
    <!-- 数据库连接信息，根据自己的数据库进行配置 -->
    <property name="connection.url">
        jdbc:oracle:thin:@localhost:1521:xe
    </property>
    <property name="connection.username">lhh</property>
    <property name="connection.password">123456</property>
    <property name="connection.driver_class">
```



```
        oracle.jdbc.OracleDriver
    </property>

    <!-- Hibernate 配置信息 -->
    <!-- dialect 方言，用于配置生成针对哪个数据库的 SQL 语句 -->
    <property name="dialect">
        <!-- 方言类，Hibernate 提供的，用于封装某种特定数据库的方言 -->
        org.hibernate.dialect.OracleDialect
    </property>
    <!-- Hibernate 生成的 SQL 是否输出到控制台 -->
    <property name="show_sql">true</property>

    <!-- 将 SQL 输出时是否格式化。为了方便截图,我将其设置为 false -->
    <property name="format_sql">false</property>

    <!-- 声明映射关系文件 -->
    <mapping resource="com/tarena/entity/Emp.hbm.xml" />

    <mapping resource="com/tarena/entity/Account.hbm.xml" />
    <mapping resource="com/tarena/entity/Service.hbm.xml" />

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

步骤六：在账务账号实体类中追加集合属性

在账务账号实体类 Account 中，追加集合属性，用于封装它对应的所有业务账号，代码如下：

```
package com.tarena.entity;

import java.sql.Date;
import java.util.Set;

public class Account {

    private Integer id;
    private Integer recommenderId;
    private String loginName;
    private String loginPassword;
    private String status;
    private Date createDate;
    private Date pauseDate;
    private Date closeDate;
    private String realName;
    private String idcardNo;
    private Date birthdate;
    private String gender;
    private String occupation;
    private String telephone;
    private String email;
    private String mailaddress;
    private String zipcode;
    private String qq;
    private Date lastLoginTime;
    private String lastLoginIp;

    // 追加关联属性，用于存储相关的 Service 信息
```

```
private Set<Service> services;

public Set<Service> getServices() {
    return services;
}

public void setServices(Set<Service> services) {
    this.services = services;
}

public Integer getId() {
    return id;
}

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

public Integer getRecommenderId() {
    return recommenderId;
}

public void setRecommenderId(Integer recommenderId) {
    this.recommenderId = recommenderId;
}

public String getLoginName() {
    return loginName;
}

public void setLoginName(String loginName) {
    this.loginName = loginName;
}

public String getLoginPassword() {
    return loginPassword;
}

public void setLoginPassword(String loginPassword) {
    this.loginPassword = loginPassword;
}

public String getStatus() {
    return status;
}

public void setStatus(String status) {
    this.status = status;
}

public Date getCreateDate() {
    return createDate;
}

public void setCreateDate(Date createDate) {
    this.createDate = createDate;
}

public Date getPauseDate() {
    return pauseDate;
}

public void setPauseDate(Date pauseDate) {
    this.pauseDate = pauseDate;
}

public Date getCloseDate() {
```

```
        return closeDate;
    }

    public void setCloseDate(Date closeDate) {
        this.closeDate = closeDate;
    }

    public String getRealName() {
        return realName;
    }

    public void setRealName(String realName) {
        this.realName = realName;
    }

    public String getIdcardNo() {
        return idcardNo;
    }

    public void setIdcardNo(String idcardNo) {
        this.idcardNo = idcardNo;
    }

    public Date getBirthdate() {
        return birthdate;
    }

    public void setBirthdate(Date birthdate) {
        this.birthdate = birthdate;
    }

    public String getGender() {
        return gender;
    }

    public void setGender(String gender) {
        this.gender = gender;
    }

    public String getOccupation() {
        return occupation;
    }

    public void setOccupation(String occupation) {
        this.occupation = occupation;
    }

    public String getTelephone() {
        return telephone;
    }

    public void setTelephone(String telephone) {
        this.telephone = telephone;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getMailaddress() {
        return mailaddress;
    }

    public void setMailaddress(String mailaddress) {
```

```

        this.mailaddress = mailaddress;
    }

    public String getZipcode() {
        return zipcode;
    }

    public void setZipcode(String zipcode) {
        this.zipcode = zipcode;
    }

    public String getQq() {
        return qq;
    }

    public void setQq(String qq) {
        this.qq = qq;
    }

    public Date getLastLoginTime() {
        return lastLoginTime;
    }

    public void setLastLoginTime(Date lastLoginTime) {
        this.lastLoginTime = lastLoginTime;
    }

    public String getLastLoginIp() {
        return lastLoginIp;
    }

    public void setLastLoginIp(String lastLoginIp) {
        this.lastLoginIp = lastLoginIp;
    }
}

```

步骤七：在账务账号映射关系文件中配置集合属性

在账务账号映射关系文件 Account.hbm.xml 中，配置追加的集合属性，代码如下：

```

<?xml version="1.0" encoding="utf-8"?>
<!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.tarena.entity.Account" table="ACCOUNT">
    <id name="id" type="integer" column="id">
        <generator class="sequence">
            <param name="sequence">ACCOUNT_SEQ</param>
        </generator>
    </id>
    <property name="recommenderId"
        type="integer" column="RECOMMENDER_ID"/>
    <property name="loginName"
        type="string" column="LOGIN_NAME"/>
    <property name="loginPassword"
        type="string" column="LOGIN_PASSWD"/>
    <property name="status"
        type="string" column="STATUS"/>
    <property name="createDate"
        type="date" column="CREATE_DATE"/>
    <property name="pauseDate"
        type="date" column="PAUSE_DATE"/>
    <property name="closeDate"
        type="date" column="CLOSE_DATE"/>

```

```
<property name="realName"
    type="string" column="REAL_NAME"/>
<property name="idcardNo"
    type="string" column="IDCARD_NO"/>
<property name="birthdate"
    type="date" column="BIRTHDATE"/>
<property name="gender"
    type="string" column="GENDER"/>
<property name="occupation"
    type="string" column="OCCUPATION"/>
<property name="telephone"
    type="string" column="TELEPHONE"/>
<property name="email"
    type="string" column="EMAIL"/>
<property name="mailaddress"
    type="string" column="MAILADDRESS"/>
<property name="zipcode"
    type="string" column="ZIPCODE"/>
<property name="qq"
    type="string" column="QQ"/>
<property name="lastLoginTime"
    type="date" column="LAST_LOGIN_TIME"/>
<property name="lastLoginIp"
    type="string" column="LAST_LOGIN_IP"/>

<!-- 配置 services 属性，采用一对多的关系 -->
<set name="services">
    <!-- 用于指定关联条件，写关联条件的外键字段 -->
    <key column="ACCOUNT_ID"/>
    <!-- 用于指定采用哪种关系，加载哪方数据 -->
    <one-to-many class="com.tarena.entity.Service"/>
</set>

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

步骤八：创建测试类

在 com.tarena.test 包下创建一个测试类 TestOneToMany，并在这个类中增加一个测试方法，查询出某条账务账号的数据，然后输出账务账号以及账务账号中追加的集合属性值，代码如下：

```
package com.tarena.test;

import java.util.Set;
import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Account;
import com.tarena.entity.Service;
import com.tarena.util.HibernateUtil;

public class TestOneToMany {

    @Test
    public void test1() {
        Session session = HibernateUtil.getSession();
        Account account =
            (Account) session.get(Account.class, 1011);
        System.out.println(
            account.getIdcardNo() + " , "
            + account.getRealName() + " , "

```

```
+ account.getBirthdate());

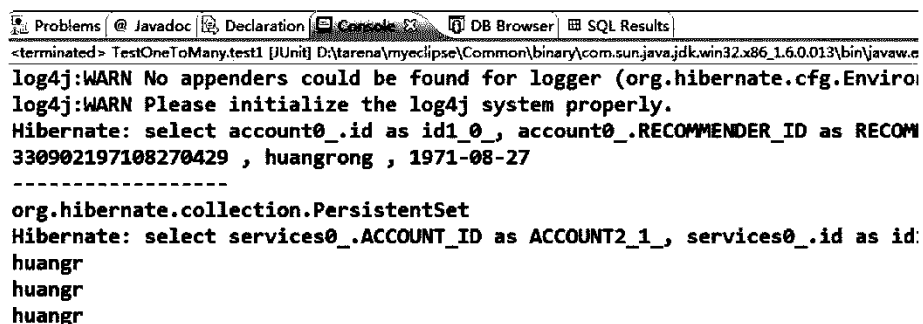
System.out.println("-----");

Set<Service> services = account.getServices();
System.out.println(services.getClass().getName());
for (Service service : services) {
    System.out.println(service.getOsUserName());
}

session.close();
}
}
```

步骤九：测试

执行这个方法，控制台输出结果如下图，可以看出查询账务账号之后，Hibernate 自动查询出了 SERVICE 数据并封装到了 services 属性中，并且对于 SERVICE 的查询是采用延迟加载机制的。



```
Problems @ Javadoc Declaration Console DB Browser SQL Results
<terminated> TestOneToMany.test1 [JUnit] D:\tarena\myeclipse\Common\binary\com.sun.java.jdk.win32.x86_1.6.0.013\bin\javaw.e
log4j:WARN No appenders could be found for logger (org.hibernate.cfg.Environment)
log4j:WARN Please initialize the log4j system properly.
Hibernate: select account0_.id as id1_0_, account0_.RECOMMENDER_ID as RECOMMENDER_ID from account0_ where account0_.id=330902197108270429
-----
org.hibernate.collection.PersistentSet
Hibernate: select services0_.ACCOUNT_ID as ACCOUNT2_1_, services0_.id as id1_0_ from services0_ where services0_.ACCOUNT_ID=330902197108270429
huangr
huangr
huangr
```

图-14

• 完整代码

以下为本案例的完整代码。

其中账务账号实体类 Account 完整代码如下：

```
package com.tarena.entity;

import java.sql.Date;
import java.util.Set;

public class Account {

    private Integer id;
    private Integer recommenderId;
    private String loginName;
    private String loginPassword;
    private String status;
    private Date createDate;
    private Date pauseDate;
    private Date closeDate;
    private String realName;
    private String idcardNo;
    private Date birthdate;
    private String gender;
    private String occupation;
```

```
private String telephone;
private String email;
private String mailaddress;
private String zipcode;
private String qq;
private Date lastLoginTime;
private String lastLoginIp;
// 追加关联属性, 用于存储相关的 Service 信息
private Set<Service> services;

public Set<Service> getServices() {
    return services;
}

public void setServices(Set<Service> services) {
    this.services = services;
}

public Integer getId() {
    return id;
}

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

public Integer getRecommenderId() {
    return recommenderId;
}

public void setRecommenderId(Integer recommenderId) {
    this.recommenderId = recommenderId;
}

public String getLoginName() {
    return loginName;
}

public void setLoginName(String loginName) {
    this.loginName = loginName;
}

public String getLoginPassword() {
    return loginPassword;
}

public void setLoginPassword(String loginPassword) {
    this.loginPassword = loginPassword;
}

public String getStatus() {
    return status;
}

public void setStatus(String status) {
    this.status = status;
}

public Date getCreateDate() {
    return createDate;
}

public void setCreateDate(Date createDate) {
    this.createDate = createDate;
}

public Date getPauseDate() {
    return pauseDate;
}
```

```
}

public void setPauseDate(Date pauseDate) {
    this.pauseDate = pauseDate;
}

public Date getCloseDate() {
    return closeDate;
}

public void setCloseDate(Date closeDate) {
    this.closeDate = closeDate;
}

public String getRealName() {
    return realName;
}

public void setRealName(String realName) {
    this.realName = realName;
}

public String getIdcardNo() {
    return idcardNo;
}

public void setIdcardNo(String idcardNo) {
    this.idcardNo = idcardNo;
}

public Date getBirthdate() {
    return birthdate;
}

public void setBirthdate(Date birthdate) {
    this.birthdate = birthdate;
}

public String getGender() {
    return gender;
}

public void setGender(String gender) {
    this.gender = gender;
}

public String getOccupation() {
    return occupation;
}

public void setOccupation(String occupation) {
    this.occupation = occupation;
}

public String getTelephone() {
    return telephone;
}

public void setTelephone(String telephone) {
    this.telephone = telephone;
}

public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}
```



```
}

public String getMailaddress() {
    return mailaddress;
}

public void setMailaddress(String mailaddress) {
    this.mailaddress = mailaddress;
}

public String getZipcode() {
    return zipcode;
}

public void setZipcode(String zipcode) {
    this.zipcode = zipcode;
}

public String getQq() {
    return qq;
}

public void setQq(String qq) {
    this.qq = qq;
}

public Date getLastLoginTime() {
    return lastLoginTime;
}

public void setLastLoginTime(Date lastLoginTime) {
    this.lastLoginTime = lastLoginTime;
}

public String getLastLoginIp() {
    return lastLoginIp;
}

public void setLastLoginIp(String lastLoginIp) {
    this.lastLoginIp = lastLoginIp;
}
}
```

账务账号映射关系文件 Account.hbm.xml 完整代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<!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.tarena.entity.Account" table="ACCOUNT">
    <id name="id" type="integer" column="id">
        <generator class="sequence">
            <param name="sequence">ACCOUNT_SEQ</param>
        </generator>
    </id>
    <property name="recommenderId"
        type="integer" column="RECOMMENDER_ID"/>
    <property name="loginName"
        type="string" column="LOGIN_NAME"/>
    <property name="loginPassword"
        type="string" column="LOGIN_PASSWD"/>
    <property name="status"
        type="string" column="STATUS"/>
    <property name="createDate"
```

```

        type="date" column="CREATE_DATE"/>
    <property name="pauseDate"
        type="date" column="PAUSE_DATE"/>
    <property name="closeDate"
        type="date" column="CLOSE_DATE"/>
    <property name="realName"
        type="string" column="REAL_NAME"/>
    <property name="idcardNo"
        type="string" column="IDCARD_NO"/>
    <property name="birthdate"
        type="date" column="BIRTHDATE"/>
    <property name="gender"
        type="string" column="GENDER"/>
    <property name="occupation"
        type="string" column="OCCUPATION"/>
    <property name="telephone"
        type="string" column="TELEPHONE"/>
    <property name="email"
        type="string" column="EMAIL"/>
    <property name="mailaddress"
        type="string" column="MAILADDRESS"/>
    <property name="zipcode"
        type="string" column="ZIPCODE"/>
    <property name="qq"
        type="string" column="QQ"/>
    <property name="lastLoginTime"
        type="date" column="LAST_LOGIN_TIME"/>
    <property name="lastLoginIp"
        type="string" column="LAST_LOGIN_IP"/>

    <!-- 配置 services 属性，采用一对多的关系 -->
    <set name="services">
        <!-- 用于指定关联条件，写关联条件的外键字段 -->
        <key column="ACCOUNT_ID"/>
        <!-- 用于指定采用哪种关系，加载哪方数据 -->
        <one-to-many class="com.tarena.entity.Service"/>
    </set>
</class>
</hibernate-mapping>

```

业务账号实体类 Service 完整代码如下：

```

package com.tarena.entity;

import java.sql.Date;

public class Service {

    private Integer id;
    private Integer accountId;
    private String unixHost;
    private String osUserName;
    private String loginPassword;
    private String status;
    private Date createDate;
    private Date pauseDate;
    private Date closeDate;
    private Integer costId;

    public Integer getId() {
        return id;
    }

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

```

```
}

public Integer getAccountId() {
    return accountId;
}

public void setAccountId(Integer accountId) {
    this.accountId = accountId;
}

public String getUnixHost() {
    return unixHost;
}

public void setUnixHost(String unixHost) {
    this.unixHost = unixHost;
}

public String getOsUserName() {
    return osUserName;
}

public void setOsUserName(String osUserName) {
    this.osUserName = osUserName;
}

public String getLoginPassword() {
    return loginPassword;
}

public void setLoginPassword(String loginPassword) {
    this.loginPassword = loginPassword;
}

public String getStatus() {
    return status;
}

public void setStatus(String status) {
    this.status = status;
}

public Date getCreateDate() {
    return createDate;
}

public void setCreateDate(Date createDate) {
    this.createDate = createDate;
}

public Date getPauseDate() {
    return pauseDate;
}

public void setPauseDate(Date pauseDate) {
    this.pauseDate = pauseDate;
}

public Date getCloseDate() {
    return closeDate;
}

public void setCloseDate(Date closeDate) {
    this.closeDate = closeDate;
}

public Integer getCostId() {
    return costId;
}
```

```

    }

    public void setCostId(Integer costId) {
        this.costId = costId;
    }

}

```

业务账号实体类 Service.hbm.xml 完整代码如下：

```

<?xml version="1.0" encoding="utf-8"?>
<!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.tarena.entity.Service" table="SERVICE">
    <id name="id" type="integer" column="id">
        <generator class="sequence">
            <param name="sequence">SERVICE_SEQ</param>
        </generator>
    </id>
    <property name="accountId"
        type="integer" column="ACCOUNT_ID"/>
    <property name="unixHost"
        type="string" column="UNIX_HOST"/>
    <property name="osUserName"
        type="string" column="OS_USERNAME"/>
    <property name="loginPassword"
        type="string" column="LOGIN_PASSWD"/>
    <property name="status"
        type="string" column="STATUS"/>
    <property name="createDate"
        type="date" column="CREATE_DATE"/>
    <property name="pauseDate"
        type="date" column="PAUSE_DATE"/>
    <property name="closeDate"
        type="date" column="CLOSE_DATE"/>
    <property name="costId"
        type="integer" column="COST_ID"/>
</class>
</hibernate-mapping>

```

主配置文件 hibernate.cfg.xml 完整代码如下：

```

<?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>
    <!-- 数据库连接信息，根据自己的数据库进行配置 -->
    <property name="connection.url">
        jdbc:oracle:thin:@localhost:1521:xe
    </property>
    <property name="connection.username">lhh</property>
    <property name="connection.password">123456</property>
    <property name="connection.driver_class">
        oracle.jdbc.OracleDriver
    </property>

    <!-- Hibernate 配置信息 -->
    <!-- dialect 方言，用于配置生成针对哪个数据库的 SQL 语句 -->
    <property name="dialect">

```

```
<!-- 方言类, Hibernate 提供的, 用于封装某种特定数据库的方言 -->
    org.hibernate.dialect.OracleDialect
</property>
<!-- Hibernate 生成的 SQL 是否输出到控制台 -->
<property name="show_sql">true</property>
<!-- 将 SQL 输出时是否格式化。为了方便截图,我将其设置为 false -->
<property name="format_sql">false</property>

<!-- 声明映射关系文件 -->
<mapping resource="com/tarena/entity/Emp.hbm.xml" />
<mapping resource="com/tarena/entity/Account.hbm.xml" />
<mapping resource="com/tarena/entity/Service.hbm.xml" />
</session-factory>
</hibernate-configuration>
```

测试类 TestOneToMany 完整代码如下：

```
package com.tarena.test;

import java.util.Set;
import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Account;
import com.tarena.entity.Service;
import com.tarena.util.HibernateUtil;

public class TestOneToMany {

    @Test
    public void test1() {
        Session session = HibernateUtil.getSession();
        Account account =
            (Account) session.get(Account.class, 1011);
        System.out.println(
            account.getIdcardNo() + " , "
            + account.getRealName() + " , "
            + account.getBirthdate());

        System.out.println("-----");

        Set<Service> services = account.getServices();
        System.out.println(services.getClass().getName());
        for (Service service : services) {
            System.out.println(service.getOsUserName());
        }

        session.close();
    }
}
```

课后作业

1. Hibernate 中对象有哪几种状态，有什么规则
2. Hibernate 中哪些查询方法具有延迟加载机制
3. Hibernate 中有几种类型的关联映射，你掌握哪几种

Hibernate 核心

Unit03

知识体系.....Page 100

多对一关联	多对一关联介绍	什么是多对一关联
		多对一关联的作用
	实现步骤	明确关系字段
		在“多”方实体类中添加实体属性 在“多”方 hbm 文件中配置关联关系
关联操作	关联查询	延迟加载
		抓取策略
	级联操作	什么是级联操作
		级联添加/修改
		级联删除
Hibernate 查询	HQL 查询	控制反转
		按条件查询
		查询一部分字段
		分页查询
	其他查询	多表联合查询
		直接使用 SQL 查询
Hibernate 高级特性	二级缓存	使用 Criteria 查询
		什么是二级缓存
	查询缓存	如何使用二级缓存
		什么是查询缓存
		如何使用查询缓存

经典案例.....Page 109

使用多对一关联映射	在“多”方 hbm 文件中配置关联关系
关联查询的一些特性	延迟加载
	抓取策略
级联添加/修改	级联添加/修改
级联删除	级联删除
HQL 查询，按条件查询	按条件查询

HQL 查询，查询一部分字段	查询一部分字段
HQL 查询，分页查询	分页查询
HQL 查询，多表联合查询	多表联合查询
Hibernate 中的 SQL 查询	直接使用 SQL 查询
使用二级缓存	如何使用二级缓存
使用查询缓存	如何使用查询缓存

课后作业.....Page 171

1. 多对一关联

1.1. 多对一关联介绍

1.1.1. 【多对一关联介绍】什么是多对一关联

知识讲解

什么是多对一关联

- 如果2张表具有多对一的关系，希望在使用Hibernate操作“多”方数据时，可以自动关联操作“一”方数据，那么这种关联映射称之为多对一关联。

1.1.2. 【多对一关联介绍】多对一关联的作用

知识讲解

多对一关联的作用

- 可以通过“多”来操作“一”，包括
 - 通过查询“多”，自动查询“一”。
 - 通过新增/修改“多”，自动新增/修改“一”。
 - 通过删除“多”，自动删除“一”。
- 注
一般情况下，多对一关系中，“多”方数据从属于“一”，比如员工从属于部门。因此很少会根据“多”方数据来增删改“一”方数据，更多的是对“一”方数据的关联查询。

1.2. 实现步骤



1.2.1. 【实现步骤】明确关系字段

知识讲解



明确关系字段

- 以service与account为例
- service与account具有多对一关系，其关系字段是service.account_id

1.2.2. 【实现步骤】在“多”方实体类中添加实体属性

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>在“多”方实体类中添加实体属性</h4> <ul style="list-style-type: none"> 在实体类Service中添加实体属性，以及get、set方法 <pre>private Account account; public Account getAccount() { return this.account; } public void setAccount(Account account) { this.account=account; }</pre> <div style="text-align: right;">  </div>
---	--



1.2.3. 【实现步骤】在“多”方 hbm 文件中配置关联关系

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>在“多”方hbm文件中配置关联关系</h4> <ul style="list-style-type: none"> 在Account.hbm.xml中配置与服务Service的关联关系 <pre><!-- many-to-one指定了关联关系， name指定了属性名， column指定了关系字段， class指定了另一方的类型。 --> <many-to-one name= "account" column= "account_id" class= "com.tarena.entity.Account" /></pre> <div style="text-align: right;">  </div>
---	--

2. 关联操作

2.1. 关联查询

2.1.1. 【关联查询】延迟加载

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>延迟加载</h4> <ul style="list-style-type: none"> 默认情况下，关联属性是采用延迟加载机制进行加载的。可以通过映射关系文件中关联属性配置标签中的lazy属性进行修改： <ul style="list-style-type: none"> lazy= "true" ，表示采用延迟加载。 lazy= "false" ，表示不采用延迟加载。 <div style="text-align: right;">  </div>
---	---

2.1.2. 【关联查询】抓取策略

<div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div>	<div><div>抓取策略</div><div><div>Tarena 达内科技</div><ul style="list-style-type: none">• 由于2张表具有关联关系，实际上可以通过一个连接查询一次性取出2张表的数据，避免进行2次查询。使用连接查询需要在映射关系文件中，关联属性标签上通过fetch属性进行设置，该设置通常称之为抓取策略<ul style="list-style-type: none">- fetch= "join"，表示在查询时使用连接查询，一起把对方数据抓取过来。- fetch= "select"，表示在查询时不使用连接查询，是默认的情况。• 注 当fetch= "join" 时，关联属性的延迟加载失效。</div><div><div>关联策略</div><div>+</div></div></div>
---	---

2.2. 级联操作



2.2.1. 【级联操作】什么是级联操作

<div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div>	<div><div>什么是级联操作</div><div><div>Tarena 达内科技</div><ul style="list-style-type: none">• Hibernate中，通过关联映射，在对一方进行增、删、改时，连带增、删、改关联的另一方数据，这种关联操作称之为级联操作。</div><div><div>关联策略</div><div>+</div></div></div>
---	--



2.2.2. 【级联操作】级联添加/修改

<div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div> <div></div>	<div><div>级联添加/修改</div><div><div>Tarena 达内科技</div><ul style="list-style-type: none">• 要实现级联添加/修改，需要在映射关系文件中的关联属性标签中，通过cascade属性进行设置，即 cascade= "save-update"</div><div><div>关联策略</div><div>+</div></div></div>
---	---

2.2.3. 【级联操作】级联删除

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>级联删除</h4> <ul style="list-style-type: none"> 要实现级联删除，需要在映射关系文件中的关联属性标签中，通过cascade属性进行设置，即 cascade= "delete" 注 若想级联添加、修改、删除一起支持，需要将cascade属性设置为all，即 cascade= "all" <div style="text-align: right;">  </div>
---	--



2.2.4. 【级联操作】控制反转

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>控制反转</h4> <ul style="list-style-type: none"> 在一对多关联中，使用级联新增、删除时，当前操作的“一”方会试图维护关联字段。然而关联字段是在“多”方对象中，它会自动维护这个字段，因此“一”方没必要做这样的处理。 要想让“一”方放弃这个处理，需要在它的映射关系文件中，关联属性配置标签上，通过inverse属性设置 <ul style="list-style-type: none"> inverse= "true"，表示控制反转，即交出控制权，“一”方将不再维护关联字段。 inverse= "false"，表示不控制反转，即不交出控制权，“一”方将维护关联字段，这是默认的情况。 <div style="text-align: right;">  </div>
---	--


3. Hibernate 查询

3.1. HQL 查询


3.1.1. 【HQL 查询】按条件查询


<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>按条件查询</h4> <ul style="list-style-type: none"> HQL中可以追加查询条件，条件中写的是属性名，之后在执行查询前用query对象为条件参数赋值，如 String hql = "from Service where unixHost=?"; Session session = HibernateUtil.getSession(); Query query = session.createQuery(hql); query.setString(0, "192.168.0.23"); List<Service> services = query.list(); <div style="text-align: right;">  </div>
---	--

3.1.2. 【HQL 查询】查询一部分字段

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>查询一部分字段</h4> <ul style="list-style-type: none"> 使用HQL查询时，可以只查询表中的一部分字段，需要在from之前追加select语句，并且明确指定要查询的列对应的属性名，如 <pre>String hql = "select id,unixHost,osUserName " + "from Service";</pre> 注意 当查询一部分字段时，query.list() 方法返回的集合中封装的不再是实体对象，而是一个Object[]，数组中的值与select语句后面的属性按顺序对应。 <div style="text-align: right;">+</div>
---	---


3.1.3. 【HQL 查询】分页查询


<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>分页查询</h4> <ul style="list-style-type: none"> Hibernate的分页查询不是通过HQL条件实现，而是通过API统一实现，需要通过API设置出分页的起点和每页显示的行数，如 <pre>int from = (page-1)*pageSize; query.setFirstResult(from); query.setMaxResults(pageSize);</pre> 注意 <ul style="list-style-type: none"> 查询的起点是本页第一行，按照JDBC计算，公式为 $(page-1)*pageSize+1$ Hibernate中行数的起点从0开始计算起，不同于JDBC中的从1开始计算，所以计算公式应该是在JDBC的计算公式基础上-1，即 $(page-1)*pageSize$ <div style="text-align: right;">+</div>
---	--


<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>分页查询（续1）</h4> <ul style="list-style-type: none"> 查询总页数 查询总页数比较简单，执行如下hql查询总行数，再根据总行数计算总页数即可 <pre>String hql = "select count(*) from Cost";</pre> <div style="text-align: right;">+</div>
---	--

3.1.4. 【HQL 查询】多表联合查询

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3 style="text-align: center;">多表联合查询</h3> <ul style="list-style-type: none"> • Hibernate支持使用HQL进行多表联合查询，不过HQL中写的是关联的对象及属性名。 • Hibernate中有3种使用HQL实现关联查询的方式，分别是 <ul style="list-style-type: none"> – 对象方式关联 – join方式关联 – select子句关联 <div style="text-align: right;">+</div>
---	---

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3 style="text-align: center;">多表联合查询（续1）</h3> <ul style="list-style-type: none"> • 对象方式关联，参考如下代码 <pre>String hql = "select s.id, s.osUserName," + "s.unixHost, a.id, a.realName, a.idcardNo " + "from Service s, Account a " + "where s.account.id=a.id";</pre> <div style="text-align: right;">+</div>
---	---


<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3 style="text-align: center;">多表联合查询（续2）</h3> <ul style="list-style-type: none"> • join方式关联，参考如下代码 <pre>String hql = "select s.id, s.osUserName," + "s.unixHost, a.id, a.realName, a.idcardNo " + "from Account a inner join a.services s ";</pre> • 注意 <p>join时，不能直接join对象，需要join关联属性。</p> <div style="text-align: right;">+</div>
---	---

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>多表联合查询（续3）</h3> <ul style="list-style-type: none"> select子句关联，参考如下代码 <pre>String hql = "select id, osUserName, " + "unixHost, account.id, account.realName, " + "account.idcardNo " + "from Service ";</pre> <div style="text-align: right;">+</div>
---	---


3.2. 其他查询

3.2.1. 【其他查询】直接使用 SQL 查询

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>直接使用SQL查询</h3> <ul style="list-style-type: none"> 如果业务太复杂，无法通过HQL实现查询功能，Hibernate也支持直接使用SQL进行查询，参考代码如下 <pre>String sql = "select * from SERVICE" + "where os_username=?"; Session session = HibernateUtil.getSession(); SQLQuery query = session.createSQLQuery(sql); query.setString(0, "huangr"); List<Object[]> list = query.list();</pre> <ul style="list-style-type: none"> 注意 本次查询返回的集合中，封装的是Object[]。 <div style="text-align: right;">+</div>
---	--

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>直接使用SQL查询（续1）</h3> <ul style="list-style-type: none"> 如果希望返回的集合中封装实体对象，需做如下处理 <pre>String sql = "select * from SERVICE" + "where os_username=?"; Session session = HibernateUtil.getSession(); SQLQuery query = session.createSQLQuery(sql); query.setString(0, "huangr"); // 指定集合中封装的类型 query.addEntity(Service.class); List<Service> list = query.list();</pre> <div style="text-align: right;">+</div>
---	--


3.2.2. 【其他查询】使用 Criteria 查询

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>使用Criteria查询</h4> <ul style="list-style-type: none"> • Hibernate还支持使用一个Criteria的API来动态拼一个HQL，参考代码如下 <pre>Criteria c = session.createCriteria(Service.class); c.add(Restrictions.eq("unixHost", "192.168.0.26")).add(Restrictions.or(Restrictions.eq("osUserName", "guojing"), Restrictions.eq("osUserName", "huangr"))); List<Service> list = c.list();</pre> <div style="text-align: right;">  </div>
---	---


4. Hibernate 高级特性

4.1. 二级缓存

4.1.1. 【二级缓存】什么是二级缓存

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>什么是二级缓存</h4> <ul style="list-style-type: none"> • 二级缓存类似于一级缓存，可以缓存对象。但它是SessionFactory级别的缓存，由SessionFactory负责管理。因此二级缓存的数据是Session间共享的，不同的Session对象都可以共享二级缓存中的数据。 • 二级缓存的适用场景 <ul style="list-style-type: none"> - 对象数据频繁共享 - 数据变化频率低
---	--



4.1.2. 【二级缓存】如何使用二级缓存

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h4>如何使用二级缓存</h4> <ul style="list-style-type: none"> • 导入ehcache.jar • 在src下添加缓存配置文件ehcache.xml • 在hibernate.cfg.xml中开启二级缓存，指定采用的二级缓存驱动类 • 在要缓存的对象对应的映射关系文件中，开启当前对象的二级缓存支持，并指定缓存策略。
---	---



<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>如何使用二级缓存（续1）</h3> <ul style="list-style-type: none"> • 二级缓存策略 <ul style="list-style-type: none"> - 只读型（read-only） 缓存不会更新，适用于不会发生改变的数据。效率最高，事务隔离级别最低。 - 读写型（read-write） 缓存会在数据变化时更新，适用于变化的数据。 - 不严格读写型（nonstrict-read-write） 缓存会不定期更新，适用于变化频率低的数据。 - 事务型（transactional） 缓存会在数据变化时更新，并且支持事务。效率最低，事务隔离级别最高。 <div style="text-align: right;">  </div>
---	--

4.2. 查询缓存

4.2.1. 【查询缓存】什么是查询缓存

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>什么是查询缓存</h3> <ul style="list-style-type: none"> • 查询缓存依赖于二级缓存，可以理解为特殊的二级缓存，也是SessionFactory级别的，也是由SessionFactory负责维护。 • 查询缓存可以缓存任何查询到的结果。 • 查询缓存是以hql为key，缓存该hql查询到的整个结果。即如果执行2次同样的hql，那么第二次执行时，此次查询可以从查询缓存中取到第一次查询缓存的内容。 • 查询缓存的适用场景 频繁使用同样的sql做查询 <div style="text-align: right;">  </div>
---	---

4.2.2. 【查询缓存】如何使用查询缓存

<hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/> <hr/>	<div style="text-align: right;">  </div> <h3>如何使用查询缓存</h3> <ul style="list-style-type: none"> • 开启二级缓存 • 在hibernate.cfg.xml中，开启查询缓存 • 在查询代码执行前，指定开启查询缓存。 <div style="text-align: right;">  </div>
---	--

经典案例

1. 使用多对一关联映射

- 问题

使用多对一关联映射，在查询业务账号时，自动查询出它对应的账务账号。

- 方案

多对一关联映射开发步骤：

- 1) 业务账号与账务账号具有多对一关系，他们的关系字段是 service.account_id。
- 2) 在业务账号中追加 Account 类型的属性，用于封装它对应的唯一账务账号。
- 3) 在业务账号映射关系文件中配置此属性。

- 步骤

实现此案例需要按照如下步骤进行。

步骤一：在业务账号实体类中追加属性

在业务账号实体类 Service 中，追加 Account 类型的属性，用于封装它对应的唯一账务账号。由于这个属性包含了账务账号的 ID，因此 accountId 属性可以去掉了，实际上这个属性必须去掉，否则会报错。代码如下：

```
package com.tarena.entity;

import java.sql.Date;

public class Service {
    private Integer id;

    // private Integer accountId;

    private String unixHost;
    private String osUserName;
    private String loginPassword;
    private String status;
    private Date createDate;
    private Date pauseDate;
    private Date closeDate;
    private Integer costId;

    // 追加属性，用于封装对应的 Account 记录
    private Account account;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
```

```
this.id = id;
}

// public Integer getAccountId() {
//     return accountId;
// }
//
// public void setAccountId(Integer accountId) {
//     this.accountId = accountId;
// }

public Account getAccount() {
    return account;
}

public void setAccount(Account account) {
    this.account = account;
}

public String getUnixHost() {
    return unixHost;
}

public void setUnixHost(String unixHost) {
    this.unixHost = unixHost;
}

public String getOsUserName() {
    return osUserName;
}

public void setOsUserName(String osUserName) {
    this.osUserName = osUserName;
}

public String getLoginPassword() {
    return loginPassword;
}

public void setLoginPassword(String loginPassword) {
    this.loginPassword = loginPassword;
}

public String getStatus() {
    return status;
}

public void setStatus(String status) {
    this.status = status;
}

public Date getCreateDate() {
    return createDate;
}

public void setCreateDate(Date createDate) {
    this.createDate = createDate;
}

public Date getPauseDate() {
    return pauseDate;
}

public void setPauseDate(Date pauseDate) {
    this.pauseDate = pauseDate;
}
```

```
public Date getCloseDate() {
    return closeDate;
}

public void setCloseDate(Date closeDate) {
    this.closeDate = closeDate;
}

public Integer getCostId() {
    return costId;
}

public void setCostId(Integer costId) {
    this.costId = costId;
}
}
```

步骤二：在业务账号映射关系文件中配置这个属性

在业务账号映射关系文件 service.hbm.xml 中，配置这个关联属性，代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<!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.tarena.entity.Service" table="SERVICE">
    <id name="id" type="integer" column="id">
        <generator class="sequence">
            <param name="sequence">SERVICE_SEQ</param>
        </generator>
    </id>
    <!--
        由于 account 属性已经体现了业务账号与账务账号的关系，
        并且 account 属性可以包含账务账号 ID，因此 accountId 可以去掉，
        实际上这里必须去掉这个属性的配置，否则会报错。
    -->
    <!-- <property name="accountId"
        type="integer" column="ACCOUNT_ID"/> -->
    <property name="unixHost"
        type="string" column="UNIX_HOST"/>
    <property name="osUserName"
        type="string" column="OS_USERNAME"/>
    <property name="loginPassword"
        type="string" column="LOGIN_PASSWD"/>
    <property name="status"
        type="string" column="STATUS"/>
    <property name="createDate"
        type="date" column="CREATE_DATE"/>
    <property name="pauseDate"
        type="date" column="PAUSE_DATE"/>
    <property name="closeDate"
        type="date" column="CLOSE_DATE"/>
    <property name="costId"
        type="integer" column="COST_ID"/>

    <!-- 配置 account 属性，采用多对一关系加载相关的 account 内容 -->
    <many-to-one name="account" column="ACCOUNT_ID"
        class="com.tarena.entity.Account"/>
</class>
</hibernate-mapping>
```

步骤三：创建测试类

在 com.tarena.test 包下，创建一个测试类 TestManyToOne，并且增加一个测试方法。在方法中查询出一条业务账号数据，然后输出这个业务账号的一些属性，同时输出 account 属性的值。代码如下：

```
package com.tarena.test;

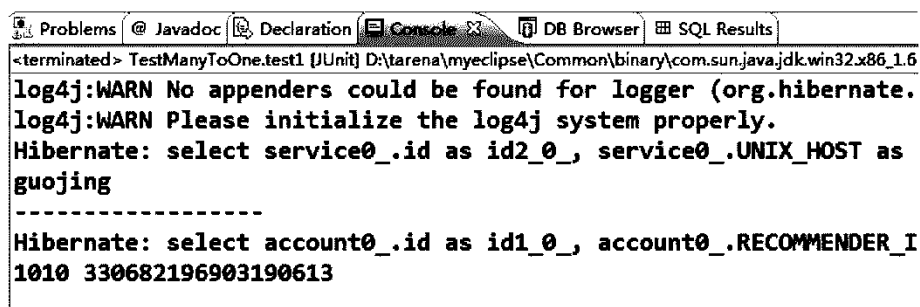
import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Service;
import com.tarena.util.HibernateUtil;

public class TestManyToOne {

    @Test
    public void test1() {
        Session session = HibernateUtil.getSession();
        Service service =
            (Service) session.get(Service.class, 2001);
        System.out.println(service.getUserName());
        System.out.println("-----");
        System.out.println(
            service.getAccount().getId() + " " +
            service.getAccount().getIdcardNo());
    }
}
```

步骤四：测试

执行这个测试方法，控制台输出结果如下图，可以看出在查询 SERVICE 的同时，Hibernate 自动查询出了它对应的 ACCOUNT 数据，并且这个关联查询是采用延迟加载机制实现的。



The screenshot shows the Eclipse IDE console with the following output:

```
<terminated> TestManyToOne.test1 [JUnit] D:\tarena\myeclipse\Common\binary\com.sun.java.jdk.win32.x86_1.6
log4j:WARN No appenders could be found for logger (org.hibernate.
log4j:WARN Please initialize the log4j system properly.
Hibernate: select service0_.id as id2_0_, service0_.UNIX_HOST as
guojing
-----
Hibernate: select account0_.id as id1_0_, account0_.RECOMMENDER_I
1010 330682196903190613
```

图-1

• 完整代码

以下为本案例的完整代码。

其中业务账号实体类完整代码如下：

```
package com.tarena.entity;

import java.sql.Date;
```

```
public class Service {

    private Integer id;
    // private Integer accountId;
    private String unixHost;
    private String osUserName;
    private String loginPassword;
    private String status;
    private Date createDate;
    private Date pauseDate;
    private Date closeDate;
    private Integer costId;
    // 追加属性, 用于封装关联的 Account 记录
    private Account account;

    public Integer getId() {
        return id;
    }

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

    // public Integer getAccountId() {
    //     return accountId;
    // }
    //
    // public void setAccountId(Integer accountId) {
    //     this.accountId = accountId;
    // }

    public Account getAccount() {
        return account;
    }

    public void setAccount(Account account) {
        this.account = account;
    }

    public String getUnixHost() {
        return unixHost;
    }

    public void setUnixHost(String unixHost) {
        this.unixHost = unixHost;
    }

    public String getOsUserName() {
        return osUserName;
    }

    public void setOsUserName(String osUserName) {
        this.osUserName = osUserName;
    }

    public String getLoginPassword() {
        return loginPassword;
    }

    public void setLoginPassword(String loginPassword) {
        this.loginPassword = loginPassword;
    }

    public String getStatus() {
        return status;
    }

    public void setStatus(String status) {
```

```

        this.status = status;
    }

    public Date getCreateDate() {
        return createDate;
    }

    public void setCreateDate(Date createDate) {
        this.createDate = createDate;
    }

    public Date getPauseDate() {
        return pauseDate;
    }

    public void setPauseDate(Date pauseDate) {
        this.pauseDate = pauseDate;
    }

    public Date getCloseDate() {
        return closeDate;
    }

    public void setCloseDate(Date closeDate) {
        this.closeDate = closeDate;
    }

    public Integer getCostId() {
        return costId;
    }

    public void setCostId(Integer costId) {
        this.costId = costId;
    }
}

```

业务账号映射关系文件代码如下：

```

<?xml version="1.0" encoding="utf-8"?>
<!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.tarena.entity.Service" table="SERVICE">
    <id name="id" type="integer" column="id">
        <generator class="sequence">
            <param name="sequence">SERVICE_SEQ</param>
        </generator>
    </id>
    <!--
        由于 account 属性已经体现了业务账号与账务账号的关系，
        并且 account 属性可以包含账务账号 ID，因此 accountId 可以去掉，
        实际上这里必须去掉这个属性的配置，否则会报错。
    -->
    <!-- <property name="accountId"
        type="integer" column="ACCOUNT_ID"/> -->
    <property name="unixHost"
        type="string" column="UNIX_HOST"/>
    <property name="osUserName"
        type="string" column="OS_USERNAME"/>
    <property name="loginPassword"
        type="string" column="LOGIN_PASSWD"/>
    <property name="status"
        type="string" column="STATUS"/>

```

```
<property name="createDate"
    type="date" column="CREATE DATE"/>
<property name="pauseDate"
    type="date" column="PAUSE_DATE"/>
<property name="closeDate"
    type="date" column="CLOSE DATE"/>
<property name="costId"
    type="integer" column="COST ID"/>

<!-- 配置 account 属性，采用多对一关系加载相关的 account 内容 -->
<many-to-one name="account" column="ACCOUNT_ID"
    class="com.tarena.entity.Account"/>
</class>
</hibernate-mapping>
```

测试类代码如下：

```
package com.tarena.test;

import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Service;
import com.tarena.util.HibernateUtil;

public class TestManyToOne {

    @Test
    public void test1() {
        Session session = HibernateUtil.getSession();
        Service service =
            (Service) session.get(Service.class, 2001);
        System.out.println(service.getOsUserName());
        System.out.println("-----");
        System.out.println(
            service.getAccount().getId() + " " +
            service.getAccount().getIdcardNo());
    }
}
```

2. 关联查询的一些特性

- 问题

请按照如下的方式使用关联映射：

- 1) 不采用延迟加载的方式查询关联属性。
- 2) 采用关联查询一次性查出 2 张表的数据。

- 方案

可以按照如下的方式实现上述问题的要求：

- 1) 通过 lazy=" false" 取消延迟加载。
- 2) 通过 fetch=" join" 设置关联查询。

- 步骤

实现此案例需要按照如下步骤进行。

步骤一：创建项目

复制项目 HibernateDay02，粘贴并修改项目名为 HibernateDay03。

步骤二：将一对多关联映射取消延迟加载

修改 HibernateDay03 项目中的 Account.hbm.xml 文件，将 services 属性的配置追加 lazy="false"，代码如下：

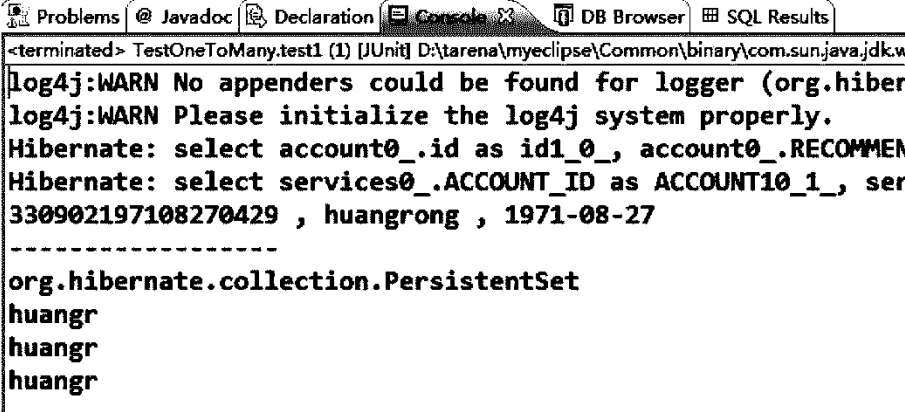
```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD
3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="com.tarena.entity.Account" table="ACCOUNT">
  <id name="id" type="integer" column="id">
    <generator class="sequence">
      <param name="sequence">ACCOUNT_SEQ</param>
    </generator>
  </id>
  <property name="recommenderId"
    type="integer" column="RECOMMENDER_ID"/>
  <property name="loginName"
    type="string" column="LOGIN_NAME"/>
  <property name="loginPassword"
    type="string" column="LOGIN_PASSWD"/>
  <property name="status"
    type="string" column="STATUS"/>
  <property name="createDate"
    type="date" column="CREATE_DATE"/>
  <property name="pauseDate"
    type="date" column="PAUSE_DATE"/>
  <property name="closeDate"
    type="date" column="CLOSE_DATE"/>
  <property name="realName"
    type="string" column="REAL_NAME"/>
  <property name="idcardNo"
    type="string" column="IDCARD_NO"/>
  <property name="birthdate"
    type="date" column="BIRTHDATE"/>
  <property name="gender"
    type="string" column="GENDER"/>
  <property name="occupation"
    type="string" column="OCCUPATION"/>
  <property name="telephone"
    type="string" column="TELEPHONE"/>
  <property name="email"
    type="string" column="EMAIL"/>
  <property name="mailaddress"
    type="string" column="MAILADDRESS"/>
  <property name="zipcode"
    type="string" column="ZIPCODE"/>
  <property name="qq"
    type="string" column="QQ"/>
  <property name="lastLoginTime"
    type="date" column="LAST_LOGIN_TIME"/>
  <property name="lastLoginIp"
    type="string" column="LAST_LOGIN_IP"/>

  <!-- 配置 services 属性，采用一对多的关系 -->
```

```
<set name="services" lazy="false">

    <!-- 用于指定关联条件，写关联条件的外键字段 -->
    <key column="ACCOUNT_ID"/>
    <!-- 用于指定采用哪种关系，加载哪方数据 -->
    <one-to-many class="com.tarena.entity.Service"/>
</set>
</class>
</hibernate-mapping>
```

执行 TestOneToMany 中的测试方法，控制台输出结果如下图，可以看出在查询账务账号之后 Hibernate 立刻查询了它对应的业务账号，已经取消了延迟加载。



```
Problems  @ Javadoc  Declaration  Console  DB Browser  SQL Results
<terminated> TestOneToMany.test1 (1) [Unit] D:\tarena\myeclipse\Common\binary\com.sun.java.jdk.w
log4j:WARN No appenders could be found for logger (org.hiber
log4j:WARN Please initialize the log4j system properly.
Hibernate: select account0_.id as id1_0_, account0_.RECOMMEN
Hibernate: select services0_.ACCOUNT_ID as ACCOUNT10_1_, ser
330902197108270429 , huangrong , 1971-08-27
-----
org.hibernate.collection.PersistentSet
huangr
huangr
huangr
```

图-2

步骤三：将多对一关联映射取消延迟加载

修改 Service.hbm.xml，将 account 属性的配置追加 lazy="false"，代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<!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.tarena.entity.Service" table="SERVICE">
    <id name="id" type="integer" column="id">
        <generator class="sequence">
            <param name="sequence">SERVICE_SEQ</param>
        </generator>
    </id>
    <!--
    由于 account_属性已经体现了业务账号与账务账号的关系，
    并且 account_属性可以包含账务账号 ID，因此 accountId 可以去掉，
    实际上这里必须去掉这个属性的配置，否则会报错。
    -->
    <!-- <property name="accountId"
        type="integer" column="ACCOUNT_ID"/> -->
    <property name="unixHost"
        type="string" column="UNIX_HOST"/>
    <property name="osUserName"
        type="string" column="OS_USERNAME"/>
    <property name="loginPassword"
        type="string" column="LOGIN_PASSWD"/>
    <property name="status"
        type="string" column="STATUS"/>
```

```
<property name="createDate"
    type="date" column="CREATE_DATE"/>
<property name="pauseDate"
    type="date" column="PAUSE_DATE"/>
<property name="closeDate"
    type="date" column="CLOSE_DATE"/>
<property name="costId"
    type="integer" column="COST_ID"/>

<!-- 配置 account 属性，采用多对一关系加载相关的 account 内容 -->

<many-to-one name="account" column="ACCOUNT_ID"
    class="com.tarena.entity.Account"
    lazy="false"/>

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

执行 TestManyToOne 中的测试方法，控制台输出效果如下图，可以看出在查询业务账号之后 Hibernate 立刻查询了对应的账务账号，取消了延迟加载。

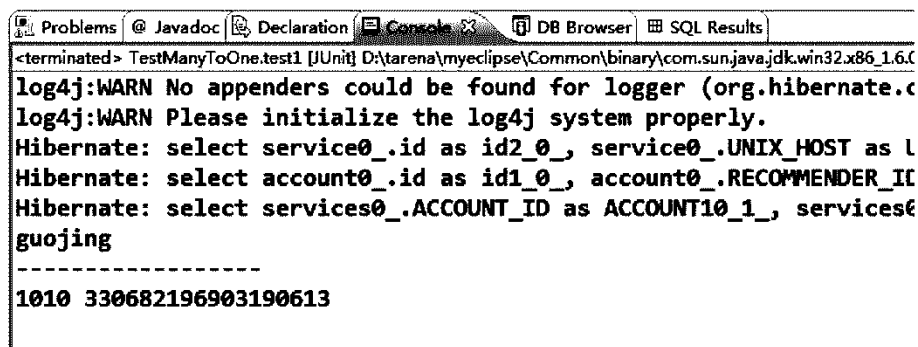


图-3

步骤四：将一对多关联映射设置为 join 方式抓取数据

修改 Account.hbm.xml，将 services 属性的配置追加 fetch="join"，代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<!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.tarena.entity.Account" table="ACCOUNT">
    <id name="id" type="integer" column="id">
        <generator class="sequence">
            <param name="sequence">ACCOUNT_SEQ</param>
        </generator>
    </id>
    <property name="recommenderId"
        type="integer" column="RECOMMENDER_ID"/>
    <property name="loginName"
        type="string" column="LOGIN_NAME"/>
    <property name="loginPassword"
        type="string" column="LOGIN_PASSWD"/>
    <property name="status"
        type="string" column="STATUS"/>
    <property name="createDate"
        type="date" column="CREATE_DATE"/>
    <property name="pauseDate"
```

```

        type="date" column="PAUSE_DATE"/>
    <property name="closeDate"
        type="date" column="CLOSE_DATE"/>
    <property name="realName"
        type="string" column="REAL_NAME"/>
    <property name="idcardNo"
        type="string" column="IDCARD_NO"/>
    <property name="birthdate"
        type="date" column="BIRTHDATE"/>
    <property name="gender"
        type="string" column="GENDER"/>
    <property name="occupation"
        type="string" column="OCCUPATION"/>
    <property name="telephone"
        type="string" column="TELEPHONE"/>
    <property name="email"
        type="string" column="EMAIL"/>
    <property name="mailaddress"
        type="string" column="MAILADDRESS"/>
    <property name="zipcode"
        type="string" column="ZIPCODE"/>
    <property name="qq"
        type="string" column="QQ"/>
    <property name="lastLoginTime"
        type="date" column="LAST_LOGIN_TIME"/>
    <property name="lastLoginIp"
        type="string" column="LAST_LOGIN_IP"/>

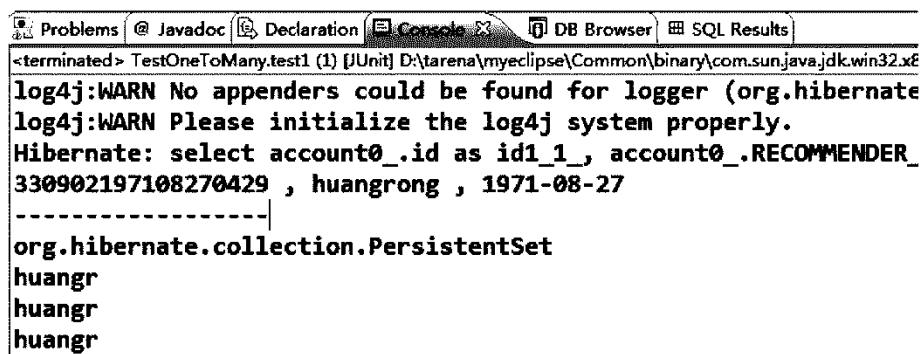
    <!-- 配置 services 属性，采用一对多的关系 -->

    <set name="services" lazy="false" fetch="join">

        <!-- 用于指定关联条件，写关联条件的外键字段 -->
        <key column="ACCOUNT_ID"/>
        <!-- 用于指定采用哪种关系，加载哪方数据 -->
        <one-to-many class="com.tarena.entity.Service"/>
    </set>
</class>
</hibernate-mapping>

```

执行 TestOneToMany 中的测试方法，控制台输出结果如下图，可以看出查询时采用了 join 方式查询。



```

<terminated> TestOneToMany.test1 (1) [JUnit] D:\tarena\myeclipse\Common\binary\com.sun.java.jdk.win32.x86
log4j:WARN No appenders could be found for logger (org.hibernate)
log4j:WARN Please initialize the log4j system properly.
Hibernate: select account0_.id as id1_1_, account0_.RECOMMENDER_
330902197108270429 , huangrong , 1971-08-27
-----
org.hibernate.collection.PersistentSet
huangr
huangr
huangr

```

图-4

步骤五：将多对一关联映射设置为 join 方式抓取数据

修改 Service.hbm.xml，将 account 属性追加 fetch="join"，代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<!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.tarena.entity.Service" table="SERVICE">
    <id name="id" type="integer" column="id">
      <generator class="sequence">
        <param name="sequence">SERVICE_SEQ</param>
      </generator>
    </id>
    <!--
      由于 account 属性已经体现了业务账号与账务账号的关系，
      并且 account 属性可以包含账务账号 ID，因此 accountId 可以去掉，
      实际上这里必须去掉这个属性的配置，否则会报错。
    -->
    <!-- <property name="accountId"
      type="integer" column="ACCOUNT_ID"/> -->
    <property name="unixHost"
      type="string" column="UNIX_HOST"/>
    <property name="osUserName"
      type="string" column="OS_USERNAME"/>
    <property name="loginPassword"
      type="string" column="LOGIN_PASSWD"/>
    <property name="status"
      type="string" column="STATUS"/>
    <property name="createDate"
      type="date" column="CREATE_DATE"/>
    <property name="pauseDate"
      type="date" column="PAUSE_DATE"/>
    <property name="closeDate"
      type="date" column="CLOSE_DATE"/>
    <property name="costId"
      type="integer" column="COST_ID"/>

    <!-- 配置 account 属性，采用多对一关系加载相关的 account 内容 -->

    <many-to-one name="account" column="ACCOUNT_ID"
      class="com.tarena.entity.Account"
      lazy="false" fetch="join"/>

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

执行 TestManyToOne 中的测试方法，控制台输出结果如下图，可以看出查询时采用了 join 方式查询。

```
<terminated> TestManyToOne.test1 [Unit] D:\tarena\myeclipse\Common\binary\com.sun.java.jdk.win32.x86_64
log4j:WARN No appenders could be found for logger (org.hibernate).
log4j:WARN Please initialize the log4j system properly.
Hibernate: select service0_.id as id2_1, service0_.UNIX_HOST
Hibernate: select services0_.ACCOUNT_ID as ACCOUNT10_1, servi
guojing
-----
1010 330682196903190613
```

图-5

• 完整代码

以下为本案例的完整代码。

其中账务账号映射关系文件完整代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<!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.tarena.entity.Account" table="ACCOUNT">
<id name="id" type="integer" column="id">
<generator class="sequence">
<param name="sequence">ACCOUNT_SEQ</param>
</generator>
</id>
<property name="recommenderId"
type="integer" column="RECOMMENDER ID"/>
<property name="loginName"
type="string" column="LOGIN NAME"/>
<property name="loginPassword"
type="string" column="LOGIN_PASSWD"/>
<property name="status"
type="string" column="STATUS"/>
<property name="createDate"
type="date" column="CREATE DATE"/>
<property name="pauseDate"
type="date" column="PAUSE_DATE"/>
<property name="closeDate"
type="date" column="CLOSE_DATE"/>
<property name="realName"
type="string" column="REAL NAME"/>
<property name="idcardNo"
type="string" column="IDCARD_NO"/>
<property name="birthdate"
type="date" column="BIRTHDATE"/>
<property name="gender"
type="string" column="GENDER"/>
<property name="occupation"
type="string" column="OCCUPATION"/>
<property name="telephone"
type="string" column="TELEPHONE"/>
<property name="email"
type="string" column="EMAIL"/>
<property name="mailaddress"
type="string" column="MAILADDRESS"/>
<property name="zipcode"
type="string" column="ZIPCODE"/>
<property name="qq"
type="string" column="QQ"/>
<property name="lastLoginTime"
type="date" column="LAST_LOGIN_TIME"/>
<property name="lastLoginIp"
type="string" column="LAST_LOGIN_IP"/>

<!-- 配置 services 属性，采用一对多的关系 -->
<set name="services" lazy="false" fetch="join">
<!-- 用于指定关联条件，写关联条件的外键字段 -->
<key column="ACCOUNT ID"/>
<!-- 用于指定采用哪种关系，加载哪方数据 -->
<one-to-many class="com.tarena.entity.Service"/>
</set>
</class>
</hibernate-mapping>
```

业务账号映射关系文件完整代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<!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.tarena.entity.Service" table="SERVICE">
  <id name="id" type="integer" column="id">
    <generator class="sequence">
      <param name="sequence">SERVICE_SEQ</param>
    </generator>
  </id>
  <!--
    由于 account 属性已经体现了业务账号与账务账号的关系，
    并且 account 属性可以包含账务账号 ID，因此 accountId 可以去掉，
    实际上这里必须去掉这个属性的配置，否则会报错。
  -->
  <!-- <property name="accountId"
    type="integer" column="ACCOUNT_ID"/> -->
  <property name="unixHost"
    type="string" column="UNIX_HOST"/>
  <property name="osUserName"
    type="string" column="OS_USERNAME"/>
  <property name="loginPassword"
    type="string" column="LOGIN_PASSWD"/>
  <property name="status"
    type="string" column="STATUS"/>
  <property name="createDate"
    type="date" column="CREATE_DATE"/>
  <property name="pauseDate"
    type="date" column="PAUSE_DATE"/>
  <property name="closeDate"
    type="date" column="CLOSE_DATE"/>
  <property name="costId"
    type="integer" column="COST_ID"/>

  <!-- 配置 account 属性，采用多对一关系加载相关的 account 内容 -->
  <many-to-one name="account" column="ACCOUNT_ID"
    class="com.tarena.entity.Account"
    lazy="false" fetch="join"/>
</class>
</hibernate-mapping>
```

3. 级联添加/修改

- 问题

使用级联添加/修改，在添加/修改账务账号时，自动添加/修改其对应的业务账号。

- 方案

通过在映射关系文件中设置 cascade="save-update"，可以支持级联添加/修改。

- 步骤

实现此案例需要按照如下步骤进行。

步骤一：在账务账号映射关系文件中设置级联添加/修改

在账务账号映射关系文件 Account.hbm.xml 中，通过 cascade= "save-update" 设置支持级联添加/修改，代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<!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.tarena.entity.Account" table="ACCOUNT">
  <id name="id" type="integer" column="id">
    <generator class="sequence">
      <param name="sequence">ACCOUNT_SEQ</param>
    </generator>
  </id>
  <property name="recommenderId"
    type="integer" column="RECOMMENDER_ID"/>
  <property name="loginName"
    type="string" column="LOGIN_NAME"/>
  <property name="loginPassword"
    type="string" column="LOGIN_PASSWD"/>
  <property name="status"
    type="string" column="STATUS"/>
  <property name="createDate"
    type="date" column="CREATE_DATE"/>
  <property name="pauseDate"
    type="date" column="PAUSE_DATE"/>
  <property name="closeDate"
    type="date" column="CLOSE_DATE"/>
  <property name="realName"
    type="string" column="REAL_NAME"/>
  <property name="idcardNo"
    type="string" column="IDCARD_NO"/>
  <property name="birthdate"
    type="date" column="BIRTHDATE"/>
  <property name="gender"
    type="string" column="GENDER"/>
  <property name="occupation"
    type="string" column="OCCUPATION"/>
  <property name="telephone"
    type="string" column="TELEPHONE"/>
  <property name="email"
    type="string" column="EMAIL"/>
  <property name="mailaddress"
    type="string" column="MAILADDRESS"/>
  <property name="zipcode"
    type="string" column="ZIPCODE"/>
  <property name="qq"
    type="string" column="QQ"/>
  <property name="lastLoginTime"
    type="date" column="LAST_LOGIN_TIME"/>
  <property name="lastLoginIp"
    type="string" column="LAST_LOGIN_IP"/>

  <!-- 配置 services 属性，采用一对多的关系 -->

  <set name="services"
    lazy="false" fetch="join"
    cascade="save-update">

    <!-- 用于指定关联条件，写关联条件的外键字段 -->
    <key column="ACCOUNT_ID"/>
  </set>
</class>
</hibernate-mapping>
```



```
<!-- 用于指定采用哪种关系，加载哪方数据 -->
<one-to-many
    class="com.tarena.entity.Service"/>
</set>
</class>
</hibernate-mapping>
```

步骤二：测试级联添加

在 `com.tarena.test` 包下，创建测试类 `TestCascade`，并在类中增加测试级联添加的方法，代码如下：

```
package com.tarena.test;

import java.util.HashSet;
import java.util.Set;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.junit.Test;
import com.tarena.entity.Account;
import com.tarena.entity.Service;
import com.tarena.util.HibernateUtil;

public class TestCascade {

    /**
     * 级联添加
     */
    @Test
    public void test1() {
        // 模拟要添加的账务账号
        Account a = new Account();
        a.setLoginName("gg");
        a.setLoginPassword("123");
        a.setRealName("gg");
        a.setIdcardNo("120392198410282549");
        a.setStatus("0");
        a.setTelephone("110");

        // 模拟要添加的业务账号
        Service s1 = new Service();
        s1.setAccount(a);
        s1.setOsUserName("gg1");
        s1.setLoginPassword("123");
        s1.setUnixHost("192.168.1.1");
        s1.setCostId(5);
        s1.setStatus("0");

        Service s2 = new Service();
        s2.setAccount(a);
        s2.setOsUserName("gg2");
        s2.setLoginPassword("123");
        s2.setUnixHost("192.168.1.2");
        s2.setCostId(5);
        s2.setStatus("0");

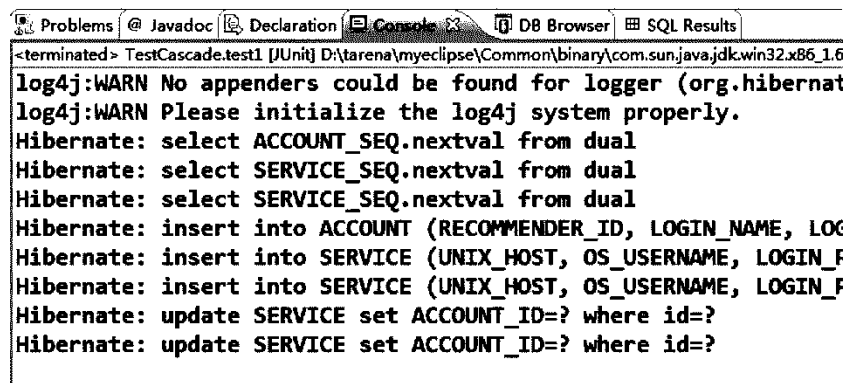
        // 将业务账号与账务账号关联
        a.setServices(new HashSet<Service>());
        a.getServices().add(s1);
        a.getServices().add(s2);
    }
}
```

```

Session session = HibernateUtil.getSession();
Transaction ts = session.beginTransaction();
try {
    session.save(a);
    ts.commit();
} catch (HibernateException e) {
    e.printStackTrace();
    ts.rollback();
} finally {
    session.close();
}
}
}

```

执行 test1() 方法, 控制台输出结果如下图, 可以看出在新增账务账号之后, Hibernate 自动新增了账务账号对应的业务账号数据, 这就是级联添加所起到的作用。



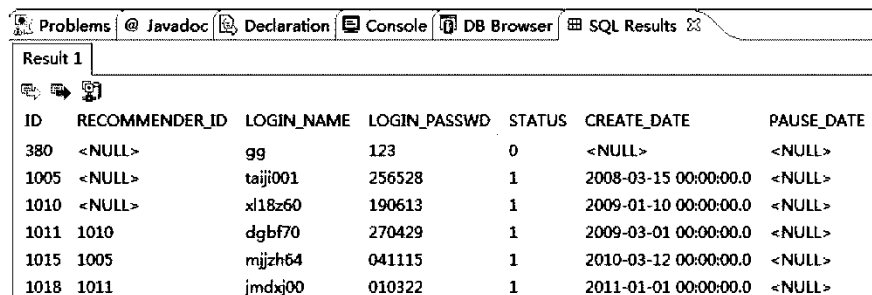
```

<terminated> TestCascade.test1 [JUnit] D:\tarena\myeclipse\Common\binary\com.sun.java.jdk.win32.x86_1.6
log4j:WARN No appenders could be found for logger (org.hibernate)
log4j:WARN Please initialize the log4j system properly.
Hibernate: select ACCOUNT_SEQ.nextval from dual
Hibernate: select SERVICE_SEQ.nextval from dual
Hibernate: select SERVICE_SEQ.nextval from dual
Hibernate: insert into ACCOUNT (RECOMMENDER_ID, LOGIN_NAME, LOGIN_PASSWD, STATUS, CREATE_DATE, PAUSE_DATE) values (1005, 'taiji001', '256528', 1, '2008-03-15 00:00:00.0', <NULL>)
Hibernate: insert into SERVICE (UNIX_HOST, OS_USERNAME, LOGIN_PASSWD, STATUS, CREATE_DATE, PAUSE_DATE, ACCOUNT_ID) values ('192.168.1.1', 'gg1', '123', 0, <NULL>, <NULL>, 1005)
Hibernate: insert into SERVICE (UNIX_HOST, OS_USERNAME, LOGIN_PASSWD, STATUS, CREATE_DATE, PAUSE_DATE, ACCOUNT_ID) values ('192.168.1.2', 'gg2', '123', 0, <NULL>, <NULL>, 1005)
Hibernate: update SERVICE set ACCOUNT_ID=? where id=?
Hibernate: update SERVICE set ACCOUNT_ID=? where id=?

```

图-6

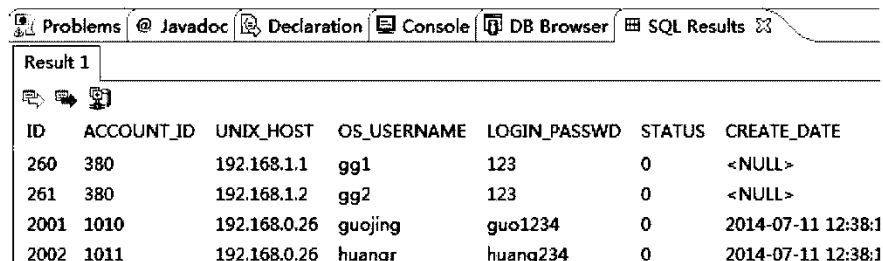
此时, 查询账务账号表, 数据如下图, 其中 id=380 的行就是刚刚添加的行。



ID	RECOMMENDER_ID	LOGIN_NAME	LOGIN_PASSWD	STATUS	CREATE_DATE	PAUSE_DATE
380	<NULL>	gg	123	0	<NULL>	<NULL>
1005	<NULL>	taiji001	256528	1	2008-03-15 00:00:00.0	<NULL>
1010	<NULL>	xl18z60	190613	1	2009-01-10 00:00:00.0	<NULL>
1011	1010	dgbf70	270429	1	2009-03-01 00:00:00.0	<NULL>
1015	1005	mjjzh64	041115	1	2010-03-12 00:00:00.0	<NULL>
1018	1011	jmdxj00	010322	1	2011-01-01 00:00:00.0	<NULL>

图-7

再查询业务账号表, 数据如下图, 其中 account_id=380 的行就是级联添加的行。



ID	ACCOUNT_ID	UNIX_HOST	OS_USERNAME	LOGIN_PASSWD	STATUS	CREATE_DATE
260	380	192.168.1.1	gg1	123	0	<NULL>
261	380	192.168.1.2	gg2	123	0	<NULL>
2001	1010	192.168.0.26	guojing	guo1234	0	2014-07-11 12:38:1
2002	1011	192.168.0.26	huangqr	huangq234	0	2014-07-11 12:38:1

图-8

步骤三：测试级联修改

在 TestCascade 测试类中，增加测试级联修改的方法，代码如下：

```
package com.tarena.test;

import java.util.HashSet;
import java.util.Set;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.junit.Test;
import com.tarena.entity.Account;
import com.tarena.entity.Service;
import com.tarena.util.HibernateUtil;

public class TestCascade {

    /**
     * 级联添加
     */
    @Test
    public void test1() {
        // 模拟要添加的账务账号
        Account a = new Account();
        a.setLoginName("gg");
        a.setLoginPassword("123");
        a.setRealName("gg");
        a.setIdcardNo("120392198410282549");
        a.setStatus("0");
        a.setTelephone("110");

        // 模拟要添加的业务账号
        Service s1 = new Service();
        s1.setAccount(a);
        s1.setOsUserName("gg1");
        s1.setLoginPassword("123");
        s1.setUnixHost("192.168.1.1");
        s1.setCostId(5);
        s1.setStatus("0");

        Service s2 = new Service();
        s2.setAccount(a);
        s2.setOsUserName("gg2");
        s2.setLoginPassword("123");
        s2.setUnixHost("192.168.1.2");
        s2.setCostId(5);
        s2.setStatus("0");

        // 将业务账号与账务账号关联
        a.setServices(new HashSet<Service>());
        a.getServices().add(s1);
        a.getServices().add(s2);

        Session session = HibernateUtil.getSession();
        Transaction ts = session.beginTransaction();
        try {
            session.save(a);
            ts.commit();
        } catch (HibernateException e) {
            e.printStackTrace();
            ts.rollback();
        } finally {

```

```

        session.close();
    }
}

/**
 * 级联修改
 */
@Test
public void test2() {
    Session session = HibernateUtil.getSession();
    // 查询出要修改的账务账号
    Account account =
        (Account) session.get(Account.class, 380);
    // 模拟对账务账号的修改
    account.setLoginName("pp");
    Set<Service> services = account.getServices();
    for (Service service : services) {
        // 模拟对业务账号的修改
        service.setLoginPassword("pp");
    }
    Transaction ts = session.beginTransaction();
    try {
        session.update(account);
        ts.commit();
    } catch (HibernateException e) {
        e.printStackTrace();
        ts.rollback();
    } finally {
        session.close();
    }
}
}

```

执行 test2()方法后，控制台输出的结果如下图，可以看出在修改完账务账号之后，Hibernate 自动修改了它对应的业务账号数据。

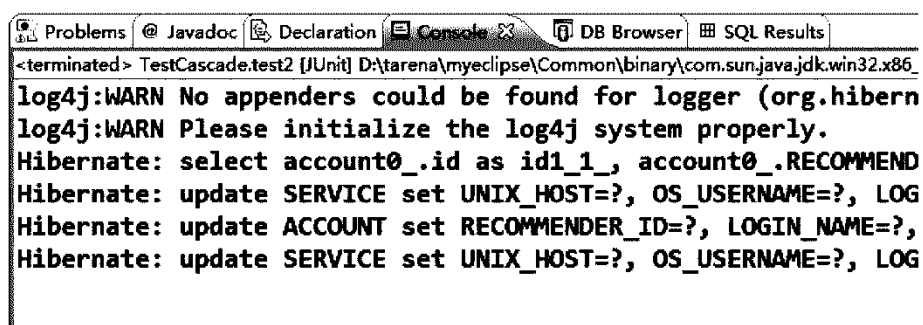


图-9

此时，查询账务账号表，id=380 的行已经发生了改变。

Result 1						
ID	RECOMMENDER_ID	LOGIN_NAME	LOGIN_PASSWD	STATUS	CREATE_DATE	
380	<NULL>	pp	123	0	<NULL>	
1005	<NULL>	taiji001	256528	1	2008-03-15 00:00:00.C	
1010	<NULL>	xl18z60	190613	1	2009-01-10 00:00:00.C	

图-10

再查询业务账号表，account_id=380 的行也发生了改变。

Result 1						
ID	ACCOUNT_ID	UNIX_HOST	OS_USERNAME	LOGIN_PASSWD	STATUS	CREATE_DATE
260	380	192.168.1.1	gg1	pp	0	<NULL>
261	380	192.168.1.2	gg2	pp	0	<NULL>
2001	1010	192.168.0.26	guojing	guo1234	0	2014-07-11 12:5
2002	1011	192.168.0.26	huangr	huang234	0	2014-07-11 12:5

图-11

• 完整代码

以下为本案例的完整代码。

其中 Account.hbm.xml 完整代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<!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.tarena.entity.Account" table="ACCOUNT">
<id name="id" type="integer" column="id">
<generator class="sequence">
<param name="sequence">ACCOUNT_SEQ</param>
</generator>
</id>
<property name="recommenderId"
type="integer" column="RECOMMENDER_ID"/>
<property name="loginName"
type="string" column="LOGIN_NAME"/>
<property name="loginPassword"
type="string" column="LOGIN_PASSWD"/>
<property name="status"
type="string" column="STATUS"/>
<property name="createDate"
type="date" column="CREATE_DATE"/>
<property name="pauseDate"
type="date" column="PAUSE_DATE"/>
<property name="closeDate"
type="date" column="CLOSE_DATE"/>
<property name="realName"
type="string" column="REAL_NAME"/>
<property name="idcardNo"
type="string" column="IDCARD_NO"/>
<property name="birthdate"
type="date" column="BIRTHDATE"/>

```

```

<property name="gender"
    type="string" column="GENDER"/>
<property name="occupation"
    type="string" column="OCCUPATION"/>
<property name="telephone"
    type="string" column="TELEPHONE"/>
<property name="email"
    type="string" column="EMAIL"/>
<property name="mailaddress"
    type="string" column="MAILADDRESS"/>
<property name="zipcode"
    type="string" column="ZIPCODE"/>
<property name="qq"
    type="string" column="QQ"/>
<property name="lastLoginTime"
    type="date" column="LAST_LOGIN_TIME"/>
<property name="lastLoginIp"
    type="string" column="LAST_LOGIN_IP"/>

<!-- 配置 services 属性，采用一对多的关系 -->
<set name="services"
    lazy="false" fetch="join"
    cascade="save-update">
    <!-- 用于指定关联条件，写关联条件的外键字段 -->
    <key column="ACCOUNT_ID"/>
    <!-- 用于指定采用哪种关系，加载哪方数据 -->
    <one-to-many
        class="com.tarena.entity.Service"/>
</set>
</class>
</hibernate-mapping>

```

TestCascade 完整代码如下：

```

package com.tarena.test;

import java.util.HashSet;
import java.util.Set;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.junit.Test;
import com.tarena.entity.Account;
import com.tarena.entity.Service;
import com.tarena.util.HibernateUtil;

public class TestCascade {

    /**
     * 级联添加
     */
    @Test
    public void test1() {
        // 模拟要添加的账务账号
        Account a = new Account();
        a.setLoginName("gg");
        a.setLoginPassword("123");
        a.setRealName("gg");
        a.setIdcardNo("120392198410282549");
        a.setStatus("0");
        a.setTelephone("110");

        // 模拟要添加的业务账号
        Service s1 = new Service();
    }
}

```

```
s1.setAccount(a);
s1.setOsUserName("gg1");
s1.setLoginPassword("123");
s1.setUnixHost("192.168.1.1");
s1.setCostId(5);
s1.setStatus("0");

Service s2 = new Service();
s2.setAccount(a);
s2.setOsUserName("gg2");
s2.setLoginPassword("123");
s2.setUnixHost("192.168.1.2");
s2.setCostId(5);
s2.setStatus("0");

// 将业务账号与账务账号关联
a.setServices(new HashSet<Service>());
a.getServices().add(s1);
a.getServices().add(s2);

Session session = HibernateUtil.getSession();
Transaction ts = session.beginTransaction();
try {
    session.save(a);
    ts.commit();
} catch (HibernateException e) {
    e.printStackTrace();
    ts.rollback();
} finally {
    session.close();
}

}

/**
 * 级联修改
 */
@Test
public void test2() {
    Session session = HibernateUtil.getSession();
    // 查询出要修改的账务账号
    Account account =
        (Account) session.get(Account.class, 380);
    // 模拟对账务账号的修改
    account.setLoginName("pp");
    Set<Service> services = account.getServices();
    for (Service service : services) {
        // 模拟对业务账号的修改
        service.setLoginPassword("pp");
    }
    Transaction ts = session.beginTransaction();
    try {
        session.update(account);
        ts.commit();
    } catch (HibernateException e) {
        e.printStackTrace();
        ts.rollback();
    } finally {
        session.close();
    }
}

}
```

4. 级联删除

- 问题

使用级联删除，在删除账务账号时，自动删除其对应的业务账号。

- 方案

通过在映射关系文件中设置 `cascade= "delete"`，可以支持级联删除，如果想全面支持级联添加、修改和删除，可以设置 `cascade= "all"`。

- 步骤

实现此案例需要按照如下步骤进行。

步骤一：在账务账号映射关系文件中设置级联删除

在账务账号映射关系文件 `Account.hbm.xml` 中，通过 `cascade= "all"` 设置支持级联删除，代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<!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.tarena.entity.Account" table="ACCOUNT">
  <id name="id" type="integer" column="id">
    <generator class="sequence">
      <param name="sequence">ACCOUNT_SEQ</param>
    </generator>
  </id>
  <property name="recommenderId"
    type="integer" column="RECOMMENDER ID"/>
  <property name="loginName"
    type="string" column="LOGIN_NAME"/>
  <property name="loginPassword"
    type="string" column="LOGIN PASSWD"/>
  <property name="status"
    type="string" column="STATUS"/>
  <property name="createDate"
    type="date" column="CREATE_DATE"/>
  <property name="pauseDate"
    type="date" column="PAUSE DATE"/>
  <property name="closeDate"
    type="date" column="CLOSE_DATE"/>
  <property name="realName"
    type="string" column="REAL_NAME"/>
  <property name="idcardNo"
    type="string" column="IDCARD NO"/>
  <property name="birthdate"
    type="date" column="BIRTHDATE"/>
  <property name="gender"
    type="string" column="GENDER"/>
  <property name="occupation"
    type="string" column="OCCUPATION"/>
  <property name="telephone"
    type="string" column="TELEPHONE"/>
  <property name="email"
    type="string" column="EMAIL"/>
  </class>
</hibernate-mapping>
```



```
<property name="mailaddress"
    type="string" column="MAILADDRESS"/>
<property name="zipcode"
    type="string" column="ZIPCODE"/>
<property name="qq"
    type="string" column="QQ"/>
<property name="lastLoginTime"
    type="date" column="LAST_LOGIN_TIME"/>
<property name="lastLoginIp"
    type="string" column="LAST_LOGIN_IP"/>

<!-- 配置 services 属性，采用一对多的关系 -->

<set name="services"
    lazy="false" fetch="join"
    cascade="all">

    <!-- 用于指定关联条件，写关联条件的外键字段 -->
    <key column="ACCOUNT_ID"/>
    <!-- 用于指定采用哪种关系，加载哪方数据 -->
    <one-to-many
        class="com.tarena.entity.Service"/>
</set>
</class>
</hibernate-mapping>
```

步骤二：测试级联删除

在 TestCascade 测试类中，增加测试级联删除的方法，代码如下：

```
package com.tarena.test;

import java.util.HashSet;
import java.util.Set;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.junit.Test;
import com.tarena.entity.Account;
import com.tarena.entity.Service;
import com.tarena.util.HibernateUtil;

public class TestCascade {
    /**
     * 级联添加
     */
    @Test
    public void test1() {
        // 模拟要添加的账务账号
        Account a = new Account();
        a.setLoginName("gg");
        a.setLoginPassword("123");
        a.setRealName("gg");
        a.setIdcardNo("120392198410282549");
        a.setStatus("0");
        a.setTelephone("110");

        // 模拟要添加的业务账号
        Service s1 = new Service();
        s1.setAccount(a);
    }
}
```

```

s1.setOsUserName("gg1");
s1.setLoginPassword("123");
s1.setUnixHost("192.168.1.1");
s1.setCostId(5);
s1.setStatus("0");

Service s2 = new Service();
s2.setAccount(a);
s2.setOsUserName("gg2");
s2.setLoginPassword("123");
s2.setUnixHost("192.168.1.2");
s2.setCostId(5);
s2.setStatus("0");

// 将业务账号与账务账号关联
a.setServices(new HashSet<Service>());
a.getServices().add(s1);
a.getServices().add(s2);

Session session = HibernateUtil.getSession();
Transaction ts = session.beginTransaction();
try {
    session.save(a);
    ts.commit();
} catch (HibernateException e) {
    e.printStackTrace();
    ts.rollback();
} finally {
    session.close();
}
}

/**
 * 级联修改
 */
@Test
public void test2() {
    Session session = HibernateUtil.getSession();
    // 查询出要修改的账务账号
    Account account =
        (Account) session.get(Account.class, 380);
    // 模拟对账务账号的修改
    account.setLoginName("pp");
    Set<Service> services = account.getServices();
    for (Service service : services) {
        // 模拟对业务账号的修改
        service.setLoginPassword("pp");
    }
    Transaction ts = session.beginTransaction();
    try {
        session.update(account);
        ts.commit();
    } catch (HibernateException e) {
        e.printStackTrace();
        ts.rollback();
    } finally {
        session.close();
    }
}

/**
 * 级联删除
 */
@Test

```

```
public void test3() {
    Session session = HibernateUtil.getSession();
    Account account = (Account) session.get(Account.class, 380);
    Transaction ts = session.beginTransaction();
    try {
        session.delete(account);
        ts.commit();
    } catch (HibernateException e) {
        e.printStackTrace();
        ts.rollback();
    } finally {
        session.close();
    }
}
```

步骤三：测试

执行 test3() 方法，控制台输出结果如下图，可以看出程序在运行时报错了：

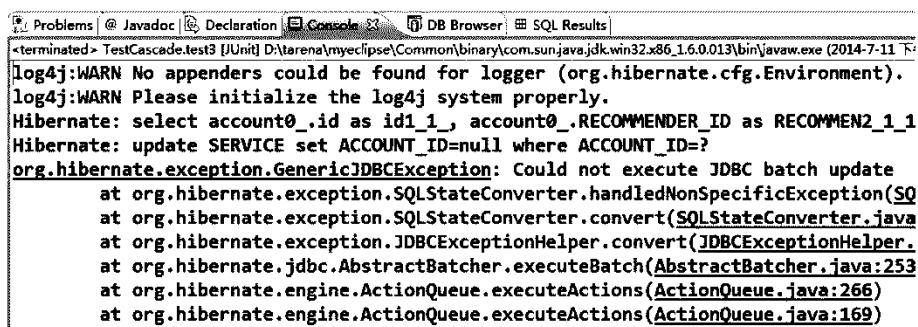


图-12

步骤四：排错

上述错误是执行第二个 SQL 时产生的，这个 SQL 的目的是维护关系字段，将其置为 null，而这个外键字段存在非空约束，因此报错。类似的事情在级联添加时也看到过，参考图-6，不同的是级联添加时要将关联字段设置为新生成的账务账号 ID。然而，在级联新增或删除时业务账号时，业务账号本身已经维护好了关联字段，因此这个额外的操作是多余的，可以去掉，我们可以在关联属性上通过 inverse= "true" 去掉这个行为，代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<!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.tarena.entity.Account" table="ACCOUNT">
    <id name="id" type="integer" column="id">
        <generator class="sequence">
            <param name="sequence">ACCOUNT_SEQ</param>
        </generator>
    </id>
    <property name="recommenderId"
        type="integer" column="RECOMMENDER ID"/>
    <property name="loginName"
        type="string" column="LOGIN_NAME"/>
    <property name="loginPassword"
        type="string" column="LOGIN_PASSWD"/>
```

```

<property name="status"
    type="string" column="STATUS"/>
<property name="createDate"
    type="date" column="CREATE_DATE"/>
<property name="pauseDate"
    type="date" column="PAUSE_DATE"/>
<property name="closeDate"
    type="date" column="CLOSE_DATE"/>
<property name="realName"
    type="string" column="REAL_NAME"/>
<property name="idcardNo"
    type="string" column="IDCARD_NO"/>
<property name="birthdate"
    type="date" column="BIRTHDATE"/>
<property name="gender"
    type="string" column="GENDER"/>
<property name="occupation"
    type="string" column="OCCUPATION"/>
<property name="telephone"
    type="string" column="TELEPHONE"/>
<property name="email"
    type="string" column="EMAIL"/>
<property name="mailaddress"
    type="string" column="MAILADDRESS"/>
<property name="zipcode"
    type="string" column="ZIPCODE"/>
<property name="qq"
    type="string" column="QQ"/>
<property name="lastLoginTime"
    type="date" column="LAST_LOGIN_TIME"/>
<property name="lastLoginIp"
    type="string" column="LAST_LOGIN_IP"/>

<!-- 配置 services 属性，采用一对多的关系 -->

<set name="services"
    lazy="false" fetch="join"
    cascade="all" inverse="true">

    <!-- 用于指定关联条件，写关联条件的外键字段 -->
    <key column="ACCOUNT_ID"/>
    <!-- 用于指定采用哪种关系，加载哪方数据 -->
    <one-to-many
        class="com.tarena.entity.Service"/>
    </set>
</class>
</hibernate-mapping>

```

步骤五：测试

再次执行 test3() 方法，控制台输出结果如下图，可以看出本次删除成功了，不但删除了账务账号数据，在此之前还删除了它所对应的所有业务账号。

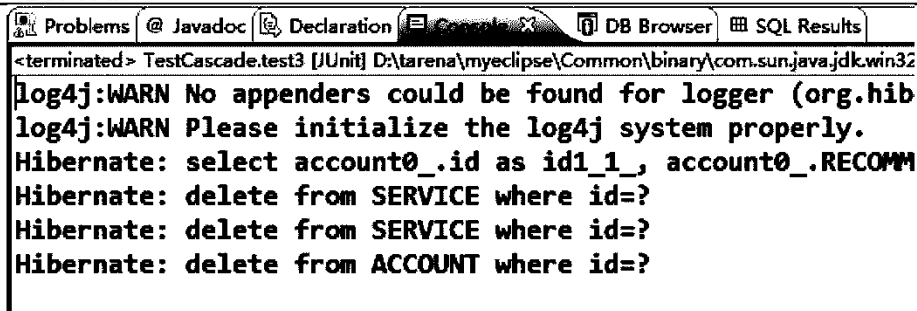


图-13

• 完整代码

以下为本案例的完整代码。

其中 Account.hbm.xml 完整代码如下：

```

<?xml version="1.0" encoding="utf-8"?>
<!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.tarena.entity.Account" table="ACCOUNT">
    <id name="id" type="integer" column="id">
      <generator class="sequence">
        <param name="sequence">ACCOUNT_SEQ</param>
      </generator>
    </id>
    <property name="recommenderId"
      type="integer" column="RECOMMENDER_ID"/>
    <property name="loginName"
      type="string" column="LOGIN_NAME"/>
    <property name="loginPassword"
      type="string" column="LOGIN_PASSWD"/>
    <property name="status"
      type="string" column="STATUS"/>
    <property name="createDate"
      type="date" column="CREATE_DATE"/>
    <property name="pauseDate"
      type="date" column="PAUSE_DATE"/>
    <property name="closeDate"
      type="date" column="CLOSE_DATE"/>
    <property name="realName"
      type="string" column="REAL_NAME"/>
    <property name="idcardNo"
      type="string" column="IDCARD_NO"/>
    <property name="birthdate"
      type="date" column="BIRTHDATE"/>
    <property name="gender"
      type="string" column="GENDER"/>
    <property name="occupation"
      type="string" column="OCCUPATION"/>
    <property name="telephone"
      type="string" column="TELEPHONE"/>
    <property name="email"
      type="string" column="EMAIL"/>
    <property name="mailaddress"
      type="string" column="MAILADDRESS"/>
    <property name="zipcode"
      type="string" column="ZIPCODE"/>
    <property name="qq"
      type="string" column="QQ"/>
  </class>

```

```
<property name="lastLoginTime"
    type="date" column="LAST_LOGIN_TIME"/>
<property name="lastLoginIp"
    type="string" column="LAST_LOGIN_IP"/>

<!-- 配置 services 属性，采用一对多的关系 -->
<set name="services"
    lazy="false" fetch="join"
    cascade="all" inverse="true">
    <!-- 用于指定关联条件，写关联条件的外键字段 -->
    <key column="ACCOUNT_ID"/>
    <!-- 用于指定采用哪种关系，加载哪方数据 -->
    <one-to-many
        class="com.tarena.entity.Service"/>
</set>
</class>
</hibernate-mapping>
```

TestCascade 完整代码如下：

```
package com.tarena.test;

import java.util.HashSet;
import java.util.Set;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.junit.Test;
import com.tarena.entity.Account;
import com.tarena.entity.Service;
import com.tarena.util.HibernateUtil;

public class TestCascade {

    /**
     * 级联添加
     */
    @Test
    public void test1() {
        // 模拟要添加的账务账号
        Account a = new Account();
        a.setLoginName("gg");
        a.setLoginPassword("123");
        a.setRealName("gg");
        a.setIdcardNo("120392198410282549");
        a.setStatus("0");
        a.setTelephone("110");

        // 模拟要添加的业务账号
        Service s1 = new Service();
        s1.setAccount(a);
        s1.setOsUserName("gg1");
        s1.setLoginPassword("123");
        s1.setUnixHost("192.168.1.1");
        s1.setCostId(5);
        s1.setStatus("0");

        Service s2 = new Service();
        s2.setAccount(a);
        s2.setOsUserName("gg2");
        s2.setLoginPassword("123");
        s2.setUnixHost("192.168.1.2");
        s2.setCostId(5);
        s2.setStatus("0");
```

```
// 将业务账号与账务账号关联
a.setServices(new HashSet<Service>());
a.getServices().add(s1);
a.getServices().add(s2);

Session session = HibernateUtil.getSession();
Transaction ts = session.beginTransaction();
try {
    session.save(a);
    ts.commit();
} catch (HibernateException e) {
    e.printStackTrace();
    ts.rollback();
} finally {
    session.close();
}
}

/**
 * 级联修改
 */
@Test
public void test2() {
    Session session = HibernateUtil.getSession();
    // 查询出要修改的账务账号
    Account account =
        (Account) session.get(Account.class, 380);
    // 模拟对账务账号的修改
    account.setLoginName("pp");
    Set<Service> services = account.getServices();
    for (Service service : services) {
        // 模拟对业务账号的修改
        service.setLoginPassword("pp");
    }
    Transaction ts = session.beginTransaction();
    try {
        session.update(account);
        ts.commit();
    } catch (HibernateException e) {
        e.printStackTrace();
        ts.rollback();
    } finally {
        session.close();
    }
}

/**
 * 级联删除
 */
@Test
public void test3() {
    Session session = HibernateUtil.getSession();
    Account account = (Account) session.get(Account.class, 380);
    Transaction ts = session.beginTransaction();
    try {
        session.delete(account);
        ts.commit();
    } catch (HibernateException e) {
        e.printStackTrace();
        ts.rollback();
    } finally {
        session.close();
    }
}
}
```

5. HQL 查询，按条件查询

- **问题**

使用带条件的 HQL 查询业务账号数据。

- **方案**

在 HQL 中拼入条件，如 name=?，然后在查询之前使用 query 给参数赋值。

- **步骤**

实现此案例需要按照如下步骤进行。

步骤一：编写按条件查询方法

在 com.tarena.test 包下创建测试类 TestHQL，并在类中增加按条件查询的测试方法，代码如下：

```
package com.tarena.test;

import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Service;
import com.tarena.util.HibernateUtil;

public class TestHQL {
    /**
     * 按条件查询
     */
    @Test
    public void test1() {
        String hql = "from Service where unixHost=?";
        Session session = HibernateUtil.getSession();
        Query query = session.createQuery(hql);
        query.setString(0, "192.168.0.20");
        List<Service> services = query.list();
        for(Service service : services){
            System.out.println(service.getId()
                + " " + service.getUnixHost()
                + " " + service.getOsUserName());
        }
        session.close();
    }
}
```

步骤二：测试

执行 test1() 方法，控制台输出效果如下图，可以看到按条件查询出的结果：

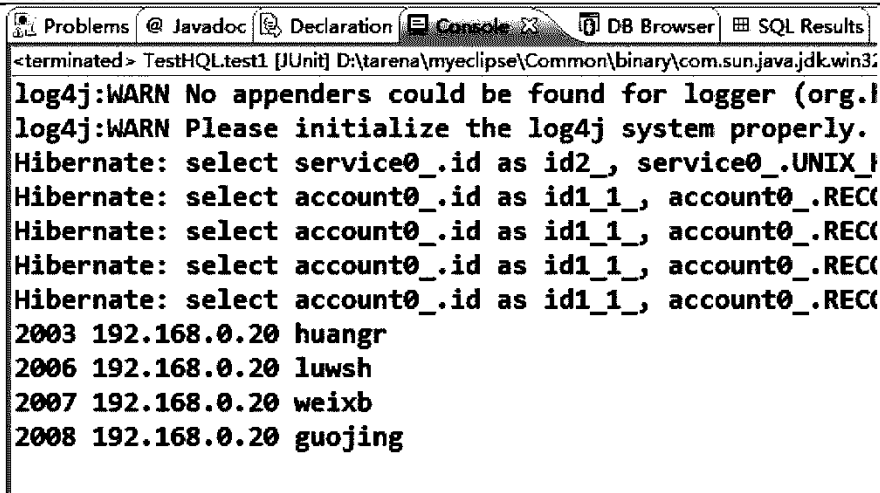


图-14

• 完整代码

以下为本案例的完整代码。

其中 TestHQL 完整代码如下：

```
package com.tarena.test;

import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Service;
import com.tarena.util.HibernateUtil;

public class TestHQL {
    /**
     * 按条件查询
     */
    @Test
    public void test1() {
        String hql = "from Service where unixHost=?";
        Session session = HibernateUtil.getSession();
        Query query = session.createQuery(hql);
        query.setString(0, "192.168.0.20");
        List<Service> services = query.list();
        for(Service service : services){
            System.out.println(service.getId()
                + " " + service.getUnixHost()
                + " " + service.getOsUserName());
        }
        session.close();
    }
}
```

6. HQL 查询，查询一部分字段

• 问题

使用 HQL 查询，要求只查询一部分字段。

• 方案

可以通过 select 子句明确指定要返回的字段 注意 HQL 中 select 子句中写的是属性，而不是表中的字段。

• 步骤

实现此案例需要按照如下步骤进行。

步骤一：编写查询一部分字段方法

在 TestHQL 中增加查询一部分字段的方法，代码如下：

```
package com.tarena.test;

import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Service;
import com.tarena.util.HibernateUtil;

public class TestHQL {
    // 其他查询方法略

    /**
     * 查询一部分字段
     */
    @Test
    public void test2() {
        String hql = "select id,unixHost,osUserName " +
            "from Service where unixHost=?";
        Session session = HibernateUtil.getSession();
        Query query = session.createQuery(hql);
        query.setString(0, "192.168.0.20");
        List<Object[]> services = query.list();
        for(Object[] service : services) {
            System.out.println(service[0]
                + " " + service[1]
                + " " + service[2]);
        }
        session.close();
    }
}
```

步骤二：测试

执行 test2()，控制台输出结果如下图，可以看到查询出来的这些字段的值。

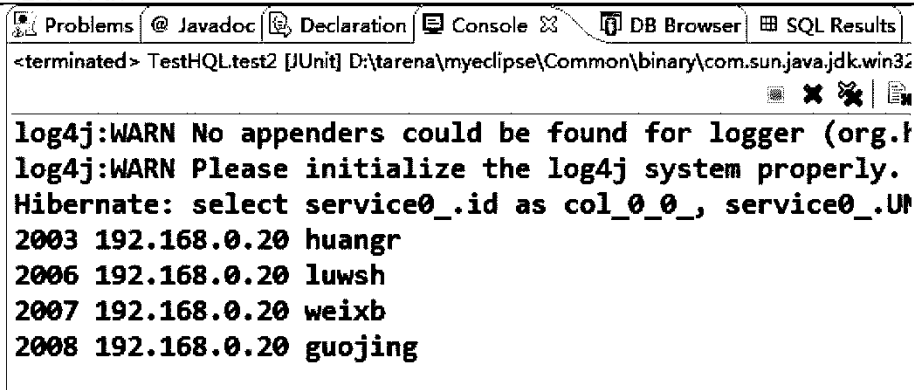


图-15

• 完整代码

以下为本案例的完整代码。

其中 TestHQL 完整代码如下：

```
package com.tarena.test;

import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Service;
import com.tarena.util.HibernateUtil;

public class TestHQL {
    /**
     * 按条件查询
     */
    @Test
    public void test1() {
        String hql = "from Service where unixHost=?";
        Session session = HibernateUtil.getSession();
        Query query = session.createQuery(hql);
        query.setString(0, "192.168.0.20");
        List<Service> services = query.list();
        for(Service service : services){
            System.out.println(service.getId()
                + " " + service.getUnixHost()
                + " " + service.getOsUserName());
        }
        session.close();
    }

    /**
     * 查询一部分字段
     */
    @Test
    public void test2() {
        String hql = "select id,unixHost,osUserName " +
            "from Service where unixHost=?";
        Session session = HibernateUtil.getSession();
        Query query = session.createQuery(hql);
        query.setString(0, "192.168.0.20");
        List<Object[]> services = query.list();
        for(Object[] service : services) {
            System.out.println(service[0])
        }
    }
}
```

```

        + " " + service[1]
        + " " + service[2]);
    }
    session.close();
}
}

```

7. HQL 查询，分页查询

• 问题

按照每页显示 3 条数据，查询第 1 页的条件，查询出业务账号表中满足条件的记录，并且查询出总页数。

• 方案

- 1) 分页查询时，在执行查询之前可以通过 query 对象设置分页参数值，query.setFirstResult(起点)，以及 query.setMaxResults(页容量)。
- 2) 查询总页数，只需要通过 hql 查询出总行数，再根据总行数计算总页数即可，而查询总行数的 hql 格式为 select count(*) from 对象。

• 步骤

实现此案例需要按照如下步骤进行。

步骤一：编写分页查询的方法

在 TestHQL 中，增加分页查询的方法，代码如下：

```

package com.tarena.test;

import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Service;
import com.tarena.util.HibernateUtil;

public class TestHQL {
    // 其他查询方法略

    /**
     * 分页查询
     */
    @Test
    public void test3() {
        int page = 1;
        int pageSize = 3;

        String hql = "from Service order by id";
        Session session = HibernateUtil.getSession();
        Query query = session.createQuery(hql);
        // 追加分页参数设置
        int from = (page - 1) * pageSize;
    }
}

```

```

        query.setFirstResult(from);// 设置起点, 从 0 开始
        query.setMaxResults(pageSize);// 设置页容量
        List<Service> services = query.list();
        for (Service service : services) {
            System.out.println(
                service.getId() + " "
                + service.getUnixHost() + " "
                + service.getOsUserName());
        }
        session.close();
    }

}

```

步骤二：测试

执行 test3(), 控制台输出结果如下图, 输出了第 1 页的 3 条数据。

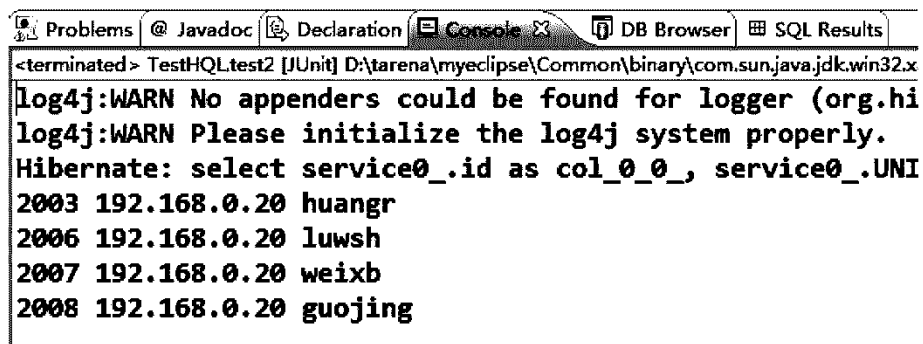


图-16

步骤三：编写查询总页数的方法

在 TestHQL 中, 增加查询总页数的方法, 代码如下:

```

package com.tarena.test;

import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Service;
import com.tarena.util.HibernateUtil;

public class TestHQL {

    // 其他查询方法略

    /**
     * 查询总页数
     */
    @Test
    public void test4() {
        int pageSize=3;
        String hql = "select count(*) from Service";
        Session session = HibernateUtil.getSession();
        Query query = session.createQuery(hql);
        int rows = Integer.parseInt(query.uniqueResult().toString());
    }
}

```

```

        int totalPages = 0;
        if(rows%pageSize == 0) {
            totalPages = rows/pageSize;
        } else {
            totalPages = rows/pageSize+1;
        }
        System.out.println(totalPages);
        session.close();
    }
}

```

步骤四：测试

执行 test4()，控制台输出结果如下图，输出了总页数。

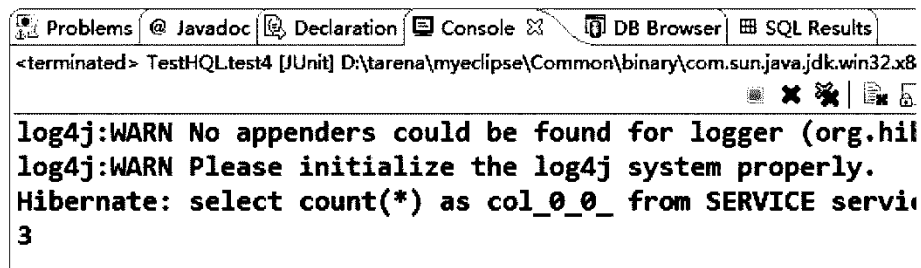


图-17

• 完整代码

以下为本案例的完整代码。

其中 TestHQL 完整代码如下：

```

package com.tarena.test;

import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Service;
import com.tarena.util.HibernateUtil;

public class TestHQL {

    /**
     * 按条件查询
     */
    @Test
    public void test1() {
        String hql = "from Service where unixHost=?";
        Session session = HibernateUtil.getSession();
        Query query = session.createQuery(hql);
        query.setString(0, "192.168.0.20");
        List<Service> services = query.list();
        for(Service service : services){
            System.out.println(service.getId()
                + " " + service.getUnixHost()
                + " " + service.getOsUserName());
        }
        session.close();
    }
}

```

```
/**
 * 查询一部分字段
 */
@Test
public void test2() {
    String hql = "select id,unixHost,osUserName " +
        "from Service where unixHost=?";
    Session session = HibernateUtil.getSession();
    Query query = session.createQuery(hql);
    query.setString(0, "192.168.0.20");
    List<Object[]> services = query.list();
    for(Object[] service : services) {
        System.out.println(service[0]
            + " " + service[1]
            + " " + service[2]);
    }
    session.close();
}

/**
 * 分页查询
 */
@Test
public void test3() {
    int page = 1;
    int pageSize = 3;

    String hql = "from Service order by id";
    Session session = HibernateUtil.getSession();
    Query query = session.createQuery(hql);
    // 追加分页参数设置
    int from = (page - 1) * pageSize;
    query.setFirstResult(from); // 设置起点, 从0开始
    query.setMaxResults(pageSize); // 设置页容量
    List<Service> services = query.list();
    for (Service service : services) {
        System.out.println(
            service.getId() + " "
            + service.getUnixHost() + " "
            + service.getOsUserName());
    }
    session.close();
}

/**
 * 查询总页数
 */
@Test
public void test4() {
    int pageSize=3;
    String hql = "select count(*) from Service";
    Session session = HibernateUtil.getSession();
    Query query = session.createQuery(hql);
    int rows = Integer.parseInt(query.uniqueResult().toString());
    int totalPages = 0;
    if(rows%pageSize == 0) {
        totalPages = rows/pageSize;
    } else {
        totalPages = rows/pageSize+1;
    }
    System.out.println(totalPages);
    session.close();
}
}
```

8. HQL 查询，多表联合查询

- **问题**

使用 HQL，联合查询出业务账号与账务账号的数据。

- **方案**

HQL 中多表联合查询的方式有 3 种，分别是

- 1) 对象方式关联
- 2) join 方式关联
- 3) select 子句关联

使用这三种方式来做业务账号与账务账号的联合查询。

- **步骤**

实现此案例需要按照如下步骤进行。

步骤一：编写对象方式关联的方法

在 TestHQL 中，增加对象方式关联的查询方法，代码如下：

```
package com.tarena.test;

import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Service;
import com.tarena.util.HibernateUtil;

public class TestHQL {
    // 其他查询方法略

    /**
     * 多表联合查询-对象方式关联
     */
    @Test
    public void test5() {
        String hql = "select " +
            "s.id," +
            "s.osUserName," +
            "s.unixHost, " +
            "a.id,a.realName," +
            "a.idcardNo " +
            "from Service s,Account a " +
            "where s.account.id=a.id ";
        Session session = HibernateUtil.getSession();
        Query query = session.createQuery(hql);
        List<Object[]> list = query.list();
        for (Object[] objs : list) {
            System.out.println(
                objs[0] + " " +
```



```

        objs[1] + " " +
        objs[2] + " " +
        objs[3] + " " +
        objs[4] + " " +
        objs[5]);
    }
    session.close();
}
}
}

```

步骤二：测试

执行 test5()，控制台输出结果如下图，可以查询出关联的数据。

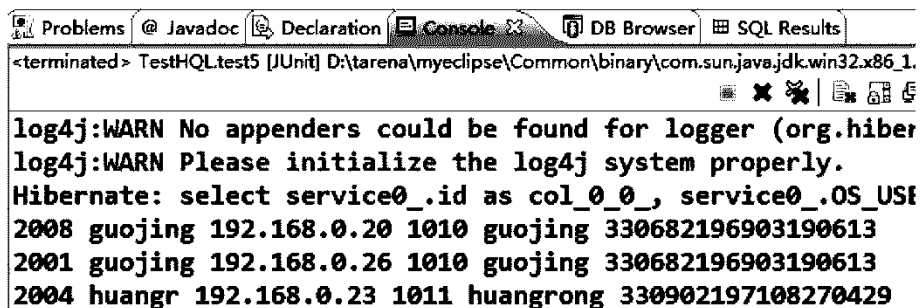


图-18

步骤三：编写 join 方式关联的方法

在 TestHQL 中，增加 join 方式关联的查询方法，代码如下：

```

package com.tarena.test;

import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Service;
import com.tarena.util.HibernateUtil;

public class TestHQL {
    // 其他查询方法略

    /**
     * 多表联合查询-join 方式关联
     */
    @Test
    public void test6() {
        String hql = "select " +
            "s.id,s.osUserName," +
            "s.unixHost, " +
            "a.id," +
            "a.realName," +
            "a.idcardNo " +
            "from Service s join s.account a ";
        Session session = HibernateUtil.getSession();
        Query query = session.createQuery(hql);
        List<Object[]> list = query.list();
        for (Object[] objs : list) {
            System.out.println(

```

```

        objs[0] + " " +
        objs[1] + " " +
        objs[2] + " " +
        objs[3] + " " +
        objs[4] + " " +
        objs[5]);
    }
    session.close();
}
}
}

```

步骤四：测试

执行 test6()，控制台输出结果如下图，可以查询出关联的数据。

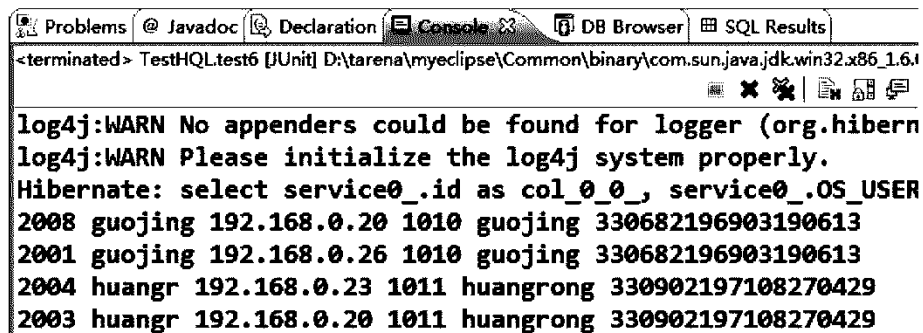


图-19

步骤五：编写 select 子句关联的方法

在 TestHQL 中，增加 select 子句关联的查询方法，代码如下：

```

package com.tarena.test;

import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Service;
import com.tarena.util.HibernateUtil;

public class TestHQL {

    // 其他查询方法略

    /**
     * 多表联合查询-select 子句关联
     */
    @Test
    public void test7() {
        String hql = "select " +
            "id," +
            "osUserName," +
            "unixHost, " +
            "account.id," +
            "account.realName," +
            "account.idcardNo " +
            "from Service ";
        Session session = HibernateUtil.getSession();
    }
}

```

```

Query query = session.createQuery(hql);
List<Object[]> list = query.list();
for (Object[] objs : list) {
    System.out.println(
        objs[0] + " " +
        objs[1] + " " +
        objs[2] + " " +
        objs[3] + " " +
        objs[4] + " " +
        objs[5]);
}
session.close();
}
}

```

步骤六：测试

执行 test7(), 控制台输出结果如下, 可以查询出关联的数据。

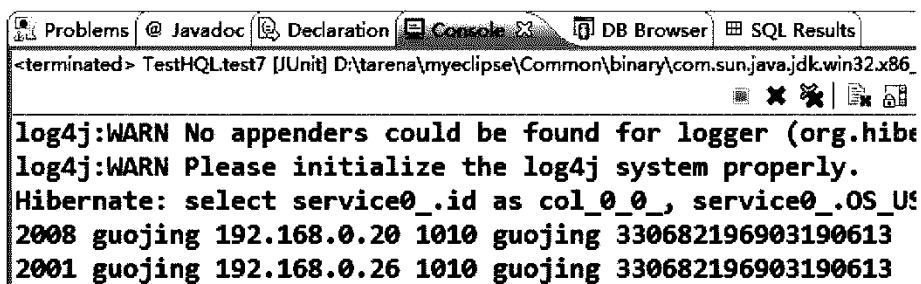


图-20

• 完整代码

以下为本案例的完整代码。

其中 TestHQL 完整代码如下：

```

package com.tarena.test;

import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Service;
import com.tarena.util.HibernateUtil;

public class TestHQL {

    /**
     * 按条件查询
     */
    @Test
    public void test1() {
        String hql = "from Service where unixHost=?";
        Session session = HibernateUtil.getSession();
        Query query = session.createQuery(hql);
        query.setString(0, "192.168.0.20");
        List<Service> services = query.list();
        for (Service service : services) {
            System.out.println(service.getId())

```

```

        + " " + service.getUnixHost()
        + " " + service.getOsUserName());
    }
    session.close();
}

/**
 * 查询一部分字段
 */
@Test
public void test2() {
    String hql = "select id,unixHost,osUserName " +
        "from Service where unixHost=?";
    Session session = HibernateUtil.getSession();
    Query query = session.createQuery(hql);
    query.setString(0, "192.168.0.20");
    List<Object[]> services = query.list();
    for(Object[] service : services) {
        System.out.println(service[0]
            + " " + service[1]
            + " " + service[2]);
    }
    session.close();
}

/**
 * 分页查询
 */
@Test
public void test3() {
    int page = 1;
    int pageSize = 3;

    String hql = "from Service order by id";
    Session session = HibernateUtil.getSession();
    Query query = session.createQuery(hql);
    // 追加分页参数设置
    int from = (page - 1) * pageSize;
    query.setFirstResult(from); // 设置起点, 从 0 开始
    query.setMaxResults(pageSize); // 设置页容量
    List<Service> services = query.list();
    for (Service service : services) {
        System.out.println(
            service.getId() + " "
            + service.getUnixHost() + " "
            + service.getOsUserName());
    }
    session.close();
}

/**
 * 查询总页数
 */
@Test
public void test4() {
    int pageSize=3;
    String hql = "select count(*) from Service";
    Session session = HibernateUtil.getSession();
    Query query = session.createQuery(hql);
    int rows = Integer.parseInt(query.uniqueResult().toString());
    int totalPages = 0;
    if(rows%pageSize == 0) {
        totalPages = rows/pageSize;
    } else {
        totalPages = rows/pageSize+1;
    }
    System.out.println(totalPages);
}

```

```
        session.close();
    }

    /**
     * 多表联合查询-对象方式关联
     */
    @Test
    public void test5() {
        String hql = "select " +
            "s.id," +
            "s.osUserName," +
            "s.unixHost, " +
            "a.id,a.realName," +
            "a.idcardNo " +
            "from Service s,Account a " +
            "where s.account.id=a.id ";
        Session session = HibernateUtil.getSession();
        Query query = session.createQuery(hql);
        List<Object[]> list = query.list();
        for (Object[] objs : list) {
            System.out.println(
                objs[0] + " " +
                objs[1] + " " +
                objs[2] + " " +
                objs[3] + " " +
                objs[4] + " " +
                objs[5]);
        }
        session.close();
    }

    /**
     * 多表联合查询-join 方式关联
     */
    @Test
    public void test6() {
        String hql = "select " +
            "s.id,s.osUserName," +
            "s.unixHost, " +
            "a.id," +
            "a.realName," +
            "a.idcardNo " +
            "from Service s join s.account a ";
        Session session = HibernateUtil.getSession();
        Query query = session.createQuery(hql);
        List<Object[]> list = query.list();
        for (Object[] objs : list) {
            System.out.println(
                objs[0] + " " +
                objs[1] + " " +
                objs[2] + " " +
                objs[3] + " " +
                objs[4] + " " +
                objs[5]);
        }
        session.close();
    }

    /**
     * 多表联合查询-select 子句关联
     */
    @Test
    public void test7() {
        String hql = "select " +
            "id," +
            "osUserName," +
            "unixHost, " +
```

```

        "account.id," +
        "account.realName," +
        "account.idcardNo " +
        "from Service ";
    Session session = HibernateUtil.getSession();
    Query query = session.createQuery(hql);
    List<Object[]> list = query.list();
    for (Object[] objs : list) {
        System.out.println(
            objs[0] + " " +
            objs[1] + " " +
            objs[2] + " " +
            objs[3] + " " +
            objs[4] + " " +
            objs[5]);
    }
    session.close();
}
}

```

9. Hibernate 中的 SQL 查询

- **问题**

在 Hibernate 中直接使用 SQL 查询业务账号数据。

- **方案**

Hibernate 中可以通过 SQLQuery 对象来执行原始的 SQL。

- **步骤**

实现此案例需要按照如下步骤进行。

步骤一：编写使用 SQL 查询的方法

在 com.tarena.test 包下，创建测试类 TestOtherQuery，并在类中增加使用 SQL 查询的方法，代码如下：

```

package com.tarena.test;

import java.util.List;
import org.hibernate.SQLQuery;
import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Service;
import com.tarena.util.HibernateUtil;

public class TestOtherQuery {

    /**
     * 使用 SQL 查询
     */
    @Test
    public void test1() {
        String sql = "select * " +
            "from SERVICE where unix_host=?";
    }
}

```

```

Session session = HibernateUtil.getSession();
SQLQuery query = session.createSQLQuery(sql);
query.setString(0, "192.168.0.20");
query.addEntity(Service.class);
//采用 Service 类型封装一条数据
List<Service> list = query.list();
for(Service service : list){
    System.out.println(
        service.getId() + " " +
        service.getOsUserName() + " " +
        service.getUnixHost() + " "
    );
}
session.close();
}
}

```

步骤二：测试

执行 test1(), 控制台输出结果如下图, 是直接执行 SQL 所查询到的内容:

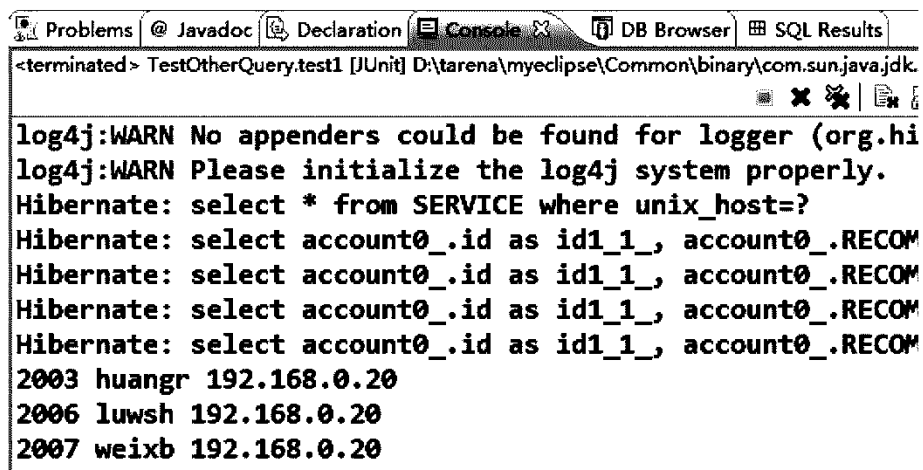


图-21

• 完整代码

以下为本案例的完整代码。

其中 TestOtherQuery 完整代码如下:

```

package com.tarena.test;

import java.util.List;
import org.hibernate.SQLQuery;
import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Service;
import com.tarena.util.HibernateUtil;

public class TestOtherQuery {

    /**
     * 使用 SQL 查询
     */
    @Test

```

```
public void test1() {
    String sql = "select * " +
        "from SERVICE where unix_host=?";
    Session session = HibernateUtil.getSession();
    SQLQuery query = session.createSQLQuery(sql);
    query.setString(0, "192.168.0.20");
    query.addEntity(Service.class);
    //采用 Service 类型封装一条数据
    List<Service> list = query.list();
    for(Service service : list){
        System.out.println(
            service.getId() + " " +
            service.getOsUserName() + " " +
            service.getUnixHost() + " "
        );
    }
    session.close();
}
}
```

10. 使用二级缓存

- **问题**

在查询员工时，使用二级缓存，使得不同的 session 可以共享这些员工数据。

- **方案**

使用 ehcache 缓存组件来支持二级缓存，二级缓存的使用步骤为

- 1) 导入二级缓存驱动包
- 2) 引入二级缓存配置文件
- 3) 在 hibernate.cfg.xml 中开启二级缓存并指定二级缓存驱动类
- 4) 在映射关系文件中设置缓存策略
- 5) 执行查询

- **步骤**

实现此案例需要按照如下步骤进行。

步骤一：导入二级缓存驱动包

导入二级缓存驱动包 ehcache-1.2.3.jar，导入后项目中包结构如下图：

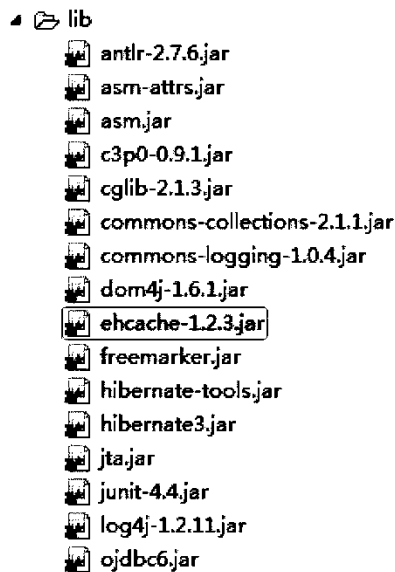


图-22

步骤二：引入二级缓存配置文件

引入二级缓存配置文件 ehcache.xml，并配置缓存参数，代码如下：

```
<ehcache>

<!--
    缓存到硬盘时的缓存路径, java.io.tmpdir 表示
    系统默认缓存路径。
-->
<diskStore path="java.io.tmpdir"/>

<!--
    默认缓存配置。
    maxElementsInMemory：二级缓存可容纳最大对象数。
    eternal：是否保持二级缓存中对象不变。
    timeToIdleSeconds：
        允许对象空闲的时间，即对象最后一次访问起，超过该时间即失效。
    timeToLiveSeconds：
        允许对象存活的时间，即对象创建起，超过该时间即失效。
    overflowToDisk：
        内存不足时，是否允许使用硬盘缓存，写入路径参见 diskStore。
-->
<defaultCache
    maxElementsInMemory="300"
    eternal="false"
    timeToIdleSeconds="120"
    timeToLiveSeconds="300"
    overflowToDisk="true"
/>

<!-- 自定义缓存配置 -->
<cache name="myCache"
    maxElementsInMemory="2000"
    eternal="false"
    timeToIdleSeconds="300"
```

```
        timeToLiveSeconds="600"
        overflowToDisk="true"
    />
</ehcache>
```

步骤三：开启二级缓存、指定二级缓存驱动类

在 hibernate.cfg.xml 中开启二级缓存，并指定二级缓存驱动类，代码如下：

```
<?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>
        <!-- 数据库连接信息，根据自己的数据库进行配置 -->
        <property name="connection.url">
            jdbc:oracle:thin:@localhost:1521:xe
        </property>
        <property name="connection.username">lhh</property>
        <property name="connection.password">123456</property>
        <property name="connection.driver_class">
            oracle.jdbc.OracleDriver
        </property>

        <!-- Hibernate 配置信息 -->
        <!-- dialect 方言，用于配置生成针对哪个数据库的 SQL 语句 -->
        <property name="dialect">
            <!-- 方言类，Hibernate 提供的，用于封装某种特定数据库的方言 -->
            org.hibernate.dialect.OracleDialect
        </property>
        <!-- Hibernate 生成的 SQL 是否输出到控制台 -->
        <property name="show_sql">true</property>
        <!-- 将 SQL 输出时是否格式化。为了方便截图，我将其设置为 false -->
        <property name="format_sql">false</property>

        <!-- 开启二级缓存 -->
        <property name="hibernate.cache.use_second_level_cache">
            true
        </property>
        <!-- 指定所采用的二级缓存驱动类 -->
        <property name="hibernate.cache.provider_class">
            org.hibernate.cache.EhCacheProvider
        </property>

        <!-- 声明映射关系文件 -->
        <mapping resource="com/tarena/entity/Emp.hbm.xml" />
        <mapping resource="com/tarena/entity/Account.hbm.xml" />
        <mapping resource="com/tarena/entity/Service.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

步骤四：设置缓存策略

在 Emp.hbm.xml 中，指定二级缓存策略为只读的，代码如下：

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
```

```

<!--//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<!-- 配置实体类和表的关系 -->
<class name="com.tarena.entity.Emp" table="emp">

    <!--
        开启二级缓存，指定二级缓存策略。
        可以用 region 属性指定自定义的缓存设置。
    -->
    <cache usage="read-only" />

    <!-- 配置主键属性和字段的关系 -->
    <id name="id" type="integer" column="id">
        <!-- 用来指明主键的生成方式 -->
        <generator class="sequence">
            <!-- 指定用于生成主键的 sequence -->
            <param name="sequence">emp_seq</param>
        </generator>
    </id>

    <!-- 配置实体类中属性与表中字段的关系 -->
    <property name="name"
        type="string" column="name"/>
    <property name="age"
        type="integer" column="age"/>
    <property name="salary"
        type="double" column="salary"/>
    <property name="birthday"
        type="date" column="birthday"/>
    <property name="lastLoginTime"
        type="timestamp" column="last_login_time"/>
    <property name="marry"
        type="yes no" column="marry"/>
</class>
</hibernate-mapping>

```

步骤五：编写查询方法，测试二级缓存

在 com.tarena.test 包下，创建一个测试类 TestSecondCache，增加一个测试二级缓存的方法，代码如下：

```

package com.tarena.test;

import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Emp;
import com.tarena.util.HibernateUtil;

public class TestSecondCache {

    /**
     * 二级缓存
     */
    @Test
    public void test1() {
        Session session1 = HibernateUtil.getSession();
        Emp e1 = (Emp) session1.get(Emp.class, 321);
    }
}

```

```

        System.out.println(e1.getName());

        System.out.println("-----");

        Session session2 = HibernateUtil.getSession();
        Emp e2 = (Emp) session2.get(Emp.class, 321);
        System.out.println(e2.getName());

        session1.close();
        session2.close();
    }

}

```

步骤六：测试

执行 test1(), 控制台输出结果如下图, 可见第二次查询没有访问数据库, 是二级缓存存在的缘故:

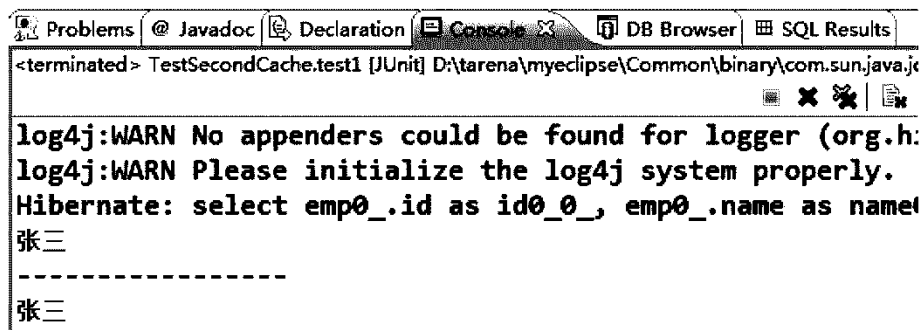


图-23

步骤七：管理二级缓存

二级缓存是 SessionFactory 级缓存, 由它负责管理, 因此需要获取到 SessionFactory 才能管理二级缓存, 我们先在 HibernateUtil 中增加获取 SessionFactory 的方法, 代码如下:

```

package com.tarena.util;

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

public class HibernateUtil {
    private static SessionFactory sessionFactory;

    static {
        // 加载 Hibernate 主配置文件
        Configuration conf = new Configuration();
        conf.configure("/hibernate.cfg.xml");
        sessionFactory = conf.buildSessionFactory();
    }

    /**
     * 创建 session
     */
    public static Session getSession() {
        return sessionFactory.openSession();
    }
}

```

```
/**
 * 返回 SessionFactory
 */
public static SessionFactory getSessionFactory() {
    return sessionFactory;
}

public static void main(String[] args) {
    System.out.println(getSession());
}
}
```

修改 test1()方法，在第二次查询前清理二级缓存，代码如下：

```
package com.tarena.test;

import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Emp;
import com.tarena.util.HibernateUtil;

public class TestSecondCache {

    /**
     * 二级缓存
     */
    @Test
    public void test1() {
        Session session1 = HibernateUtil.getSession();
        Emp e1 = (Emp) session1.get(Emp.class, 321);
        System.out.println(e1.getName());

        System.out.println("-----");

        HibernateUtil.getSessionFactory().evict(Emp.class);

        Session session2 = HibernateUtil.getSession();
        Emp e2 = (Emp) session2.get(Emp.class, 321);
        System.out.println(e2.getName());

        session1.close();
        session2.close();
    }
}
```

再次执行 test1()，控制台输出结果如下图，由于清理了二级缓存，因此第二次查询时访问了数据库。

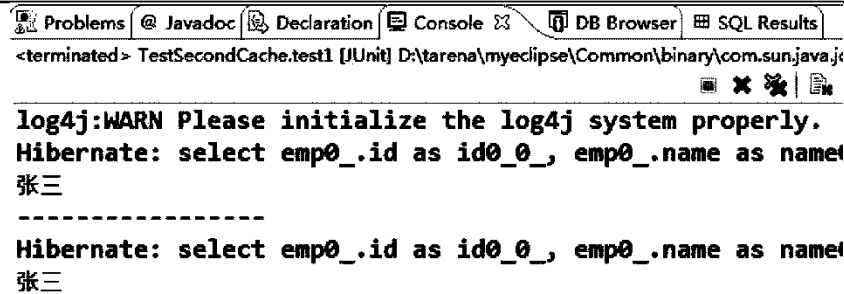


图-24

• 完整代码

以下是本案例的完整代码。

其中缓存配置文件 ehcache.xml 完整代码如下：

```
<ehcache>
  <!--
    缓存到硬盘时的缓存路径，java.io.tmpdir 表示
    系统默认缓存路径。
  -->
  <diskStore path="java.io.tmpdir"/>

  <!--
    默认缓存配置。
    maxElementsInMemory: 二级缓存可容纳最大对象数。
    eternal: 是否保持二级缓存中对象不变。
    timeToIdleSeconds:
      允许对象空闲的时间，即对象最后一次访问起，超过该时间即失效。
    timeToLiveSeconds:
      允许对象存活的时间，即对象创建起，超过该时间即失效。
    overflowToDisk:
      内存不足时，是否允许使用硬盘缓存，写入路径参见 diskStore。
  -->
  <defaultCache
    maxElementsInMemory="300"
    eternal="false"
    timeToIdleSeconds="120"
    timeToLiveSeconds="300"
    overflowToDisk="true"
  />

  <!-- 自定义缓存配置 -->
  <cache name="myCache"
    maxElementsInMemory="2000"
    eternal="false"
    timeToIdleSeconds="300"
    timeToLiveSeconds="600"
    overflowToDisk="true"
  />
</ehcache>
```

hibernate.cfg.xml 完整代码如下：

```
<?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>
  <!-- 数据库连接信息，根据自己的数据库进行配置 -->
  <property name="connection.url">
    jdbc:oracle:thin:@localhost:1521:xe
  </property>
  <property name="connection.username">lhh</property>
  <property name="connection.password">123456</property>
  <property name="connection.driver_class">
    oracle.jdbc.OracleDriver
  </property>

  <!-- Hibernate 配置信息 -->
  <!-- dialect 方言，用于配置生成针对哪个数据库的 SQL 语句 -->
  <property name="dialect">
    <!-- 方言类，Hibernate 提供的，用于封装某种特定数据库的方言 -->
    org.hibernate.dialect.OracleDialect
  </property>
  <!-- Hibernate 生成的 SQL 是否输出到控制台 -->
  <property name="show_sql">true</property>
  <!-- 将 SQL 输出时是否格式化。为了方便截图，我将其设置为 false -->
  <property name="format_sql">false</property>

  <!-- 开启二级缓存 -->
  <property name="hibernate.cache.use second level cache">
    true
  </property>
  <!-- 指定所采用的二级缓存驱动类 -->
  <property name="hibernate.cache.provider_class">
    org.hibernate.cache.EhCacheProvider
  </property>

  <!-- 声明映射关系文件 -->
  <mapping resource="com/tarena/entity/Emp.hbm.xml" />
  <mapping resource="com/tarena/entity/Account.hbm.xml" />
  <mapping resource="com/tarena/entity/Service.hbm.xml" />
</session-factory>
</hibernate-configuration>

```

emp.hbm.xml 完整代码如下：

```

<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
  <!-- 配置实体类和表的关系 -->
  <class name="com.tarena.entity.Emp" table="emp">
    <!--
      开启二级缓存，指定二级缓存策略。
      可以用 region 属性指定自定义的缓存设置。
    -->
    <cache usage="read-only" />

    <!-- 配置主键属性和字段的关系 -->
    <id name="id" type="integer" column="id">
      <!-- 用来指明主键的生成方式 -->
      <generator class="sequence">
        <!-- 指定用于生成主键的 sequence -->
        <param name="sequence">emp seq</param>
      </generator>
    </id>

```

```
<!-- 配置实体类中属性与表中字段的关系 -->
<property name="name"
    type="string" column="name"/>
<property name="age"
    type="integer" column="age"/>
<property name="salary"
    type="double" column="salary"/>
<property name="birthday"
    type="date" column="birthday"/>
<property name="lastLoginTime"
    type="timestamp" column="last_login_time"/>
<property name="marry"
    type="yes no" column="marry"/>
</class>
</hibernate-mapping>
```

TestSecondCache 完整代码如下：

```
package com.tarena.test;

import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Emp;
import com.tarena.util.HibernateUtil;

public class TestSecondCache {

    /**
     * 二级缓存
     */
    @Test
    public void test1() {
        Session session1 = HibernateUtil.getSession();
        Emp e1 = (Emp) session1.get(Emp.class, 321);
        System.out.println(e1.getName());

        System.out.println("-----");
        HibernateUtil.getSessionFactory().evict(Emp.class);

        Session session2 = HibernateUtil.getSession();
        Emp e2 = (Emp) session2.get(Emp.class, 321);
        System.out.println(e2.getName());

        session1.close();
        session2.close();
    }
}
```

11. 使用查询缓存

- **问题**

在查询多条员工数据时，使用查询缓存，缓存查询的 HQL，使得再次使用同样 HQL 查询时不必重新访问数据库。

- **方案**

查询缓存的使用步骤是

- 1) 开启二级缓存
- 2) 在 hibernate.cfg.xml 中开启查询缓存
- 3) 在查询之前, 开启查询缓存

• 步骤

实现此案例需要按照如下步骤进行。

步骤一：开启二级缓存

开启二级缓存, 由于上个案例已经完成, 这一步就可以省略。但是要注意, 查询缓存是基于二级缓存的, 使用查询缓存的前提是开启二级缓存。

步骤二：开启查询缓存

在 hibernate.cfg.xml 中开启查询缓存, 代码如下：

```
<?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>
    <!-- 数据库连接信息, 根据自己的数据库进行配置 -->
    <property name="connection.url">
      jdbc:oracle:thin:@localhost:1521:xe
    </property>
    <property name="connection.username">lhh</property>
    <property name="connection.password">123456</property>
    <property name="connection.driver_class">
      oracle.jdbc.OracleDriver
    </property>

    <!-- Hibernate 配置信息 -->
    <!-- dialect 方言, 用于配置生成针对哪个数据库的 SQL 语句 -->
    <property name="dialect">
      <!-- 方言类, Hibernate 提供的, 用于封装某种特定数据库的方言 -->
      org.hibernate.dialect.OracleDialect
    </property>
    <!-- Hibernate 生成的 SQL 是否输出到控制台 -->
    <property name="show_sql">true</property>
    <!-- 将 SQL 输出时是否格式化。为了方便截图, 我将其设置为 false -->
    <property name="format_sql">false</property>

    <!-- 开启二级缓存 -->
    <property name="hibernate.cache.use_second_level_cache">
      true
    </property>
    <!-- 指定所采用的二级缓存驱动 -->
    <property name="hibernate.cache.provider_class">
      org.hibernate.cache.EhCacheProvider
    </property>

    <!-- 开启查询缓存 -->
```

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

<!-- 声明映射关系文件 -->
<mapping resource="com/tarena/entity/Emp.hbm.xml" />
<mapping resource="com/tarena/entity/Account.hbm.xml" />
<mapping resource="com/tarena/entity/Service.hbm.xml" />
</session-factory>
</hibernate-configuration>
```

步骤三：查询前开启查询缓存

在 TestSecondCache 中，增加测试查询缓存的方法，使用相同的 HQL 执行 2 次查询，每次查询前都设置开启查询缓存，代码如下：

```
package com.tarena.test;

import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Emp;
import com.tarena.util.HibernateUtil;

public class TestSecondCache {

    /**
     * 二级缓存
     */
    @Test
    public void test1() {
        Session session1 = HibernateUtil.getSession();
        Emp e1 = (Emp) session1.get(Emp.class, 321);
        System.out.println(e1.getName());

        System.out.println("-----");
        HibernateUtil.getSessionFactory().evict(Emp.class);

        Session session2 = HibernateUtil.getSession();
        Emp e2 = (Emp) session2.get(Emp.class, 321);
        System.out.println(e2.getName());

        session1.close();
        session2.close();
    }

    /**
     * 查询缓存
     */
    @Test
    public void test2() {
        Session session = HibernateUtil.getSession();

        String hql = "from Emp";
        Query query = session.createQuery(hql);
        // 开启查询缓存
        query.setCacheable(true);
        List<Emp> emps = query.list();
        for(Emp e : emps) {
            System.out.println(e.getId() + " " + e.getName());
        }
    }
}
```

```

    }

    System.out.println("-----");

    hql = "from Emp";
    query = session.createQuery(hql);
    // 开启查询缓存
    query.setCacheable(true);
    emps = query.list();
    for(Emp e : emps) {
        System.out.println(e.getId() + " " + e.getName());
    }

    session.close();
}

}

```

步骤四：测试

执行 test2(), 控制台输出结果如下图, 可见在使用相同 HQL 查询时, 第二次查询不必再次访问数据库。

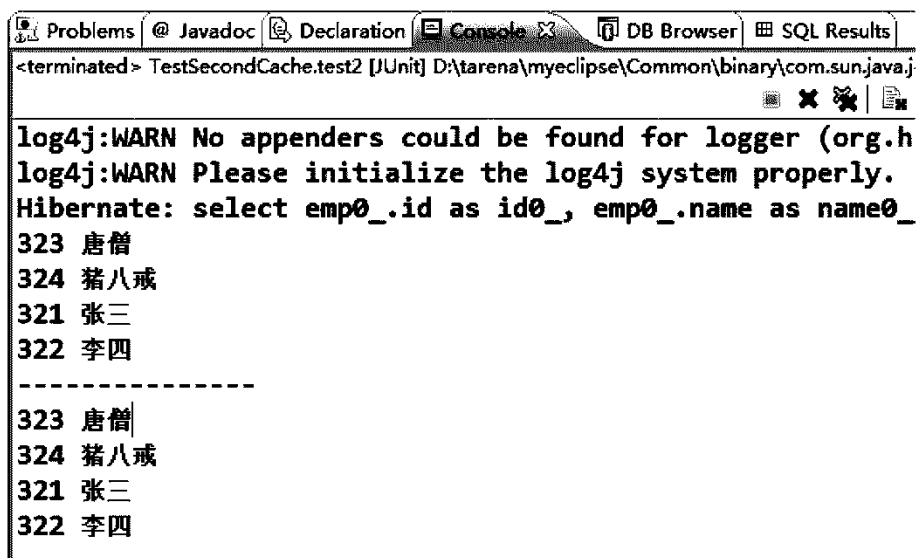


图-25

步骤五：管理查询缓存

修改 test2(), 在第二次查询之前清理查询缓存, 代码如下:

```

package com.tarena.test;

import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Emp;
import com.tarena.util.HibernateUtil;

public class TestSecondCache {

    /**

```

```

* 二级缓存
*/
@Test
public void test1() {
    Session session1 = HibernateUtil.getSession();
    Emp e1 = (Emp) session1.get(Emp.class, 321);
    System.out.println(e1.getName());

    System.out.println("-----");
    HibernateUtil.getSessionFactory().evict(Emp.class);

    Session session2 = HibernateUtil.getSession();
    Emp e2 = (Emp) session2.get(Emp.class, 321);
    System.out.println(e2.getName());

    session1.close();
    session2.close();
}

/**
* 查询缓存
*/
@Test
public void test2() {
    Session session = HibernateUtil.getSession();

    String hql = "from Emp";
    Query query = session.createQuery(hql);
    // 开启查询缓存
    query.setCacheable(true);
    List<Emp> emps = query.list();
    for(Emp e : emps) {
        System.out.println(e.getId() + " " + e.getName());
    }

    System.out.println("-----");

    HibernateUtil.getSessionFactory().evictQueries();

    hql = "from Emp";
    query = session.createQuery(hql);
    // 开启查询缓存
    query.setCacheable(true);
    emps = query.list();
    for(Emp e : emps) {
        System.out.println(e.getId() + " " + e.getName());
    }

    session.close();
}
}

```

再次执行 test2(), 控制台输出结果如下图, 可见第二次查询也访问了数据库, 主要是清理了查询缓存中数据的缘故。

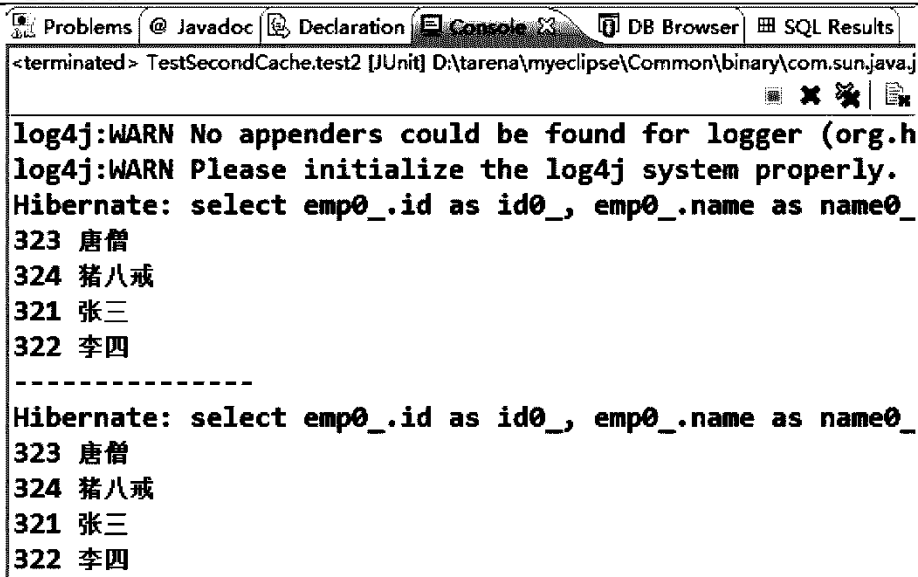


图-26

• 完整代码

以下为本案例的完整代码。

其中 Hibernate.cfg.xml 完整代码如下：

```
<?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>
    <!-- 数据库连接信息，根据自己的数据库进行配置 -->
    <property name="connection.url">
      jdbc:oracle:thin:@localhost:1521:xe
    </property>
    <property name="connection.username">lhh</property>
    <property name="connection.password">123456</property>
    <property name="connection.driver.class">
      oracle.jdbc.OracleDriver
    </property>

    <!-- Hibernate 配置信息 -->
    <!-- dialect 方言，用于配置生成针对哪个数据库的 SQL 语句 -->
    <property name="dialect">
      <!-- 方言类，Hibernate 提供的，用于封装某种特定数据库的方言 -->
      org.hibernate.dialect.OracleDialect
    </property>
    <!-- Hibernate 生成的 SQL 是否输出到控制台 -->
    <property name="show sql">true</property>
    <!-- 将 SQL 输出时是否格式化。为了方便截图，我将其设置为 false -->
    <property name="format sql">false</property>

    <!-- 开启二级缓存 -->
    <property name="hibernate.cache.use_second_level_cache">
      true
    </property>
    <!-- 指定所采用的二级缓存驱动 -->
    <property name="hibernate.cache.provider class">
      org.hibernate.cache.EhCacheProvider
```

```
</property>
<!-- 开启查询缓存 -->
<property name="hibernate.cache.use query cache">
    true
</property>

<!-- 声明映射关系文件 -->
<mapping resource="com/tarena/entity/Emp.hbm.xml" />
<mapping resource="com/tarena/entity/Account.hbm.xml" />
<mapping resource="com/tarena/entity/Service.hbm.xml" />
</session-factory>
</hibernate-configuration>
```

TestSecondCache 完整代码如下：

```
package com.tarena.test;

import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.junit.Test;
import com.tarena.entity.Emp;
import com.tarena.util.HibernateUtil;

public class TestSecondCache {

    /**
     * 二级缓存
     */
    @Test
    public void test1() {
        Session session1 = HibernateUtil.getSession();
        Emp e1 = (Emp) session1.get(Emp.class, 321);
        System.out.println(e1.getName());

        System.out.println("-----");
        HibernateUtil.getSessionFactory().evict(Emp.class);

        Session session2 = HibernateUtil.getSession();
        Emp e2 = (Emp) session2.get(Emp.class, 321);
        System.out.println(e2.getName());

        session1.close();
        session2.close();
    }

    /**
     * 查询缓存
     */
    @Test
    public void test2() {
        Session session = HibernateUtil.getSession();

        String hql = "from Emp";
        Query query = session.createQuery(hql);
        // 开启查询缓存
        query.setCacheable(true);
        List<Emp> emps = query.list();
        for(Emp e : emps) {
            System.out.println(e.getId() + " " + e.getName());
        }

        System.out.println("-----");
        HibernateUtil.getSessionFactory().evictQueries();
    }
}
```

```
hql = "from Emp";
query = session.createQuery(hql);
// 开启查询缓存
query.setCacheable(true);
emps = query.list();
for(Emp e : emps) {
    System.out.println(e.getId() + " " + e.getName());
}

session.close();
}
}
```

课后作业

1. Hibernate 一对多级联修改/删除时，需要注意哪些问题
2. Hibernate 有哪几种查询方式
3. 请简述 Hibernate 一级缓存和二级缓存的区别和联系